BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation:

LAMP: TOOLS FOR CREATING APPLICATION-SPECIFIC FPGA COPROCESSORS

By

THOMAS DAVID VANCOURT

B.S., Cornell University, 1978
M.S., Boston University, 2001

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2006

APPROVED BY

First reader:	____________________________________ Prof. Martin Herbordt, Ph.D. Professor of Electrical and Computer Engineering
Second reader:	____________________________________ Prof. Roscoe Giles, Ph.D. Professor of Electrical and Computer Engineering
Third reader:	____________________________________ Prof. Allyn Hubbard, Ph.D. Professor of Electrical and Computer Engineering
Fourth reader:	____________________________________ Prof. Sandor Vajda, Ph.D. Professor of Biomedical Engineering
Chair:	____________________________________ Prof. Wei Qin, Ph.D. Professor of Electrical and Computer Engineering

LAMP: TOOLS FOR CREATING APPLICATION-SPECIFIC FPGA COPROCESSORS

(Order No.)

THOMAS DAVID VANCOURT

Boston University, College of Engineering 2006

Major Professor: Martin C. Herbordt, Ph.D.

ABSTRACT

Field Programmable Gate Arrays (FPGAs) have begun to appear as accelerators for general computation. Their potential for massive parallelism, high on-chip memory bandwidth, and customizable interconnection networks all contribute to demonstrated 100-1000× increases in application performance relative to current PCs. FPGA coprocessors have been available in niche markets for years, and are now appearing in mainstream supercomputers from vendors including Cray and Silicon Graphics.

Available development tools do not address developers of computing applications, however. Traditional FPGA design tools meet the gate-level needs of logic designers, but present a computing model that vanishingly few software developers can use. Likewise, logic designers understand logic structures for high computing performance, but rarely know the biology, biochemistry, or other applications that need acceleration. Logic designers and application developers must both participate in creating efficient, useful accelerators, but their different kinds of participation are not supported by current tools.

This work presents two major sets of contributions. The first is proof by example that FPGAs give 100-1000× speedups for large families of applications in bioinformatics and computational biology (BCB), including sequence alignment, molecule docking, and string analysis. These demonstrations also provide the beginnings of a library of reusable computing structures.

The second set of contributions appear as novel features of accelerator design tools based on Logic Architecture by Model Parameterization (LAMP). The LAMP tools address broad, customizable families of applications, not point solutions to narrow problem statements. LAMP also separates the logic designers, who create efficient hardware computing structures, from the application specialists who tailor the accelerator to specific members of the application family. This separation enables accelerator hardware customization without access to hardware design skills. Finally, LAMP provides mechanisms for automating the tradeoff between complexity and quantity of parallel processing elements (PEs), allowing fewer large PEs or larger numbers of small ones, subject to the the FPGA's resource constraints. This creates a unique ability to allocate the FPGA's computing resources differently for each member of an application family, according to the datatypes and functions specific to that family member. Performance results based on prototype LAMP tools are presented, using sample BCB applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
b	bit
B	byte
BCB	Bioinformatics and Computational Biology
BLAST	Basic Local Alignment Search Tool [Alt90]
CASE	Computer Aided Software Engineering
CLAMP	Convenience LAMP, secondary text format for LAMP too input
CPU	Central Processing Unit, the functional core of a PC, one of many in an MPP
DIP	Dependency Inversion Principle, an object oriented design principle stating importance of component interfaces over component implementations.
DNA	Deoxyribose Nucleic Acid
DSP	Digital Signal Processor.
EDA	Electronic Design Automation
FFT	Fast Fourier Transform
FLOPs	Floating point operations per second

FPGA	Field Programmable Gate Array
Gb	Gigabits, 2^{30} (~10^9) bits
GB	Gigabytes, 2^{30} (~10^9) bytes
GFLOPs	Giga FLOPS, 10^9 floating point operations per second
GPGPU	General Purpose computation on GPU
GPU	Graphics Processing Unit
GROMACS	Groningen Machine for Chemical Simulations, an open-source molecular modeling program
GUI	Graphical User Interface
HDL	Hardware Design Language, for example VHDL or Verilog
HLL	High Level Language, specifically for software design, as opposed to an HDL
IDE	Interactive Development Environment
IDL	Interface Definition Language [OMG01]
IP	Intellectual Property
IUPAC	International Union of Physical and Analytical Chemists
JTAG	Joint Test Action Group. IEEE standard 1149.1
Kb	Kilobits, 2^{10} (~10^3) bits
KB	Kilobytes, 2^{10} (~10^3) bytes

LAMP	Logic Architecture Model Parameterization, novel EDA tool set
LAMPML	LAMP Markup Language, the XML-based markup language used as the intermediate representation in the LAMP tool set
LSB	Least Significant Bit
LSP	Liskov Substitutibility Principle, stating semantic condition under which subclass objects can safely be used by superclass references
LUT	Look Up Table, the hardware element that underlies programmable functions in FPGAs.
MAC	Media Access Control
Mb	Megabits, 2^{20} ($\sim 10^6$) bits
MB	Megabytes, 2^{20} ($\sim 10^6$) bytes
MPP	Massively Parallel Processor
MSB	Most Significant Bit
OMG	Object Management Group, maintainer of UML and other standards
OO	Object Oriented, or Object Orientation
OOD	Object Oriented Design
PAR	Place And Route
PC	Personal Computer
PCI	Peripheral Component Interconnect, a common system bus

PE	Processing Element
PFLOPs	Peta FLOPS, 10^{15} floating point operations per second
PHY	Physical level network protocol
RTL	Register Transfer Level.
SA-C	Single-Assignment C (pronounced “sassy”) [Boh01]
SGI	Silicon Graphics, Inc.
SIMD	Single Instruction, Multiple Data
SOM	Self Organizing Map
SPI	Service Provider Interface, a system-defined interface that an application must export
TB	Terabytes, 2^{40} ($\sim 10^{12}$) bytes
TFLOPs	Tera FLOPS, 10^{12} floating point operations per second
UI	User Interface
UML	Unified Modeling Language, system description notation standard [OMG03]
VHDL	VHSIC Hardware Description Language [IEEE02]
VLIW	Very Long Instruction Word
XML	eXtensible Markup Language [W3C04]

GLOSSARY

API	Application Programming Interface. The set of services defined by a system as utilities, callable by an application. cf. SPI.
bit file	Exact binary image containing the full set of programming information for a FPGA.
HLL	A high level language for software design, as opposed to an HDL for hardware design. Some languages, including Java/JHDL and C++/SystemC have been used for both purposes, so calling the language an HLL or HDL depends on the application.
indel	When comparing two strings, an indel represents an insertion into one or a deletion from the other, such as the letter *k* in a comparison of *tracker* and *tracer*.
JBITS	A Xilinx API for controlled access to the files containing FPGA bit patterns. This lets sophisticated users create custom design tools with complete control over FPGA resource allocation [Xil03].
SPI	Service Provider Interface. A set of interface definitions that must be implemented by an application, allowing its execution environment to access that application, using sequencing and control defined by the execution environment. Modern networking and user interface systems commonly define SPIs for input events, allowing callbacks into the application so that it can process specific kinds of service requests. cf. API.

1 RETHINKING TOOLS FOR FPGA-BASED COMPUTATION

Researchers in many fields have a seemingly insatiable need for high-speed computation. When single workstations lack the necessary computing power, users traditionally turn to super-computers and computing clusters for increased computing capacity. These create their own problems, however, so the search continues for affordable ways to put massive computing capability into more researchers' hands. Field programmable gate arrays (FPGAs) offer a promising approach to acceleration of a wide range of computing applications. Unlike fixed-function processors, FPGAs allow the developer to create computing structures customized to the application at hand. The FPGA contains a pool of uncommitted computing resources, which can be built into hundreds of task-specific processors. Properly configured, FPGA-based coprocessors have demonstrated 100-1000× speedups relative to PCs [Con04, Van04, Van04a, Van05].

The FPGA programming model is fundamentally different from the sequential-program model of computing. A stored program in a standard computer distributes the steps of an algorithm over time by ordering many successive uses of a few predefined functional units. An FPGA program distributes parts of an algorithm spatially by ordering concurrent use of hundreds or thousands of customized functional units. FPGAs have only recently emerged as computation accelerators from their origin as a replacement for discrete logic. As a result, the most popular hardware design languages (HDLs), VHDL and Verilog, still address the bit-level needs of the logic designer, at a semantic level below the barest layer of assembly programming. High level languages (HLLs) including

C, Java, and derivatives have been compiled into FPGA logic, but current commercial products [Mit05] claim only 10-30× typical performance increase over a standard CPU. Achieving the FPGAs' 100-1000× performance potential requires a new generation of tools for implementing application logic. That is the goal of this work: to create new programming tools able to exploit an FPGA's full capability, while presenting a familiar interface to the application specialist that has the computing needs.

FPGAs have long, successful history in digital signal processing (DSP) applications. Vendors have created FPGA design tools specifically for DSP applications [Xil02d]. The fundamental hypothesis of the current work is that FPGAs are ready to expand beyond this narrow domain, and into broad areas of computationally intensive applications. Although many application areas could benefit from FPGA-based acceleration, this research focuses on applications in bioinformatics and computational biology (BCB). In developing a number of BCB/FPGA applications, it became apparent that such applications tend to be used in many variant forms. Addressing the whole range of variants required insight into the science of each application and also manual development procedures impossible within the scopes of exiting tools.

Our tools address these observations with three central ideas. The first is that two skill sets are required for FPGA accelerator development, skills that rarely (if ever) occur together in one developer. The *application specialist* must bring broad, deep understanding of a BCB algorithm and its many different variations. A logic designer is also needed for casting the algorithm into terms that an FPGA can implement efficiently. The second idea is to extend the underlying HDL to allow data types and functional behavior to be expressed as component parameters, and to allow kinds of reuse not possible within the HDLs' current standard. The third idea is to create new optimization techniques that derive parallelism from the logic usage in application-specific processing elements, up to the capacity of the FPGA's fabric. LAMP tools automate maximal use of FPGA resources for each specific application, and allow larger FPGAs to improve performance without source changes to the application.

We have integrated these ideas into a proof-of-concept version of the Logic Architecture Model Parameterization (LAMP) design tools. With LAMP, a logic designer, working together with an application specialist, can create an application environment corresponding to a family of applications. Once created, many application specialists can then use the environment, independent of the logic designer, to create FPGA accelerator instantiations specific to the details of their respective applications. In the language of software design patterns, LAMP allow a logic designer to create accelerators with detailed behavior phrased as a parameter, to *"implement the invariant parts of an application once, and leave it up to the [application specialists] to implement the behavior that can vary"* [Gam95].

This first chapter starts by describing the FGPA's basic capabilities and the kinds of problems best suited to FPGA-based computation. Based on case studies created during this research, this chapter develops the idea of an *application family.* Family members have the same flow of data, memory access patterns, and synchronization structures, but differ in the details of data types and specific functions. This also introduces the different participants in designing FPGA-based accelerators: the *application specialist* with a computation need, and the *logic designer* able to create efficient computing structures using FPGA logic. Conflicting requirements for FPGA design tools arise from these basic facts of technology, applications, and participants. The result of this research is a set of novel tools that resolve these conflicts, enabling kinds of FPGA applications and application families that are not feasible with existing tools.

The second chapter presents a set of sample applications that we created in order to evaluate state of the art FPGA design language, and to gain understanding about the needs of FPGA-based application acceleration. This chapter starts with a brief description of factors that predispose an application to successful FPGA implementation. Next, this chapter examines a set of case studies in biochemistry and computational biology (BCB) that create the conceptual basis for the current work. These case studies are interesting in themselves, but also demonstrate the application demands that current HDLs do not address. In particular, these case studies demonstrate the idea of *application families*, groups of different applications that share a common computation structure. This chapter ends with an examination of features common across different application families, despite the very different structures of their computations. These common features suggest a set of requirements for design tools aimed specifically at application acceleration.

Chapter three examines previous work, and shows how current logic design software fails to meet the needs of accelerator design. It opens with a brief survey of the hardware systems that have been used for BCB computations. The first part of the survey examines a representative sample of hardware systems that have been demonstrated, or that are emerging as platforms for high performance computing. These systems all suffer problems in technology or usability, and FPGAs are shown to address many of these problems. The second section of the survey examines major categories of logic design tools that have been used for FPGA applications. These cover a number of different approaches to software design and logic specification, but generally present computing paradigms that do not match the needs of high-performance computation. Chapter three ends with a discussion of object-oriented (OO) programming languages, with emphasis on the ways in which hardware environments demand reconsideration of traditional OO semantics.

Chapter four summarizes the requirements for accelerator design tools, as distinct from any other logic design task, in terms of the background information provided in chapters two and three. It ends with a discussion of design decisions behind the implementation of the LAMP tool set in its current form.

Chapter five explains the LAMP tool flow, the steps taken by the different people involved in the design process, as well as the technical content of the tools and data involved. Next, this chapter provides a description of the LAMP tool features, showing how the meet they demands and implement the concepts laid out in previous chapters. This chapter presents the details of defining an application family using the LAMP tools, and how the LAMP specifications translate into synthesizable FPGA designs.

Chapter six summarizes the results of this research: the approach is effective in addressing families of application accelerators, and the LAMP tools are effective in implementing that approach. The chapter recapitulates the basic problems in creating application accelerators using FPGA coprocessors. It then shows how the LAMP tools address those problems, and summarizes the novel features of the LAMP tools. Finally, it suggests areas in which the LAMP tools can be extended, to improve their usability, robustness, and semantic richness.

In order to keep the discussions of chapters 1-6 concise, significant but lengthy supporting information has been moved to the appendices. Appendix A describes the LAMPML design language in detail. This is an XML-based representation, and is the basic definition of LAMP tools. Although XML is a poor interface for application development, it is a clear and well-supported representation for machine interpretation. As such, it is a useful intermediate, internal representation of the LAMP models. All other representations of LAMP applications rely on the semantics of this format.

Appendix B lays out the notion of a computation’s *growth law*, the algebraic description of the resources needed for computing arrays of different structure and connectivity. This concept is central to the idea of repeating arrays of processing elements, since it specifies permissible and desirable array sizes. Combined with information about application details and FGPA resource limits, this provides the basic idea underlying automatic sizing of computation arrays.

The final appendix, C, analyses a detailed example of LAMP usage, including the logic for sizing computation and memory arrays according to available FPGA resources. This omits the VHDL code that would be needed for synthesis, but demonstrates how LAMP features are used in defining efficient, configurable application accelerators.

The rest of this introduction starts with a basic statement of this work’s goals. It briefly describes how FGPAs differ from traditional processors, in both their strengths and their weaknesses. This leads to a discussion of the kinds of computations suited to the FPGA platform, and sketches of case studies of applications that take advantage for FPGAs’ unique strengths. Basic features of FPGAs and lessons from the case studies present contradictory demands to the tool developer, described in section 1.5. Section 1.6 outlines LAMP’s answers to these conflicting requirements, and section 1.7 summarizes the basic contributions of this research. This introduction ends with a description of typographic and diagrammatic conventions used through the other chapters of this presentation.

1.1 Summary of goals

The primary objective of this work is to examine the factors that make existing logic design tools unsuitable for developing FPGA-based application accelerators, and to demonstrate prototype tools that fix these defects. The tools' major goals include support for:

· 100-1000× acceleration of selected application families relative to PC performance, through application-specific configuration of processing and communication elements, not through incremental improvements in existing computing structures.

· Families of applications, related by common control and communication structures but differing in details of data types and "leaf" computations.

· Reusability of FPGA accelerators across different FPGA chips and different applications, with modest incremental cost in design for reuse.

· Automated exploitation of computing resources, by adapting the degree of parallelism to the problem at hand and resources available, as opposed to fixed-size processing arrays.

· Effective collaboration between hardware designers who understand efficient computing structures and application specialists who know the details of each different usage of the acceleration hardware.

· Application specialists' ability to modify an accelerator's data types and operations within the framework of the application family, without direct support from hardware designers.

Having stated the goals of this research, it is equally important to set constraints that bound the scope of the study. The problem space as a whole is huge, and requires a focus of attention that necessarily sets aside other considerations. In particular, the following are *not* among the goals of this work:

· Point solutions to individual problems. Feasible hardware accelerators must be widely applicable, but still deliver the performance of custom designs across many users' different forms of a given computation.

· Complete generality of solution. The current work addresses only problem families where dramatic increases in performance warrant the cost and effort of using an FPGA accelerator. Many problems do not embody the features that work well with FPGA-based computation, and no effort is made to fit them to an inappropriate platform.

1.2 Emergence of FPGA computing

FPGAs have been available since the 1980s, but have only recently started to become popular as computation accelerators. In order to understand why, it is necessary to understand something about their basic technology. That technology explains the FGPAs' strengths and limitations as application accelerators, and shows why FPGA programming is so different from traditional kinds of application programming.

1.2.1 FPGA technology

FPGAs are reprogrammable chips containing large numbers of configurable logic gates, registers, and interconnections. FPGA programming means defining the bit-level configuration of these resources in order to create a circuit that implements a desired function. The FPGAs of interest store their configuration data in static RAM cells distributed across the chip, so they can be reused indefinitely for different computations defined by different logic configurations. Recent families FPGAs also contain dedicated multipliers and on-chip RAMs, along with vastly larger amounts of the traditional logic resources.

These are the same logical building blocks that go into a CPU's arithmetic units, registers, caches, and other subsystems. The smaller FPGAs of the past were sometimes used to created dedicated processing elements, especially for digital signal processing (DSP) applications. CPU designers had a long head start, however, and cheap, fast processors gave adequate performance at lower cost in all but a few applications. Still, FPGAs grow [But03, Tre04] according to a loose interpretation of Moore's law, a doubling of capacity about every 18 months [Pag04]. Their overall capacity has reached a threshold at which they have begun to look cost effective for general computation. Recent product announcements support this claim: Cray [Cra05] and Silicon Graphics [Sil04] have new FPGA-based add-ons for their standard supercomputer products. These main-stream vendors joins specialty vendors that have sold FPGA based processors or accelerators for many years, including Star Bridge Systems [Sta04], SRC Computers [SRC05], Nallatech [Nal05], and Annapolis Microsystems [Ann05].

FPGAs differ from application-specific integrated circuits (ASICs) in one basic feature: the ASIC's capabilities fixed when the chip is fabricated, but the FPGA's behavior is set differently by each application's configuration data. As a result, FPGAs usually require more transistors and more power for any given operation, and typically do not achieve the performance that ASICs get from transistor-level tuning. In their favor, FPGAs are commodity parts. They allow custom logic implementations without the expense and development time of custom integrated circuits. They amortize the one general-purpose chip's development costs across many applications. They also avoid obsolescence, since FPGA vendors offer new or enhanced products at frequent intervals. ASICs upgrades usually require a new investment of some or all of their original development cost, and may cost more than the original as costs of fabrication masks continue to rise.

1.2.2 FPGA application development

Creating an FPGA application (often called *FPGA programming*) differs fundamentally from programming of standard CPUs. The stored-program model uses a CPU of fixed configuration, with fixed memory, registers, arithmetic or logical units, and fixed connections between them. Programming consists of writing instructions that sequentially reuse these resources to perform a desired algorithm. Developing an applicaiton for an FPGA means defining the set of function units, registers, and interconnections that create a hardware implementation of the algorithm. Individual CPUs execute instructions one at a time (or appear to), but all of an FPGA's logic executes in parallel. This is where the FPGA's potential for computing performance arises: hundred or thousands of concurrent operations, contention free access to hundreds of memory busses and registers, and communication networks with arbitrary topology and nanosecond latency. Standard hardware implementation techniques also contribute to FPGAs' performance advantage: massive pipelining can often eliminate costs due to load/store operations, memory indirection, and loop overhead.

Most of the FPGA's logic resources operate at the bit level, so they must be ganged to form the larger data elements that represent an application's logic: integers, address, and other values. Programmers familiar with a few fixed data sizes find this surprising in two ways. The first is that every word size must be specified individually. The second and bigger surprise is the range of word sizes that can be chosen, from one or two bits to thousands. Dozens or hundreds of register arrays, functional units, and on-chip memory banks can be configured with the same flexibility, but with the same need for low-level design specification.

Tools for creating FPGA-based designs have not caught up to the abrupt opening of this new field of FPGA usage. The two most popular HDLs for large FPGA-based logic designs are Verilog and VHDL. They have some syntactic resemblance to the HLLs C and Ada, respectively, but profoundly different semantics.

One difference between HDLs and conventional HLLs is that all statements in an HDL program execute concurrently, except for a few special constructs. The other major difference is that HDL programmers must create function units and registers in their custom processors, instead of using the ones provided by a CPU designer.

HDLs offer low levels of semantic representation, well below the assembly level of standard CPUs. They meet the needs of logic designers dealing with pervasive parallelism, gate-level resource allocation, and harsh timing requirements, not the needs of scientific application developers dealing with complex algorithms. HDLs expose nearly all of the FPGA's capability and concurrency, but also expose the circuit level complexities that have traditionally distinguished logic design from software development. They are effectively unusable by the large majority of people developing compute-intensive applications.

1.2.3 FPGAs as computation accelerators

Despite at least fifteen years of discussion [Gra89], FPGAs have had limited acceptance as platforms for general computing. One reason is that logic design is rare skill compared to standard programming, leading to the objservation that that "*10×-100× of performance ... has been at the cost of 10×-100× increase in difficulty in application development.*" [Gok00]. Another reason is that, FGPAs did not have enough computational capacity to justify the difficulty in using them, except in research or niche applications. Several factors have recently combined to make FPGAs more attractive as computation engines, however.

The most obvious factor is recent, rapid increases in the amount of computing capacity per FPGA. The Xilinx Virtex-II Pro family, for example, features up to 444 hardware multipliers, 99K programmable logic elements, 7.9Mb of block RAM, twenty multi-gigabit IO ports, and over 1000 uncommitted IO pins [Xil04]. Newer chip families have even higher capacity in many of these dimensions. Because FPGAs are now said to be

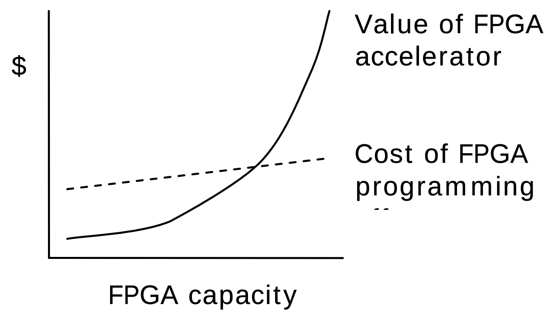


Figure 1. FPGA acceleration: Qualitative cost and value of application development as

drivers of chip process development, a role formerly held by DRAMs [But03, Tre03], one should expect exponential growth similar to Moore's Law.

The large capacity of current and anticipated FPGAs has a surprising benefit, shown qualitatively in Figure 1. One might expect FPGA applications to become increasingly complex, in proportion with the number of programmable elements per chip. If true, the performance per unit of development effort would have remained roughly constant, or perhaps worsened due to $O(N^2)$ pairwise interaction effects. This is not necessarily the case, however. Experience shows that many FPGA-based computations are repetitive, scalable arrays of computing elements. Maximum array size increases with FPGA capacity, and potential application performance increases directly with array size. Array design effort, however, is largely independent of array size. The result is that larger FPGAs have the potential for higher performance per unit of application development effort, because multiple PEs multiply the value of the effort in designing the PE. Larger FPGAs only increase the potential number of PEs, and further increasing the value of the development effort. Superlinear increases in performance are also possible, if larger FPGAs can reduce the overhead incurred by partitioning the problem into pieces that fit into the FPGA. Also, newer generations of FPGAs sometimes introduce dedicated functions such as multiplication blocks, and incorporate other technological changes that increase performance.

1.2.4 Acceleration's possibilities and limits

It is commonly observed that a small portion of any program consumes the majority of its computation cycles. This is often true in BCB applications and becomes more true over time, as new applications reduce older ones to individual steps. For example, alignment of biological sequences is a common, basic operation. It is repeated many times in finding a best match within some set of sequences. Best-match searches happen repeatedly in building a phylogenetic tree, and many individual trees combine to make a consensus tree [Fel04]. Consensus trees of multiple species, in turn, are combined in studying cophylogeny of host/parasite or other biological relationships [Pag03]. Each layer of algorithms multiplies the time saved by accelerating alignments; each additional layer of algorithms multiplies the time savings by another factor.High overall performance improvement often comes from accelerating small program segments. This creates a good match to the physical relationship between a CPU and an FPGA accelerator. Multi-GB memories are typical for current processors, and are able to hold programs at least 10^8 MB in length. FPGAs, however, contain only about 10^4 to 10^5 programmable logic elements. Without trying to equate some number of logic elements to a byte of program code, the FPGA clearly has far smaller programmable capacity than the CPU. Assigning a small but performance-critical part of the program to a small but performance-enhancing accelerator creates a good match of application demand and hardware availability. Amdahl's law [Amd67] originally described a different computing model than the single CPU with accelerator, but it provides a quantitative measure of potential speedup that still applies. Equation 1 derives T_{acc}, the accelerated execution time, from T_{orig} is the original, unaccelerated execution time, F_{acc} is the fraction of execution time eligible for acceleration, and S_{acc} is the factor of speedup of the accelerated fraction.

$$T_{acc} = T_{orig} \times \left(\left(1 - F_{acc}\right) + \frac{F_{acc}}{S_{acc}} \right)$$

Equation 1. Potential speedup in accelerated applications [after

The ideal case (i.e., the smallest T_{acc}) occurs when F_{acc} approaches 1 and S_{acc} is large. High F_{acc} is in fact found in many compute-intensive BCB applications, and S_{acc} values of 100 to 1000 or more have been reported in FGPA acceleration of a number of applications.

1.3 Computations amenable to FPGA acceleration

Any computing platform works better for some applications than for others, partly because of the physical structure of the computing hardware. In order to understand which problems work best with FPGA acceleration, one must understand the general structure of the FPGA acceleration hardware and its relationship to the rest of the computing system.

1.3.1 FPGA and accelerator hardware

Figure 2 illustrates the most common kinds of system configuration. Typical configurations are based on PCs or multi-CPU supercomputers. The host system has one or more CPUs, one or more main memory units, and standard disk and network IO. The exact type of system interconnect varies: PCs typically have PCI busses, Silicon Graphics products use their proprietary NUMAlink, and other vendors use other interconnection hardware. One or more FPGAs are built into a board or board set, along with some amount of on-board memory. Memory configurations differ significantly between specific accelerator products, but often feature several separately addressable memory banks per FPGA totaling a few MB to a few hundred MB – far less than host memory. If multiple FPGAs are present in one accelerator configuration, connections between them can total tens of Gb/s bandwidth and latencies well under a microsecond.

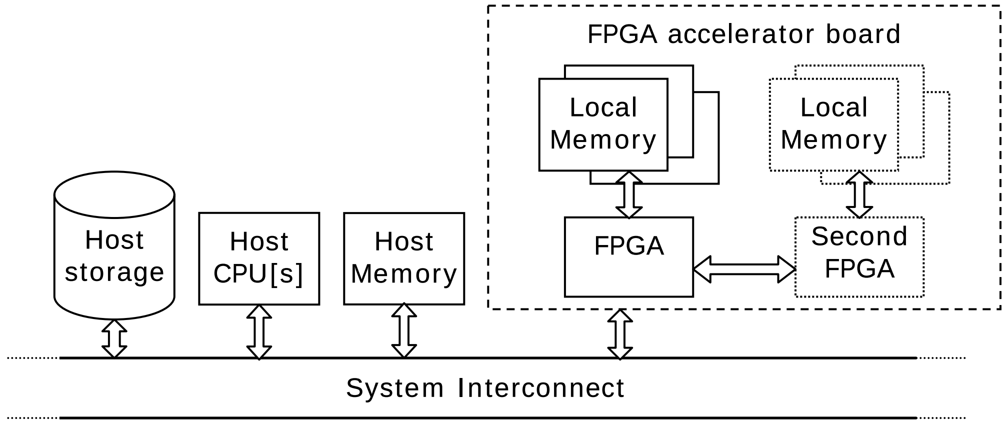


Figure 2. Host computer with FPGA accelerator

Some configurations allow the FPGA accelerator access to the host memory, but with significantly worse bandwidth and latency than access to the local memory. Host access to the accelerator board's local memory is limited by the system interconnect bandwidth, further slowed by on-board switching and arbitration. Some accelerator boards have additional IO features, but those are not used in the applications being addressed.

The FPGA has a few Mb of on-chip RAM, and typically a few MB to a few hundred MB RAM local to the FPGA accelerator. This creates an FPGA-centric memory hierarchy of on-chip RAM, on-board RAM, and host memory. Bandwidth and latency worsen by an order of magnitude or more at each step away from the FPGA, but capacity increases dramatically. Some host systems introduce additional levels of hierarchy, due to the costs of disk IO and potentially complex memory access networks.

1.3.2 Characterization of good FPGA candidates

The physical facts of the FPGA accelerator define the kinds of problems that it handles best. Problems with the following characteristics are generally good candidates for FPGA-based acceleration:

- **Massive, open-ended parallelism**. BCB applications are highly parallel, with the possibility of thousands of operations being executed concurrently. Many BCB applications also feature *open-ended* parallelism, in the sense that there is effectively no upper bound on the number of PEs that can be applied to the calculation. These applications map well onto devices with thousands of concurrent processing elements (PEs).

· **Dense, regular communication patterns**. Communication is generally regular and local: on any iteration, data only need to be passed to adjacent PEs. The FPGA's large number of communication paths ensures that all PEs can send and receive data every cycle, while the local communication ensures low latency.

· **Manageable data set sizes**. Working sets are often on the order of a few MBs. This fits comfortably into on-chip memory or memory local to the FGPA accelerator hardware, minimizing relatively slow access to host memory. For many applications, there is massive reuse with each element being used at least $O(N^2)$ times.

· **Deterministic data access**. When the working sets are too large to fit on-chip, they usually have predictable reference patterns. This allows the relatively high latency of off-chip transfers to be hidden by the high off-chip bandwidth (500 signal pins). In extreme cases, such as when processing large databases, data can be streamed through the FPGA at multi-Gb rates by using the dedicated I/O transceivers.

· **Data elements with small numbers of bits**. Reducing the precision of the function units to that required by the computation allows the FPGA to be configured into a larger number of function units.

· **Simple processing kernels**. Many computations are repetitive with relatively simple processing kernels being repeated large numbers of times. The fine-grained resource allocation within an FPGA allocates only as many logic resources as needed to each PE. Simpler kernels, requiring less logic each, allow more PEs to be built in a given FPGA – a tradeoff of computation complexity versus parallelism not available on fixed processors.

· **Associative computation**. FPGA hardware works well with common associative operators: broadcast, match, reduction, and leader election. In all of these cases, FPGAs can be configured to execute the associative operator using the long communication pathways on the chip. The result is that rather than being a bottleneck, these associative operators afford perhaps the greatest speed-up of all: processing at the speed of electrical transmissions.

Many important problems match that profile, not just in BCB. Many BCB problems, however, embody a number of the factors that work toward success in FPGA implementations.

Not all problems work well in FPGAs, however. Floating point calculations generally consume so many logic resources that there is little opportunity for on chip parallelism. In many cases, however, applications implemented in floating point on standard processor can be re-implemented in fixed point or other arithmetic, with little or no cost in accuracy.

1.4 Case studies

The initial phase of this work used standard FPGA development tools to create examples of application accelerators and families of accelerators. Experience with these applications creates the conceptual foundation underlying the idea of reusable accelerator families. Chapter 2 discusses each case study in more detail, including the lessons learned from each one. Two features recurred throughout these studies.

First, the performance-critical subsystems within each accelerator generally come from insights specific to the computation at hand, using reasoning that could not credibly be automated. This makes it clear that a skilled logic designer must participate in accelerator design, contradicting the idea that a programmer of typical skill with a "C to gates" compiler can achieve the performance levels reached in these studies.

Second, the reusable subsystems in each application are the patterns of communication, parallelism, synchronization, and memory access, not the leaf data types or calculations. This inverts the usual idea of hardware components: that they are indivisible "black boxes," and that component-based application development consists of creating connections between them and to the system memory. Instead, the application-specific functions and data types must be treated as parameters to the design of an application accelerator. In the language of software design patterns, it should be logic designer's job to *"implement the invariant parts of an application once, and leave it up to the [application specialists] to implement the behavior that can vary"* [Gam94].

Analysis of microarray data

DNA microarrays measure the amounts of many different RNA transcripts in a cell or tissue. Different transcripts can be associated with life processes in normal and diseased states, and hold promise as tools for medical diagnosis and for determining response to treatment [Kim01]. This accelerator parallelizes the linear regression using dedicated PEs, connected to custom memory and bus subsystems [Van03, Van04]. The memory subsystem and distribution network create high levels of data reuse, reducing the amount of host communication required. The data distribution network also provides a striking example of the complexity of the relationships between the sizes of computing arrays in different parts of a system.

Sequence alignment

Smith-Waterman, Needleman-Wunsch, and related algorithms are staples for approximate string matching in bioinformatics and other areas. This application is noteworthy for its many options in data types, scoring algorithms, and gap and end rules, affecting different structural levels of the implementation. It is equally noteworthy for the brittleness of the hardware implementations that have been made public [e.g. Yu03], their complete inability to accommodate any variation in any of the choices made affecting the algorithm's behavior. This case study [Van04a] presents a wide and extensible range of behavioral options. It also demonstrates the performance benefit of fine-tuning each application accelerator to its specific task, and shows some of the weaknesses of mainstream FPGA design tools in handling highly configuration logic designs.

Rigid molecule interactions

Since its introduction [Kat92], 3D correlation has become a staple in estimating the strength of interactions between two molecules. Transform techniques are commonly used to reduce the polynomial complexity of the problem. This case study [Van04b] demonstrates a significant speedup using direct summation on hundreds to thousands of PEs implemented in one FPGA. This allows generalized correlations able to model chemical phenomena that are infeasible or impossible using transform techniques. It also uses dedicated logic for three-axis rotation and data filtering to reduce dealys caused by host interactions.

Molecular Dynamics

The rigid molecule interaction application examines fixed points in time, using fixed positions for the effects of each atom in the system. Molecular dynamics (MD) allows each atom to move independently, subject to spring-like forces due to bonding, electrostatic effects, and van der Waals interactions. This case study examines the consequences of converting floating point MD applications to FPGA-compatible fixed point [Gu05, Gu06]. The results show that fixed point calculations give acceptable accuracy and numerical stability, and show how new analysis of the algorithm offers many different opportunities for optimization.

Sequence analysis

Sequence alignment is the best-known of the techniques used for analyzing biological sequences, but does not answer all kinds of questions. Palindromes define the structure and therefor function of many RNA molecules, and repeating sequences help identifying medically important mutations in pathogens and in human genetics. This study [Con04] examines both palindromes and repetitions. Similar computing structures show hundred- to thousand-fold speedups relative to PCs, given a streaming data source. This application again demonstrates the value of reusable, customizable components, and the value of associative operations in extracting meaning from large arrays of data.

Object recognition in 3D voxel data

Although correlation is a standard step in many 2D object recognition, it suffers "the curse of dimensionality" when extended to 3D applications. Data grows from $O(N^2)$ to $O(N^3)$, in for edge size N. Discrete rotations, however, go from $O(N)$ to $O(N^3)$. The problem as a whole jumps from $O(N^3)$ in two dimensions to $O(N^6)$ in three. This case study [Van05] re-applies the structures used in rigid molecule interaction to the 3D recognition problem. The computing structure handles multispectral data in a natural way, but also handles the spatially oriented data that occur in some medical imaging technologies, and reuses image rotation logic for correcting the anisotropic sampling grids that often occur.

Iterative optimization

This case study notes that many optimization techniques execute iteratively. They start with one candidate solutions, try some set of variations based on the best solutions to date, and use one of those results as the starting candidate for the next iteration. This highly general statement of the algorithm matches hill-climbing, simulated annealing, Gibbs sampling, and others. Variations include a randomized Metropolis criterion for acceptance of a new solution, as well as different strategies for generation of the next variations from the current candidate.

1.5 Basic contradictions in FPGA programming

Experience with these applications shows many weaknesses in current design tools' support for application acceleration. Those weaknesses arise, in part, form the tools' attempt to address three sets of contradictory demands imposed accelerator development. First, even closely related applications require significant customization in order to approach their performance potential, but an accelerator must be applicable to a wide range of problems for development to be cost-effective. Second, application specialists must have the ability to modify the data types and leaf computations to their unique and changing requirements, but even experienced software developers are generally unfamiliar with efficient structures for FPGA-based computing. Third is that application developers want to use the full capacity of their computing platform, but do not want to change their applications when additional computing resources become available.

1.5.1 Customization vs. generality

Customized application accelerators can fail in several ways: through over-generality, over-specificity, or (paradoxically) both at once. Consider a string-alignment application, applied to DNA sequences, to protein sequences, or to natural language text. DNA's four-letter alphabet can be encoded in two bits, but ASCII encoding represents English text in eight bits. The eight-bit data path can clearly handle the two-bit alphabet, but wastes 75% of its data path doing so. Those idle hardware resources could have been dedicated to active computing elements.

The eight-bit data path can also handle the 20-letter (five-bit) alphabet of protein sequences. Biological sequence comparisons, however, rarely use a binary match/no-match equality test. Instead, character comparisons report a graded goodness of match, according to the biological importance of a chemical difference. Text-based equality hardware can not adapt to this 'soft' comparison. A text-oriented application is both too specific to handle protein sequences, and also too general to give good performance in particular cases. Experiments have shown, for the string alignment problem, a 4:1 performance improvement for the dedicated DNA solution over a similar, over-general structure [Van04a, Van06], and 7:1 performance improvements for tailored accelerators for rigid molecule interactions [Van04b, Van06a].

1.5.2 Hardware design skills vs. end-user programming

Chapter 2 describes case studies of important families of applications in biochemistry and computational biology (BCB). Most of those use computation structures that can not be deduced from a straight forward representation of the algorithm in any high-level language. Hardware designers are able to employ design idioms such as 1000-stage pipelines, 1000-bit data words, low-latency broadcast networks, customized memory interleaving, and associative operations. Full exploitation of the FPGA's potential requires such techniques, not just a recreation of the standard sequential processor in FPGA logic. Software developers can not be expected to develop systolic, multi-bus structures, assuming they can be represented in a high level language at all.

Some few software developers do have experience writing highly parallel programs for clusters and supercomputers, but those skills are not readily transferable to the FPGA's kind of parallelism. Supercomputers, clusters, and grid computing commonly offer hundreds to thousands of processing elements, as FPGAs do. Parallelism in traditional computers favors large quanta of work per processor, tasks requiring seconds or longer, with communication latencies upwards of milliseconds. Parallelism within an FPGA differs radically. It favors small quanta of work, down to individual arithmetic operations, and allows massive on-chip communication with latencies in the nanosecond range.

The entry of software application developers into the FPGA design world requires rethinking of traditional truisms. For example, "*the electronic design automation industry has … given us a sustained 23% annual increase in designer productivity over decades. In a saner world, this would qualify electronic design automation for a major industry award. The problem, however, is that this 23% increase falls way short of the 60% or so increase in complexity each year coming from the underlying technology, such as Moore's Law*" [Pag04]. This statement includes the hidden assumption that designer skill holds roughly constant. Application developers without appreciable hardware skill can not be expected achieve the same productivity with those tools as logic designers, so that the combined productivity change due to the combination of tools and tool users falls well below the 23% cited. Given this new population of developers, hardware is pulling ahead of productivity faster than ever.

Even with the use of hardware design languages (HDLs) like Verilog and VHDL, current hardware design environments operate at a low semantic level. They require bit-level algorithm specifications, using state machines instead of familiar control structures. One might even draw an analogy to the earliest programmable computers with delay-line memories, where the exact timing of each instruction or data item had critical effect on performance. Hardware design still concerns itself with pipeline synchronization at a similar conceptual level, VHDL or no.

As noted above, application specialists must customize accelerators to get the computation behavior they require at performance levels approaching the FPGA's potential. BCB application specialists do not generally have hardware design skills to make the changes, and rarely have access to hardware designers. In fact, even if a hardware designer's service cost only \$100K per year, performance-per-dollar considerations would probably favor a 100-node cluster instead. The hardware designer's contribution must be used, and used differently, by many application specialists, to amortize the cost of development.

Both the logic designer and the application specialist must contribute to the accelerator design. Further, one logic designer's work must serve many application specialists. Deign tools should allow both to make their contributions independently, at different times, with no need for direct interaction between them.

1.5.3 Full utilization vs. changing loads and capacity

Performance in an FPGA-based accelerator generally comes from high parallelism, and FPGA parallelism is constrained by three things: the structure of the application family, logic resources used by data elements and operations specific to a member of the application family, and the amounts of computing resources available on a given FPGA.

Many algorithms scale in highly nonlinear ways. Hardware computation structures such as square convolution kernels and binary trees grow in predictable but irregular increments. Also, hardware implementations of many algorithms involve multiple computing structures with interdependent sizes. Often, the growth of different subsystems is constrained by availability of different FPGA resources. It is not, in general, adequate to assume linear growth of computing arrays with additional computing resources.

Even within an application family, different family members typically use data values and operations of different complexity. The fine grain of FPGA logic resources allows fine resolution in deciding whether to allocate more logic to each processing element (PE), or to reduce the amount of logic per PE in favor of more PEs. Within one accelerator, different subsystems typically use different types of data and operations, which further complicates sizing decisions.

FPGAs' computing capacity also grows according to Moore's Law. PC users have some to expect Moore's Law to deliver 2× performance improvements every 18 months or so, with no change to existing applications, or at most recompilation. Users of accelerated applications will surely expect similar improvements, also without source-level changes to their applications.

Performance of FPGA accelerators comes largely from their capacity for parallel processing. Increased parallelism depends on nonlinearities in problem scaling, on application-specific implementation details, and on the amounts of one or more different limiting resources in a given FPGA. Changes to application-specific details or use of a larger FPGA should automatically drive non-linear changes in an accelerator's degree of parallelism, but no known FPGA design tools can perform that tradeoff.

1.6 LAMP in brief

LAMP addresses the contradictions of section 1.5 with a prototype tool set. These tools have the following major features:

· **Two-level accelerator implementation**. LAMP recognizes the necessary and very different contributions of the logic designer and the application specialist in creating a customized application accelerator. Logic designers use VHDL (with LAMPML annotation), in order to exploit as much as possible of the FPGA's performance potential. Application specialists customize the accelerator using syntax similar to that used in main-stream OO programming languages like Java and C++.

· **Object oriented (OO) system design**. The LAMP design language, LAMPML, uses OO design techniques to create the interface between the accelerator's general model and its specific application. The logic designer defines the LAMP for an accelerator family using abstractions of the functions and datatypes. The application specialist tailors the accelerator by creating concretions for those abstract declarations. OO interface techniques allow flexible but clearly specified interfaces between the abstract model and the accelerator concretion, so the application-specific details can be filled in without changes to the HDL text. This also allows multiple levels of specialization of a widely applicable accelerator.

· **Data polymorphism**. The OO features of LAMP allow application data types to be changed and extended, while offering a high level of type safety. Since LAMP data and functions are cast into VHDL for synthesis, this requires a degree of type flexibility not present in pure VHDL. LAMP semantics extend VHDL, allowing a new degree of type-safe component reusability.

· **Automated accelerator sizing**. The LAMP tools provide primitives for estimating the amount of logic needed for concretions of data types and functions. These application specifics combine with the logic designer's understanding of the accelerator structure and the FPGA's specific resource pools, in order to create the best-performing accelerator for any specific member of the accelerator's application family.

· **Integration with existing HDLs**. Current HDLs give the logic design a familiar interface to the FPGA's logic resources, and have good support in optimizing compilers from many vendors. LAMP takes advantage of existing HDLs wherever possible, because of the advantages in vendor support and user familiarity. This lets the LAMP tools focus on novel semantics, rather than reinventing the whole world of logic synthesis.

Together, these features address the contradictions of section 1.5.3. LAMP offers the efficiency of a highly customized accelerator with the ability to target the accelerator to wide ranges of choices within a family of applications. It allows many different application specialists to customize their accelerators independently, without ongoing support by the logic designer. This separates the computing skills of the application specialist from the hardware skills of the logic designer, both in the logical structure of the accelerator and in the time at which each user participates in design of the customized accelerator. Automated sizing also lets the application specialist trade the degree of parallelism in the accelerator against the complexity of each processing element (PE) in it. This grants automated access to a unique capability of FPGAs: the ability to reduce logic usage some set of PEs, then recommit those logic resources to creation of additional PEs.

The prototype LAMP tools run as Java command line programs. The primary input is LAMPML, and XML-based notation. LAMPML defines the abstraction for the application model, implements the application-specific concretions, and integrates the set of definitions needed for any one application family or family member. A secondary format, CLAMP, serves many of the same purposes in a more intuitive syntax.

1.7 Summary of contributions

The major contributions of this research fall into three major categories: accelerator implementations, LAMP tool features, and reusable computing components. The implementations address BCB problems that had not previously been reported in the open literature, and offer levels of flexibility far beyond those reported in solutions to traditional problems. Novel LAMP tools address accelerator development problems that have not previously been addressed, including automated sizing of computation arrays to the resources available.

BCB application accelerators

At the time this research began, it was not at all clear that FPGA acceleration would be effective in more than a few isolated BCB applications. The first contribution of this research is proof by example that FPGA acceleration can and does yield speedups of 100× or more for BCB applications and application families that differ widely in the structure of their computaitons. Individual applications, especially various forms of string alignment, have been demonstrated in the past, including some that have not appeared in open literature. This work, however, shows that statistical applications, computational chemistry, and string algorithms (both traditional and novel) all respond well to FPGA acceleration. Based on these experiences, there is every reason to believe that many other applications would respond equally well, if they were to be studied.

Experience in creating these applications also raises the level of the targets of FPGA acceleration, from point solutions to customizable application frameworks. Reports in the open literature generally present brittle solutions to precisely specified computing problems. These computing problems, however, present themselves in broad families characterized by common synchronization, parallelism, and communication structure. The broad *application family* is the most attractive target of acceleration, not any one specific member of the application family. These examples demonstrate several different families, and many more families exist than this effort had resources to examine.

Even with their very different structures, the application families all benefited from the FPGA's massive parallelism. The accelerators are all built around computing arrays of several sorts, and larger arrays are preferable in all of the cases examined. Many computations consist of interlocked computing arrays, subject to multiple nonlinear constraints, dependent on application specifics, and limited by multiple FPGA resources. This combination of factors turns array sizing into a difficult choice.

These examples demonstrate serious weaknesses in existing logic design tools when applied to application accelerators. First, current HDLs present a basic model of computation unfamiliar to the application specialists who define the details of any computation. Second, current tools do not permit parameterization of the kinds needed to support families of applications, including handling of data types and functions as parameters. Third, current tools do not support the designer in choosing the maximal computing array for a given application on a given FPGA. Together, these failures of current tools distinguish accelerator design from traditional logic design, and suggest ways in which accelerator design tools should differ from existing tools.

Accelerator development tools

Lessons learned in the application case studies suggest features for accelerator design tools that logic design tools do not address. These lessons informed the design of the LAMP tools developed during this research. In particular, LAMP tools demonstrate support for:

- **Application families, not just point solutions**. LAMP closes a historical gulf between specialized, brittle FPGA implementations of BCB algorithms and the versatility of the algorithms themselves. The LAMP tools embody the idea that an algorithm's general structure is the reusable design element, and that different families of algorithms typically have very different structures.

- **Functions and data types as design parameters**. Existing HDLs have limited reconfiguration capability, based on selections from fixed lists or sizing in terms of integer values. LAMP introduces features of common OO languages to allow the data types and calculations in an application to change, while holding the communication structure fixed. This follows the "inverted" flow of control common in large software systems, but unlike traditional logic design.

- **Automated sizing of computation arrays**. The applications under study display open-ended parallelism, in the sense that more PEs can always be put to use in the computation. When applications display open-ended parallelism, the right size of computation array is "as big as possible." That indefinite number depends on the capacity of the given FPGA, the complexity of each computation, and the application's communication and synchronization structure. The possible degree of parallelism can change each time the application specialist decides to use more or fewer bits per datum, or decides on more or less complex leaf calculations. The LAMP tools do not require the degree of parallelism as input. Instead, it is a consequence of other design decisions.

- **Combination of design skills**. A successful FGPA-based accelerator requires contributions from at least two participants with different skills: a logic designer, able to exploit the full potential of custom logic circuitry, and the application specialist with the idiosyncratic and changing computation needs. LAMP allows these users to make their respective contributions using different input representations and at different times, independently of each other.

Some of these features have appeared in previous logic design tools, especially parameterization of functions and data types using object oriented language features. No other documented system addresses all of these design goals, however.

Reusable computing structures

Yet other results of this research arise from the specific case studies. One is the novel combinatorial memory network used in the microarray application (section 2.1.1), which can be put to use in any application based on small subsets of large data sets. Solutions to little-studied string analysis problems (section 2.1.5) address applications of scientific interest in themselves. Some structures within those solutions are highly reusable, including the networks for examining substrings of many sizes, with or without insertions and deletions. Success in these applications suggests that many other areas of string analysis are amenable to FPGA acceleration. Acceleration of rigid molecule interactions (section 2.1.3) also addresses a problem of scientific and commercial importance. It does not just solve forms of the problem from the open literature, but offers researchers new opportunities for modeling chemical phenomena not handled by standard techniques. It also demonstrates how techniques developed for one application area (computational chemistry) sometimes benefit problems emerging because of advances in other areas (3D data analysis, section 2.1.6). This application integrates components that had been developed for other application areas, but uses the LAMP tools' capabilities to generalize the functions in ways not possible for other design tools. In addition, demonstrates the potential value of mining existing designs for efficient computing structures, and extracting useful communication and synchronization structures that could not previously be reused.

These reusable components expand the traditional ideas of what can be reused and what is a component in a logic design. The familiar idea treats a component as a black box, with no visible internal structure; reuse means creating communication and synchronization around them. Instead, these components *are* communication and control structures reused by defining their inner content.

Also, this work demonstrates the practical value of object oriented design techniques and notations as an architectural tool for describing families of application accelerators. It also demonstrates informal use of parts of the UML standard as design and documentation aids for system design.

1.8 Notational conventions

Typographic conventions

A term is printed in *italics* where it is first used and defined. Literal programming language keywords or examples of machine-readable text appear in `mono-spaced` text. Unless some particular language is specified, such text should be read as illustrative pseudocode, not necessarily valid code in any real programming language. If **`bold`** and `plain` mono-space are distinguished, bolded text represents keywords defined by the language and plain text contains symbols or data chosen by the programmer. Mathematical and formal notations are specified at the point where they are used.

UML class diagrams

The UML (Unified Modeling Language [OMG03]) is used to describe many aspects of system design and behavior. Although there are many subsets to the UML graphical notation, its *class diagrams* are used most commonly throughout this discussion. Figure 3 summarizes the subset of class diagram notation used here. This discussion uses analogies to C++ or Java for explanatory purposes, not as a precise semantic definition.

The actual system behavior differs in some ways, and need not correspond exactly to any previous language or system.

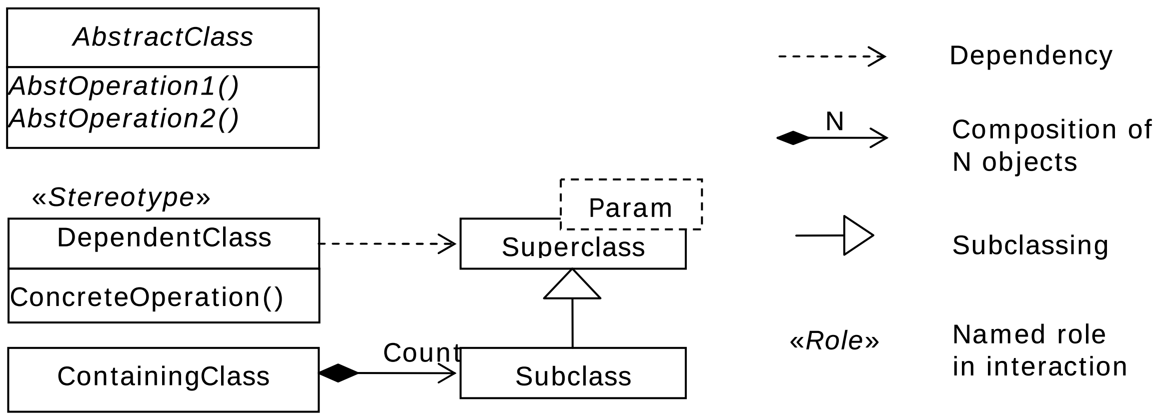


Figure 3. UML class diagram notation

A *class* is an identified set of *interface items*: operations or functions, constant values, type declarations, and state, all of which are optional. The interface item is an externally visible portion of the class definition, whether or not an implementation (also called concretion or realization) is given. A *concrete class* is one in which all interface items have implementations. An *abstract class*, identified by italicized captions, is one in which implementations for one or more interface items have not been provided. A *parameterized class* is defined in terms of one or more values or types. It represents a family of class types, abstract or concrete, where a family member is uniquely identified by the parameter bindings. Figure 2 shows class `Superclass` with one parameter `Param`. C++ syntax would write this as `Superclass<Param>`. Concrete classes can be instantiated, i.e. objects of that class type can be created. Abstract classes, because they are not completely specified, can not be instantiated. UML classes correspond to C++ or Java `class` constructs. UML abstract classes correspond to C++ virtual classes, to Java `abstract` classes, and to Java `interface` constructs. C++ and Java both allow default definitions for any, possibly all interface items in an abstract class. Meanings can be assigned to other languages, too – abstract classes arguably map to VHDL `component` definitions, and concretions to the bodies defined in `entity`/ `architecture` definitions.

A *subclass* exports all the same interface items as its superclass, possibly more. It provides implementations for some or all of the superclass' abstract interface items. It can also replace (or *over-ride*) implementations of any superclass interface item, as long as the superclass interface is preserved. If the subclass does not re-implement a concrete interface item of the superclass, the superclass implementation is used and the subclass is said to *inherit* the implementation. If the subclass does not implement an abstract interface item of the superclass, the subclass still exports the abstract interface item, but the subclass itself is abstract. No distinction is made between subclass inheritance and concretion of an abstract interface.

The subclass relation creates a partial ordering over the classes of a system, and is transitive. If class C is a subclass of B, and B a subclass of A, then C is a subclass of A. When the distinction matters, C is said to be an *immediate subclass* of B, but not of A; the immediate subclass relation is not transitive. Since a subclass exports all interface items that its superclass does, and since any class trivially exports its own interface, any class is normally considered a subclass of itself. The superclass relationship has a similar definition. It is also a transitive relation, and a class is normally considered to be a member of its own set of superclasses.

The UML standard makes clear distinction between *aggregation* and *composition* as two ways of collecting several object references within some other object. Composition is a relationship in which the contained objects (at the head of the arrow) exist only with the containing object (at the tail of the arrow), and exist only as long as the container's lifetime. For example, a bicycle is usually considered a composition of its parts – when the bicycle is discarded, all of its parts are automatically discarded. In aggregation, the parts have an existence independent of the collection. A project team is an aggregation of its human members, for example, but those people go on with their lives even if the team dissolves. Because of the static nature of hardware designs, dynamic association and dissociation of objects is not possible. As a result, this discussion refers only to composition, unless otherwise specified. Composition allows any number, zero or more, contained objects within a containing object. Constraints on the cardinality (e.g. "exactly one" or "one or more") are indicated using text labels on the arrowhead indicating composition.

A *stereotype* label, enclosed in «guillemots», is used as identification. It most commonly refers to a role that one class plays with respect to others, where that role is given a narrative description in the related text.

A *dependency* relation is a general statement that one class (at the tail of the arrow) makes use of another (at the head of the arrow), without specifying what kind of use is made. This could be a subclass relation, composition, or any other relation such that the dependent class' behavior can change when the class on which it depends is changed.

Navigability is the knowledge in one object or type that it has some relationship to another object or type. This property is not, in general, symmetric: object *A*'s ability to access *B* does not imply *B*'s ability to access *A*. Navigability is shown by the direction of the arrow, including the hollow-headed arrow used for subclassing.

These definitions are intuitive rather than formal, and are described using familiar programming languages as examples. Those examples are only illustrative, not exclusive and not necessarily exact. Different languages have different subclassing rules, for example, so the exact meaning of a subclass relation is different in the C++ [Str97], Java [Gos05], C# [ECMA05], Smalltalk [Liu00], JavaScript [ECMA99], Eiffel [Mey92, ECMA05a], and Beta [Mad93] languages. Object instantiation must also be different in HDLs than in software-oriented languages. Both software and hardware instances must contain instance data. Hardware instances normally replicate the behavioral logic, as well, where software systems share one code instance among arbitrary numbers of data instances. Since logic instantiation lies outside the UML standard, class diagrams should not be understood to represent logic instantiation.

UML class diagrams are used to help improve the understandability of a system design. The full system of UML notation is rich and complex. No effort is made to apply every notational feature to every diagram, since excessive detail and visual clutter tend to work against the goal of clarity. Other notations than UML are used, as well, so diagrams based on UML class diagram notation are labeled as such.

2 CASE STUDIES IN FGPA APPLICATION ACCELERATORS

The first phase of this research implemented a series of FPGA-base accelerators for BCB applications. These case studies gave crucial insight into the features that allow an accelerator to address a whole range of related applications, and gave direct experience with the problems that current logic design tools create for these applications.

Section 1.3.2 identified a set of characteristics that predispose an application to successful FPGA implementation. Many computations, including those in bioinformatics [Ste01], embody those characteristics:

· **Massive, open-ended parallelism**. Processing of long strings parallelizes at the character level, and grid-based molecule interactions parallelize at the level of grid cells. Many BCB applications share this level of parallelism, despite their different basic units of computation.

· **Dense, regular communication patterns**. String processing, alignment by dynamic programming, 3D correlation, and other applications all meet this description. This allows well understood hardware techniques to be employed, including systolic arrays and pipelines with hundreds or thousands of steps. The communication pattern is often the reusable system component, connecting replaceable leaf computations with replaceable data elements.

· **Modest working sets and deterministic data access**. Although BCB data sets can be large, they generally are often amenable to partitioning and to heavy reuse of data within partitions.

· **Data elements with small numbers of bits**. Many BCB applications naturally use small data values, such as characters in the 4-letter nucleotide alphabet, or bits and fixed-point values for grid models of molecules. Although standard implementations generally use floating point, analysis often shows that simpler values work equally well.

· **Simple processing kernels**. Although BCB calculations often benefit from large computing arrays, processing elements within the arrays are typically repetitive and relatively simple. This allows small definitions for large computing arrays, since a small repeating element and a relatively simple rule for repetition can define arrays of any size.

· **Associative computation**. High connectivity and massive on-chip communication rates allow complex calculations to be performed at electronic speeds. High fanout enables one data access to fulfill many data references, one kind of data reuse. High fanin works lets algorithm developers use priority encoders, wide OR-sums, and other large computations.

These are the features that open an application to acceleration using specialized hardware. Empirical results with common algorithms [Sal98] show great opportunities for parallelism. This means that common algorithms can be accelerated, but they are also likely to accelerate incrementally with incremental acceleration hardware.

This section starts with brief descriptions of a number BCB computations that the current work has examined at various levels of detail. Discussion covers basic characteristics of the data involved and the general structure of the compute-intensive part of each application. It will be shown that a number of different applications share important characteristics, and that these characteristics can translate directly into hardware control structures.

2.1 Summaries of case studies

2.1.1 Analysis of Microarray Data

Kim, et al. [Kim01] proposed the following problem: find the set of three genes whose expression can be used to determine whether cancer tissue samples are metastatic or non-metastatic, by performing linear regressions of diagnosis against all size-3 subsets of gene expression levels. Instead of Kim's data, this case study uses publicly available sample data from Charles Perou's research [Per00], consisting of microarray data for roughly 100 samples, each of 10^4 genes. Each sample represents measurements from one biopsy. The set of biopsies included a few healthy controls, a number of diagnosed tumors, and some repeated samples from individual patients, before and after chemotherapy. The data do not contain any information that identify the individuals from whom the biopsies were taken. This study [Van03, Van04] implements Kim's technique, but uses the cancerous vs. healthy state as the independent variable.

In the Perou data set, that meant 10^{11} to 10^{12} sets of regressions needed to be processed, to cover all size-3 gene subsets. Although simple to implement, they reported that this computation was intractable even on their small cluster of PCs. Tests of a serial implementation concurred: for roughly 100 samples, 10,000 genes, and 13μs per iteration, the computation would have taken nearly three weeks on a 1.7GHz Pentium 4. This FPGA implementation resulted in a speed-up of a factor of more than 1500×, resulting in an estimated compute time less than 20 minutes. The computation of each set requires three steps: dot products and sums, covariance and inversion, and regression and correlation. To take advantage of the parallelism available in the FPGA fabric, processing

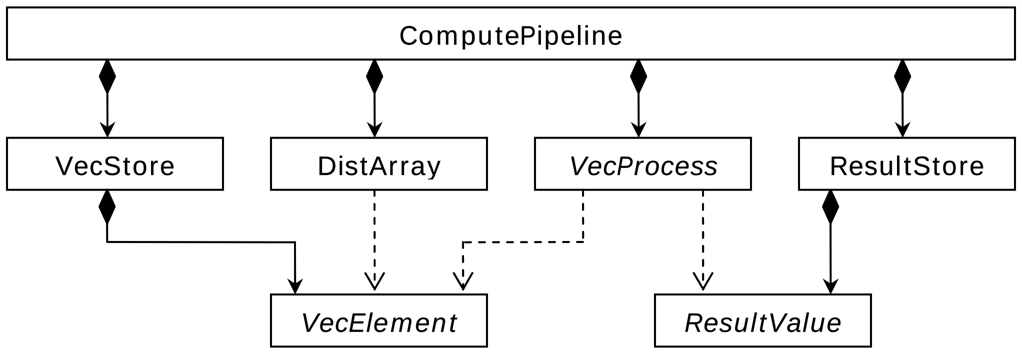


Figure 4. Class diagram for microarray data processing pipeline

elements were maximally replicated. Data inputs to the computation array come from a vector store and data distribution unit based on a combinatorial design theory [Har94], optimized for reuse of the vector data. Other notable features of the implementation include the handling of missing data, speed-matching the components, and precision management. Besides the speed-up and the techniques invented as part of this solution, the significance of this case study also lies in the ease with which it can be extended to many other microarray data analysis applications. Figure 4 illustrates the general structure of the calculation.

The computation pipeline as a whole has a fixed structure, composed of a vector store, a data distribution array, vector processing unit, and a result store. The two main application data types are the vector element and the result value, although values representing intermediate results arise internally to the vector processing unit. Numeric parameters, not shown in Figure 4, control the sizes of RAMs within the vector store, the size of the distribution array, the number instances of vector processing units, and the size of the result store. The vector elements and result values are shown as abstract definitions, so exact definitions need to be supplied for any specific application. The vector processing unit is also shown in abstract form, indicating that the general structure for supplying vector data and collecting results can be reused for different calculations. Two calculations have been prototyped, the linear regression of Kim's original request and a linear classifier [Joh02].

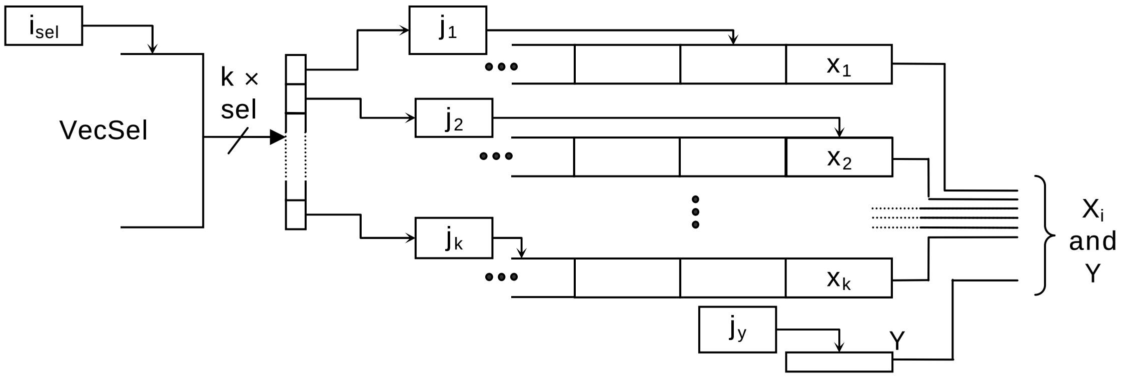


Figure 5. Vector store for combinatorial distribution network

The vector store and distribution network are illustrated in Figure 5 and Figure 6, respectively. The distribution network of uses a relatively small number of data memory accesses to supply size-3 sets to a large number of processing elements. Given *m* values from the *X* data vectors, this distributes $C_3^k = {}^{(k!)}/_{3!(k-3!)}$ sets of size three to the computation array, for the cost of reading *k* values from memory. *Y*, the dependent variable, is used by all PEs for all regressions. When $k = 9$ that means C_3^9 or 84 PEs can be supplied with *X* data values, 252 data elements total, for 28× reuse of each *X* data

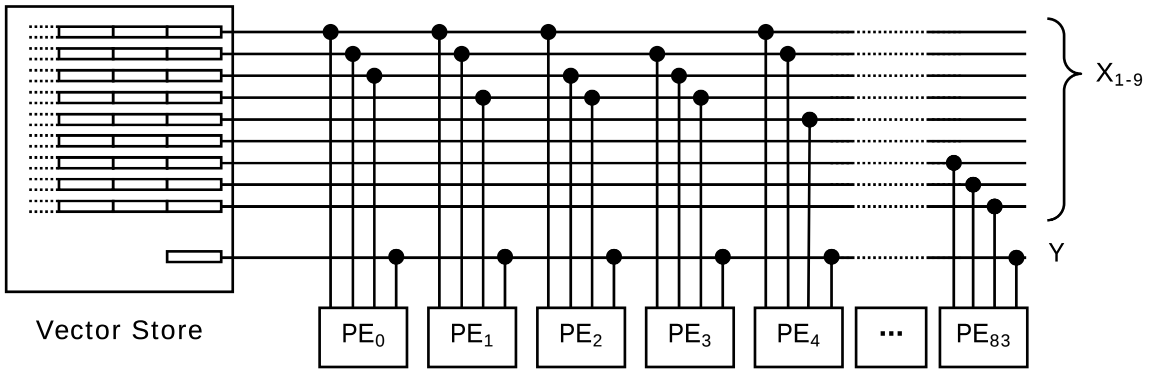


Figure 6. Combinatorial data distribution bus

value. When $k = 10$ elements of X data are available, $C_3^{10}=120$ sets of 3 or 360 data elements total are presented to the computation array, for 36× reuse of every X value read from memory.

Figure 5 shows the data store that supplies the distribution network. This two-level structure consists of a vector selection memory, labeled *VecSel*, and storage for the X and Y vector data. There is only one instance of the Y vector, the dependent variable (diagnosis), and that is reused in every correlation. Figure 5 also shows k memories of X data, the microarray samples or independent variables. Vectors in the X memories are reused many times so as to create different subset of X vectors for the distribution network, where each set of X vectors is defined by one word of the *VecSel* memory. It is the host's job to select the set of X vectors to be loaded into each X memory, and to create the set of index values to be stored in the *VecSel* memory.

The i_{sel} index fetches each set of selection indices from the *VecSel* memory sequentially. Each selection index in the set points to the start of a vector within one of the X vector memories. That initializes one j counter value for each for the X vector memories to the starting position of the vector chosen from that memory. The Y vector is reused across all sets of X vectors. A single set of calculations consists of sequential access to each element of the chosen vectors, so that corresponding elements from all vectors are presented in any one memory access cycle. Once the j counters have walked the length of their respective vectors, the i_{sel} value is incremented, a new set of indices is read from the *VecSel* memory, and the next set of vectors is processed. Careful pipelining allows back to back processing of successive sets of vectors.

In this example, the FPGA fabric could have held a few more processing elements than the number that could be supplied by the combinatorial network. The combinatorial network is crucial to this application's performance, however. It reuses *X* data elements at two levels, within the distribution network, and in multiple combinations of other *X* vectors chosen by the *VecSel* memory. Both kinds of reuse take advantage of fast, on-chip communication resources within the FPGA. Without that, the system as a whole would have been limited by the host's ability to send *X* values across the system bus. Although the combinatorial network does not support the highest computation parallelism possible for the FPGA fabric, it offers such an advantage in communication speed that it more than makes up for the slight loss of potential parallelism. RAM resources within the FPGA are not adequate to hold the entire data set at the same time, so *X* vector storages and *VecSel* index tables need to be reloaded repeatedly to complete the calculation. The vector store of Figure 5 can hold produce a large number of combinations from a modest amount of X vector RAM, so the entire structure can be double-buffered. Part or all of a vector store can be reloaded while the other is in use, minimizing idle time for loading of data.

This case study provided valuable insight in several areas. The most important result is that effective use of FPGA resources often requires a thorough rethinking of the problem.

The stated problem was to process all possible subsets of size three, taken from a set of data. Code Sample 1, below, shows a straightforward implementation of that statement. There is no logical path from the serial implementation of Code Sample 1 to the memory structure of Figure 6 and Figure 5. Cases like this reinforce the belief that effective use of FPGA resources demands a hardware designer’s insight – no credible amount of automated optimization could achieve the same level of performance.

```
for i = 0 to nDataItems-1
    for j = 0 to i-1
        for k = 0 to j-1
            process(data[i], data[j], data[k]);
```

Code Sample 1. Processing all data subsets of size 3, basic serial

Complex relationships between different parts of the computation pipeline also appear in this case study. Problem decomposition yields a number of interlocked subsystems, each with its own resource constraints and “magic numbers” for array sizes. The limits imposed by the combinatorial network made it clear that optimal system performance did not necessarily make maximal use of the FPGA’s computing resources. Overall system performance comes from combinations of resource usage terms, not from maximization of any one term.

Data used for testing this implementation were real results of microarray experiments, in which over 3% of all data items were recorded as missing or unusable values. Every tissue sample contained missing data elements for at least one gene, and over 45% of all genes had missing values for at least one sample. Missing data could not be ignored, and imputation (estimation) of missing values was problematic. Instead, this implementation took missing values into account, including genes where absence of control samples made analysis meaningless. This required a rephrasing of many arithmetic operations in terms of missing values and in terms of "quality control" failures in the data and computations.

This case study also suggested a revision of a traditional belief in hardware design, that reusable components are necessarily leaf components and that communication between them is not reusable. This example's vector store and distribution network form a reusable component, most readily described as one that can contain a variety of different vector analysis components.

2.1.2 Sequence alignment

Sequence alignment using dynamic programming based methods [Van04a], including Smith-Waterman and similar methods for approximate string matching, are perhaps the most accelerated applications in BCB. Aside from expensive commercial solutions (e.g. [Tim05]), however, most of these implementations have been extremely brittle, addressing only one or a few of a large number of possible options [Yu03], and with debatable biological significance.

There is an obvious gap between clever accelerator design and biologically significant control over what is being accelerated. The combinatorics for the problem explain part of the gap: there are just too many useful variations, and biologists keep coming up with new ones. If a fully generalized accelerator could possibly be designed, it would suffer inherent inefficiency due to feature bloat. The second reason for the gap between design and biological significance is that the users, biologists, are often unable to express their requirements in mathematical terms explicit enough for hardware implementation [Bia04].

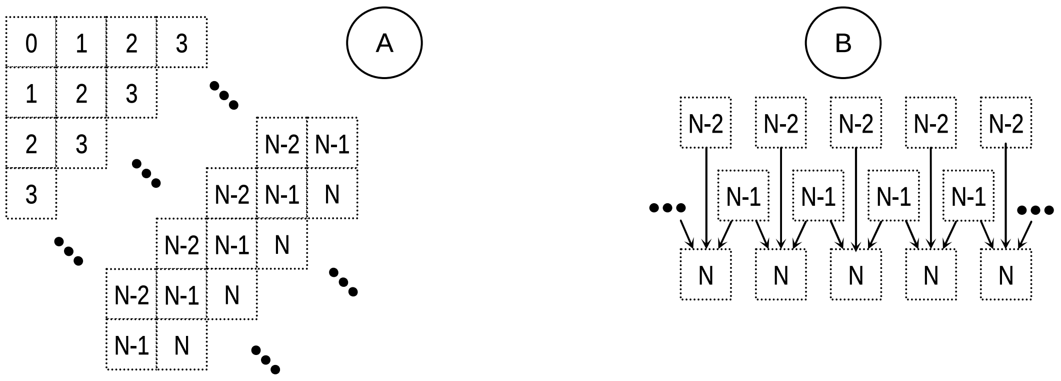


Figure 7. Approximate matching array.

(A) 2D structure of the computation, showing order of grid cell evaluation.

(B) Linear computation structure for cells evaluatable at one time step.

This case study implements a broad range of behavioral options (e.g. complex scoring functions) without losing efficiency due to feature bloat. This family of implementations computes from 2 to 10 billion cell updates per seconds, according to the options chosen.

C code running on a 3GHz Xeon processor runs at 29 to 46 million cell updates per second, 77 to 210× slower. Figure 7A shows the standard form of the computing array for dynamic programming of the type found in Smith-Waterman, Needleman-Wunsch and other sequence processing algorithms, labeled with the time steps at which each computation cell's dependencies are satisfied. Figure 7B extracts the one antidiagonal row from Figure 7A, showing a linear structure for the computation array. With proper assignment of computation data types to each of the cells, this can easily be implemented in FPGA logic.

This family of dynamic programming algorithms for approximate matching has variations in several dimensions. First, there are a number of common choices for character data types, including amino acids (normally a 20-letter alphabet), nucleotides (four), IUPAC ambiguity codes (16), or codons (64). Mixed comparisons are possible, including IUPAC wildcards vs. nucleotides or codons vs. amino acids. Second, there are at least sixteen overlap rules in the Needleman-Wunsch algorithm, which set the policy for scoring when one string is allowed to be a substring of the other [Dur98]. Third, gap penalties are usually scored with affine functions dependent on gap length, but arbitrarily complex functions have been suggested [Wat76]. Finally, there are many different policies for goodness of match scores when comparing dissimilar characters. At least eight different models underlie scoring for DNA string comparison, phrased in terms of two, four, or more free parameters [Nei00]. At least 83 different amino acid (protein) substitution matrices have been reported [Kaw00], reflecting different beliefs about evolution [Dim02], protein function [Ng00], and background probability models [Yu05]. Many of these substitution matrices are parameterized, for example the PAM N matrices which are defined as matrix exponentiation (PAM 1)^N, creating effectively infinite families of matrices.

Figure 8 summarizes the computing structure used for approximate string matching. The `Sequencer` object selects between two kinds of matching tasks: finding just the score representing goodness of match between the two strings in one pass, or a second

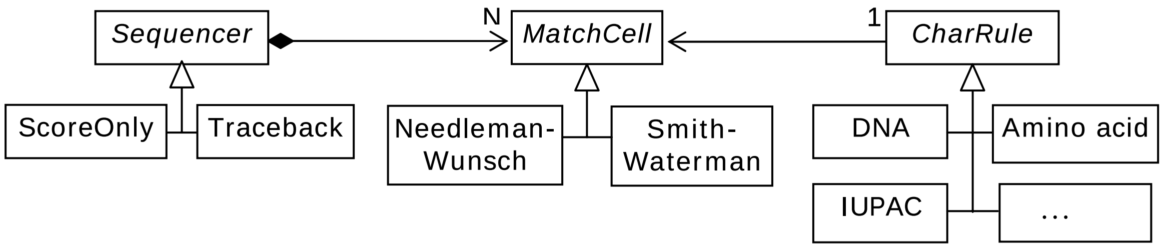


Figure 8. Class Diagram: Approximate String Matching

traceback pass through the computation yielding to find the character by character alignment. The `MatchCell` component implements the logic for one of the two recurrence relations, Needleman-Wunsch global alignment or Smith-Waterman local alignment. These differ in two ways: first, global alignment has 16 variations for special handling of mismatches or overhangs at the ends of strings but local alignment does not. Second, global alignment maintains only enough information in each computation cell for scoring from the string start to the current position in the array. Local alignment also holds information about the best local score found anywhere so far. The character rule (labeled `CharRule`) defines the data types of the characters in each string, not necessarily the same, and the substitution matrix that performs individual character comparisons. (Despite its name, the “matrix” is often a procedurally defined function.) As noted earlier, substitution matrices are usually parameterized. At a minimum, it’s often desirable to scale the range of values, allowing the tradeoff of higher scoring precision at a cost of more logic per comparison and fewer comparison units. Other multiple parameters have application-specific meanings, such as evolutionary closeness, background probabilities of nucleotides, or likelihoods of different kinds of mutation events. In general, each substitution matrix has some set of control parameters, the number and meaning of which are unique to that matrix.

This case study emphasizes a number of points regarding accelerator design. First, it shows the value well designed interfaces and interchangeable implementations of them. In Figure 6, the top level structure of the accelerator is captured in the `Sequencer`, `MatchCell`, and `CharRule` component definitions, all of which are abstract. Any combination of choices of concrete definitions for components is acceptable, as long as only one choice is made for all instances of a given component type. Much of the system’s flexibility comes from the multiplicative effects of the independent choices. This level of configurability exceeds VHDL’s capability, because of significant changes in parameter date types. As a result, component selection is done using file-swaps, outside of the VHDL language definition. The second important point demonstrated by this study is the importance of configurability, where different components are expected to have different numbers of tunable values, with different meanings, and of different types. Again, this level of flexibility was difficult for VHDL, and was solved largely by replacing the `generic` parameter list with one `generic` string, able to encode any desired set of options and parameters, parsed *ad hoc* by the various configurable components. Almost as much as component selection, this bypassed most of VHDL's configurability features. The third major point brought out by this case study is the performance advantage of application-specific configuration. If one general, programmable accelerator had been used, it would have required a character data path wide enough for the largest data type expected, six-bit codons. It would also have required the most general, RAM-based structure for substitution matrices, a significant amount of hardware per computation cell, and the global maximum score information required by Smith-Waterman but not Needleman-Wunsch. This general engine could certainly have handled two-bit nucleotide values, by letting $^2/_3$ of the character data path stand idle. It could also implement exact scoring based on exact equality or inequality of nucleotides, even though simple combinational logic would have required fewer gates and shorter signal delay. According to performance measurements, the DNA-specific implementation with gated equality tests would run at least 4× faster. About half of that gain comes from the larger number of computation cells possible when each cell uses fewer resources, and half from faster logic within each cell.

2.1.3 Rigid Molecule Interactions

This application [Van04b] deals with interactions between a protein of biological interest and a second, small molecule. The goal is to determine whether the small molecule is expected to have a strong interaction with the protein. If so, that compound (or a close relative) could have pharmacological activity. Although real molecules often have bonds that flex or rotate, it is prohibitively expensive to simulate such motion when screening 10^4 to 10^5 molecules for potential activity. Instead, it is common to select one or more configurations of the molecule and protein, and perform a six-dimensional search for the most favorable three-axis offset and three-axis rotation (or *pose*) between the two fixed configurations. One common approach performs the correlation in three translational dimensions at many (typically 10^3 to 10^5) three-axis rotations [Kat92], and collects the best-scoring among the poses. These algorithms digitize the two molecules, *A* and *B*, onto 3D voxel grids, then perform a generalized 3D correlation to detect a match between the molecule surfaces:

$$S_{x,y,z} = \sum_{i,j,k} F(a_{x+i,y+j,z+k}, b_{i,j,k})$$

Equation 2. Generalized correlation

In order to allow for errors due to approximation and digitization, multiple high-scoring poses are chosen for more detailed analysis, not just the one best-scoring pose. In standard correlation, *a* and *b* values are scalars (possibly complex numbers), and *F* is multiplication. The generalized form, however, uses the function *F* to represent the chemical behavior assumed by the specific application. The *a* and *b* voxel values are complex numbers, tuples, vectors, or any other value needed to represent the chemical phenomena of interest. In some applications the substrate and ligand molecules have different representations [Che02], suggesting that different data types be used for the two molecules' voxel values. Data types of interest have included

· Rewards for surface interaction and penalty for interior collisions [Kat93],

· Hydrophobicity [Vak94],

· Surface normal vector opposition, solid angle complementarity, and other shape descriptors [Gol00], and

· Electrostatics, counts of neighboring atoms, and atomic contact potential models of desolvation energy [Che03a, Che03b].

This list just indicates some of the phenomena and combinations that have been used in the scoring function *F*, with no attempt at completeness. It is widely accepted that pharmaceutical companies have proprietary and highly confidential scoring functions. For a scoring technique to have commercial value, such customers must be able to modify an accelerator in order to implement their proprietary scoring functions.

When function *F* in Equation 2 is a product or linear combination of products, it is common to compute the result using Fourier transforms. While asymptotically favorable, this introduces far more precision than is usually needed and also makes it impossible to apply non-linear force models. Instead, this implementation uses an FPGA algorithm based on the direct summation. It uses a well-known systolic design extended to three dimensions [Swa87]. Using the hardware pipeline suggested by Figure 9 and a simple force model, a prototype implementation achieved a speedup over 200× relative to a PC implementation based on a standard FFT implementation, for problems of similar size.

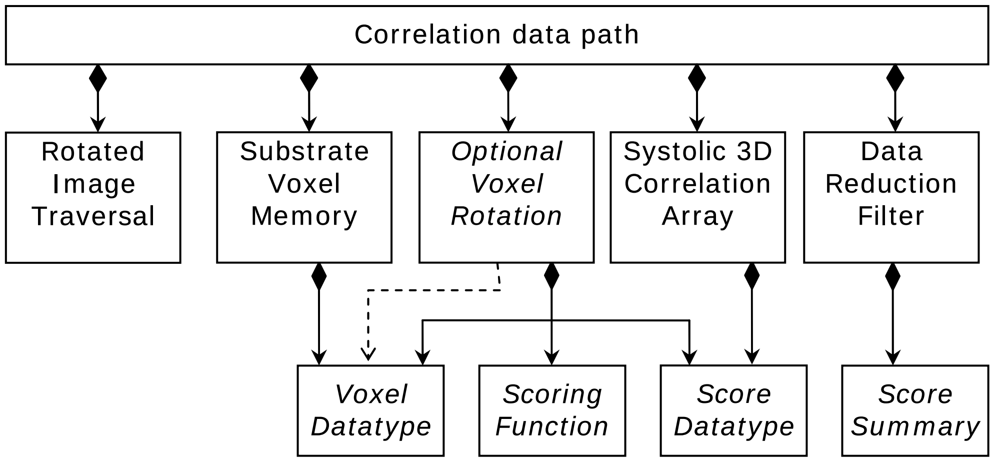


Figure 9. Class diagram of docking application

Figure 9 illustrates the basic structure of the accelerator's computation pipeline, reading the major components of the correlation data path in left to right order. These components are:

· **Rotated image traversal.** This eliminates three-axis rotation as a separate step, by performing optimized linear transforms on traversal indices, so the molecule image is indexed and padded in rotated order. There is no additional memory buffer for the rotated image, and only a modest cost in addressing logic and padding. Logic delays can be pipelined, so the rotated traversal has no effect on system throughput.

· **Substrate voxel memory**. The larger molecule voxels are stored in the FPGA’s on-chip memory. This is a RAM of conventional structure, with indexing logic for the three-dimensional grid containing the molecule image. This stores the voxel values for the larger of the molecules, typically 50×50×50 to 100×100×100, depending on the number of bits per voxel and the capacity of the FPGA’s on-chip RAM.

· **Optional voxel rotation**. Some models of chemical interaction involve spatially oriented vector values. Examples include normal vectors for checking that molecule surfaces are roughly parallel, or values representing the directional specificity of hydrogen bond donors and receptors. Although architecturally present in every instance of this pipeline, it’s an identity transform (pass-through) when voxel values do not contain spatially oriented components.

· **Systolic 3D correlation array**. This consists of a three-dimensional array or processing elements (PEs). The array serves two purposes: each processing element stores one voxel value for the second molecule, and it performs the scoring and summation arithmetic. Because FPGA logic can hold fewer voxels than the block RAMs can, this array stores the smaller of the two molecule grids. This also includes RAM-based FIFOs for holding partially summed score values.

- **Data reduction filter.** The correlation result could consists of 10^6 or more individual scores, but only the highest-scoring positions in the correlation result are of interest. Instead of transferring the whole set to the host processor, this block summarizes the best scores in the correlation. Because of approximations in the computation, and because of the possibility of multiple binding sites, this collects summaries of multiple different high-scoring poses. It is well known that large values often cluster around local maxima [Che03b]. To avoid the problem of repeated reporting of broad maxima, the filter collects one scoring maximum from each of many regions of the correlation result. That allows multiple local maxima to be recognized, even when large numbers of point-scores in the correlation represent one broad maximum. Figure 10 illustrates this scheme for reporting multiple maxima.

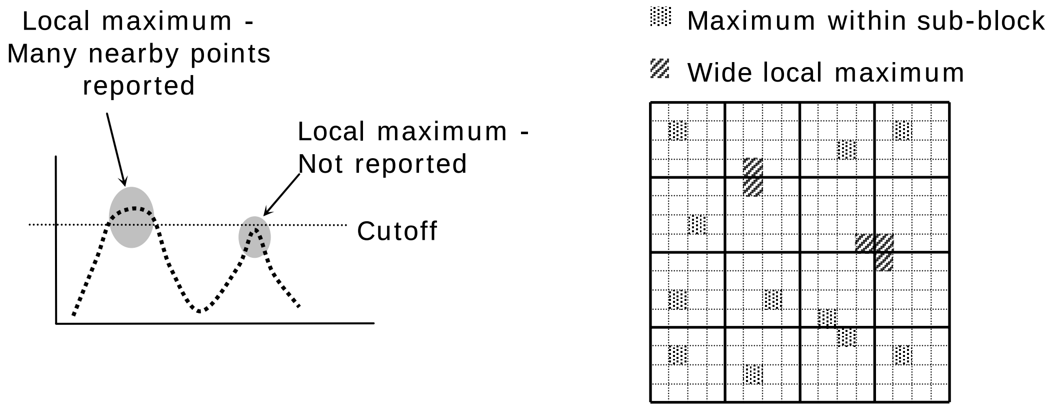

Local maximum - Many nearby points reported
Local maximum - Not reported
Cutoff
Maximum within sub-block
Wide local maximum

Figure 10. Data reduction filter: Region-based score selection reduces multiple reports of broad maxima.

The initial implementation of this structure used a fixed "score summary" function for reducing the volume of data output by the correlation array: a strict maximum of all scores found in some 3D subregion. As Figure 11 suggests, however, energetically favorable but entropically inaccessible solutions are not often useful. There is some belief that, rather than a deep, narrow minimum, a broad "funnel" in the energy surface is preferable, since it allows more approach paths for the partially docked system. There is no general agreement on how to quantify the funnel-like character of a minimum. It does seem likely that any such measure would combine scores for regions of the molecule interactions, and therefore fit the summary scheme in this computing structure, at least approximately. Two such summary functions have been implemented, in addition to the max described above. In one, each of the sub-regions of maintains a count of poses that exceed some threshold score. In the second experimental summary function, each subregion does not just count the number of poses exceeding some cutoff, but rewards such poses according to the amount by which they exceed the cutoff. The chemical validity of summaries based on alternative summary functions has not been compared to the max-score summary. The architectural fact has been demonstrated, however:

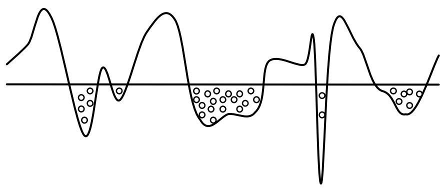
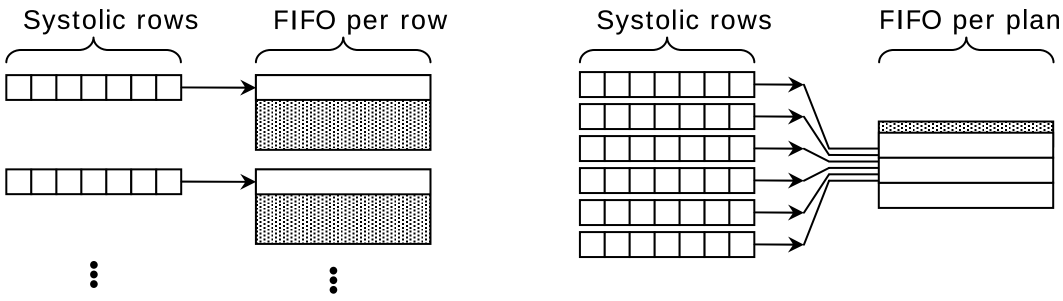


Figure 11 . Sketch of free-energy landscape of a molecule interaction, after [Koz05]

alternative summary functions have been implemented, and application specialists are welcome to create new ones.

The RAM-based FIFOs inside the correlation array deserves special attention. Figure 12 shows how each plane within the cubical array consists of some number of rows. Each row is a systolic array with a FIFO for matching intermediate sums to the input data schedule. FIFO words, in this example, are 10 bits wide. The entire array runs on the same clock, so all of the fixed-length FIFOs accept input and generate output synchronously. Initially row were phrased as the set of computation cells plus FIFO, as in Figure 12A. Each was allocated a separate RAM for implementing its FIFO. Since RAMs come in indivisible units 36 bits wide, each FIFO wasted 72% of the width each RAM.



A) Each FIFO handles one row. Allocation overhead lost in every row.

B) Wide FIFO handles whole plane. One allocation overhead per plane.

Figure 12. Word-merging across computation plane increases efficiency of RAM width usage.

▨ Wasted width

An improved design (Figure 12B) made the FIFO into a component of the plane rather than the row of computation cells. Since the FIFOs run synchronously, results from all rows in a plane are concatenated into one word, 140 bits wide in one implementation. That 140 bit word, when read, is broken up into its original 14 words of 10 bits. Four RAM blocks can be ganged into one FIFO to handle that 140-bit word. The ganged width is 36×4=144 bits, of which four bits are wasted – 97% efficient use of RAM width, up from 28%.

One implementation detail inside the data reduction filter came from consideration of the chemistry problem being addressed. It would have been straightforward to subdivide the correlation result along the axes of the correlation result's grid. As different iterations processed different three-axis rotations of the substrate, however, different subsets of the substrate molecule would correspond to a given subdivision within the collecting grid. This would make it difficult to compare results from different rotations, since the summary values from corresponding elements of the data reduction filter's memory would not represent the same substrate voxels. Instead, the rotated index from the rotated image traversal logic is used. That leads to an irregular reference pattern in the data reduction filter's memory, which is handled within the filter pipeline using the FPGA RAM's dual-porting feature. Together, these allowed the filter to run at the full speed of the computation array. The result was that a given subdivision within the data reduction memory always corresponded to the same set of voxels in the substrate memory, making it much more meaningful to combine results of multiple rotations.

This case study reinforces the lessons learned previously. First, there is a wide range of force laws in the open literature, and a common belief that pharmaceutical companies use their own proprietary and closely guarded force laws. As a result, efficient use of FPGA resources requires support for user-defined scoring functions, and requires data paths that can trade off computational complexity against degree of parallelism. Second, good performance demands a broad understanding and analysis of the computation being performed. The users' initial request was that their 3D FFTs be made to run faster. If that had been done, only modest speedups would have been possible. By implementing the correlation sum directly, this solution uses well-studied signal processing structures that make effective use of the FPGA's capabilities. In addition to accelerating the generalized correlation, this also enables use of nonlinear scoring functions, which are impossible to the transform-based implementation. Third, this application demonstrates that implementation idioms natural to hardware designers bear little resemblance to software-oriented forms of the problem statement.

It is not credible that automated translation could convert any normal representation of correlation into the computing structure used here: a systolic array supplied by a broadcast network, interleaved with RAM-based FIFOs. Neither is the wide-word FIFO a software idiom or simple deduction from the problem statement. Likewise, hardware dual-porting and elaborate pipeline control were needed to achieve single-cycle operation of the data reduction filter. That would have been, at best, difficult and un-natural to express in any standard programming language. It is doubtful that any automated analysis would derive any of these structures from typical HLL code describing the problem.

2.1.4 Molecular Dynamics

Molecular dynamics is the study of molecule behavior over time, whether one molecule (as in protein folding), two (as in substrate-ligand interactions), or many (as in water molecule or cell membrane interactions with large biomolecules). The simplest approaches use a family of spring-and-ball models to represent atoms bound covalently, requiring $O(N)$ computations at each time step, for the known covalent interactions. Other forces, including desolvation effects and hydrogen bonding, define the molecule's local or secondary structure, global or tertiary structure, and intermolecular interaction. Because, in principle, any atom in a molecule can twist around into interaction range of any other atom, all-to-all or $O(N^2)$ force interactions must be computed. This is generally the dominant phase of computation at each time step, and 10^4 to 10^7 time steps may be required for understanding some molecular behaviors.

This can be hard to accelerate using FPGAs because of the precision needed for accuracy in the simulation, and because of the limited support for floating point on FPGAs. At least one previous attempt actually resulted in a slowdown [Azi04] relative to a PC. This study [Gu05] has two aspects: implementing the most time-consuming computations (modeling the long range forces and motion updates) and a conducting a rudimentary examination of simulation accuracy as a function of precision. We found that with conservative assumptions we could obtain a speed-up of more than 40× while with relaxed, but viable, assumptions we could obtain a speed-up of over 80×. Work continues in extending this work to periodic boundary conditions.

This case study also required careful analysis of the application. Force computations are normally computed using single- or double-precision floating point vales. Analysis and testing showed that fixed-point values of reasonable size could meet the precision requirements. Parts of the analysis were complicated by the fact that the many-body computation is chaotic. Even small variations in initial conditions or computation technique can cause solutions to diverge. This is unavoidable, and it is realized that "*obtaining a high degree of accuracy in the [particle] trajectories is neither a realistic nor a practical goal*" [Rap95]. Instead, different solutions are considered as different samples drawn from the distribution of possibilities. Not the exact solution, but the general, statistical character of any new solution must be compared to that of accepted solutions, and quality control measurements (e.g. conservation of energy and momentum) must be made and verified.

2.1.5 Sequence analysis

Although sequence alignment, or approximate string matching is the mostly widely known of sequence analysis problems, other kinds of analysis have been reported. *Tandem repeats* are consecutive repetitions of a sequence of nucleotides, possibly with nucleotides changed, inserted, or deleted. They are associated with gene regulation, polymorphisms useful for identifying individuals or phylogenies [Fon04, Che05], bioterror agent identification [Jac98] and human hereditary illness, including fragile-X syndrome and Huntington's disease [Ben99].

Palindromes are paired nucleotide sequences, back to back or with a short gap in the middle, where one is the reverse or complemented reverse of the other. They largely define the structure of functional RNA molecules, are associated with some cancers, and have biological significance in other contexts. Both palindrome analysis and tandem repeat analysis are complicated by various kinds of mismatches, and by the fact that the

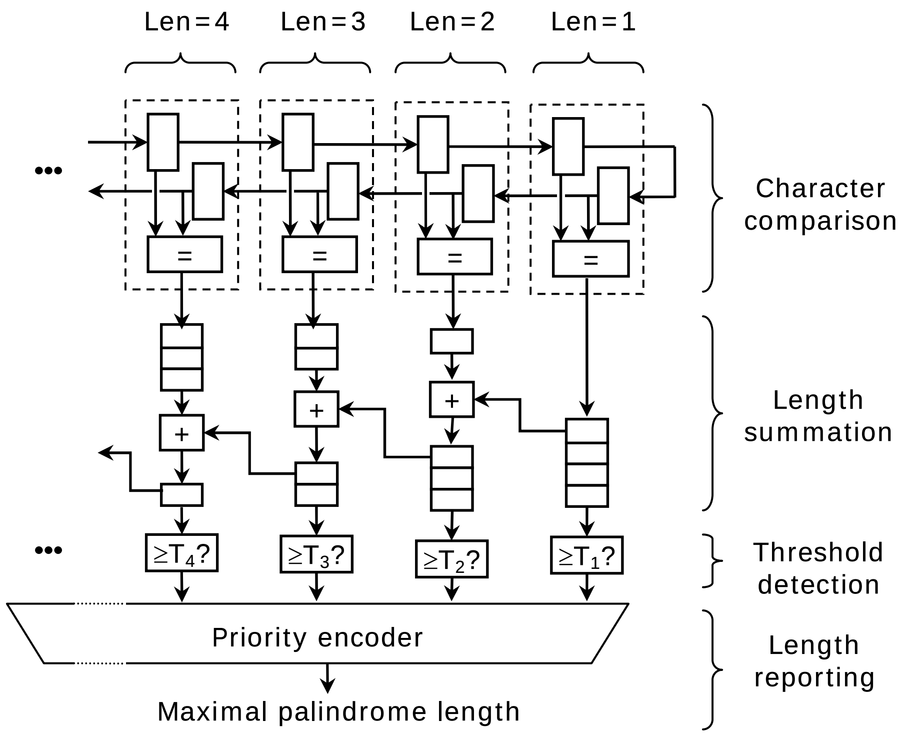


Figure 13. Systolic array for palindrome detection

length of the substrings of interest are typically not known in advance. Hardware accelerators for both problems have been built [Con04], offering computation speedups of 250× to 6000× over PC-based implementations.

Both search operations, tandem repeats and palindromes, can be implemented using linear systolic arrays. Figure 13 shows the array for palindromes. As a string streams through the *Character comparison* portion of the array, it is folded back on itself. The center of the fold is considered to be position 0. Characters at positions $+n$ and $–n$ relative to the fold are stored in the same cell and compared, giving a value of +1 for matching success or 0 for failure. The 1 and 0 outputs are summed across the length of the folded string, starting from position 0. As long as the sum at position n has the value n, then all characters from the fold to that point match and the palindrome is exact. Rather than create a summing chain (and propagation delay) the full length of the character comparison array, this implementation pipelines summation so that only one addition is performed per clock cycle, but two or more additions per level could be used instead. Summation results are lagged so that all of the length totals for a single time step exit the *Length summation* section together.

Perfect palindromes have exact character matches in each position, but biological processes often create errors in the replication. Even with small numbers of mutations, the palindrome could still have biologically significance. For example, a score of 9 at position 10, given an exact character match at position 10, means that the palindrome has length 10 allowing for one mutation. Longer palindromes might allow increasingly many mutations, reflecting any probability model that the researcher considers meaningful. The *Threshold detection* section of Figure 13 determines whether the number of matching characters identifies a palindrome or not, according to the thresholds (T_1, T_2, …) chosen by the researcher. The *Length reporting* row of Figure 13 examines the thresholding results, working right to left, to determine the first point at which the length threshold check fails – everything up to that point is part of the identified palindrome. If short palindromes, lengths, from 1 up to some minimal length, are not of interest, they need not be presented to the priority encoder. If exact matches are required, however, the entire *Length summation* and *Threshold detection* sections can be omitted and the *Character comparison* results fed directly to the priority encoder. The position of the palindrome within the input string is implicit in the time at which the *Length reporting* section's output is examined.

Figure 13's structure does not allow for insertions or deletions (indels). Although conceptually simple, the brute force implementation is tedious. In the *Character comparison* section of Figure 13, each incoming character *n* is compared to outgoing character *n*, as shown. It is also compared to outgoing character $n+1$, to detect a deletion on the inbound side with respect to the output, and to character $n-1$ to detect an insertion. If multiple indels area allowed, comparisons to $n\pm2$, $n\pm3$, etc. are added as well. Realistically, indels may be tolerated (and indel logic added) only for palindromes of some minimum length L_1, double indels may be tolerated only above some other length L_2

> L_1, and so on. The analysis array (summation, thresholding, and reporting) must also be modified in any of several ways, according to the detection and reporting policy chosen. This approach, allowing q indels in palindromes up to length L, requires qL comparators, and possibly qL columns for summation and thresholding, depending on reporting policy. Tandem repeats can be detected by a similar computation array. Figure 14 illustrates the basic logic of this array. Given some fixed length L of the rep-unit to be detected, select some point as the reference point at the end of one rep-unit and just before the second rep-unit. At the each time step, shift the entire string right by one position, to represent data streaming through the computation array. One new character moves into the array, and one old character moves out. If the new character entering the array matches the reference character, increment a counter. If the old character leaving the array matches the reference character, decrement the counter. Together, the four possible combinations of entry and exit matches can increment the counter, decrement it, or leave it unchanged. The array of Figure 15 implements this logic, folded in the middle, where each cell in the

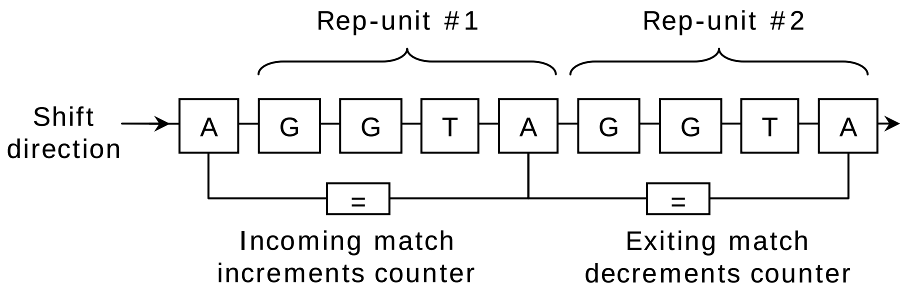

Rep-unit #1
Rep-unit #2
Shift direction
A
G
G
T
A
G
G
T
A
=
=
Incoming match increments counter
Exiting match decrements counter

Figure 14. Detecting repetions (length 4 shown)

array handles a different length *L*. Because all cells operate concurrently, all lengths *L* are checked in every cycle. Broadside output from this linear array is handled much the same way as for palindromes, including tolerance for occasional mismatches.

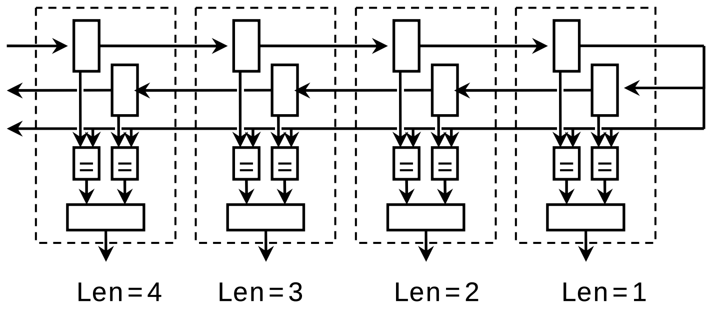


Figure 15. Systolic array for tandem repeat

Tandem repeats require different analysis from that used for palindromes. For example, counters can be added to measure the number of consecutive cycles in which a length-3 repeat (3-mer) is detected, indicating multiple back to back repetitions. Repetitions have other complexities in their analysis, as well. Suppose that a long string of repeating 6-mers were detected, and also a long string of 3-mers. The 6-mer report would be redundant, because it would just be reporting pairs of 3-mers (e.g., *AGTAGT*). Detecting 6-mers but not 3-mers would indicate that the smallest repeating unit in fact has length 6 (e.g. *AGTCCA*). This kind of logic can easily be added to the analysis array (not shown) that processes the length-specific outputs of Figure 14.

These two applications have strong similarities: the initial comparison array accepts serial data, creates large numbers of comparison results at every clock cycle, and uses vector-reduction operations to detect phenomena of interest in the input. Each task has a variety of processing options, including thresholding, indel, and minimum length policies. Another option, not shown in Figure 13, allows some gap between the forward and reverse substrings, modeling RNA structural motifs of biological significance. Dramatic speedups were demonstrated in the computational core of these applications, shown in Figure 13 and Figure 15.

The most interesting aspect of these demonstrations was in the over-all system design. These are essentially streaming applications, but the PC/coprocessor model is built around the batch processing paradigm. These applications accept data of effectively infinite length and make one pass over each string. Even with filtering, result output appear at rates up to the input clock rate. This does not make effective use of the PC/accelerator architecture under consideration. Accelerator performance is limited by IO rates from the host, presenting strings to analyze and collecting results. When that happens, it's very often the case that the host could have performed the calculation roughly as fast as it could transfer data to and from the coprocessor. Impressive performance in the FPGA's computation core does not always translate to comparable performance in the application as a whole. It can, however, be valuable in emerging hardware environments where streaming data occurs naturally [Cha05a].

2.1.6 Object Recognition in 3D Voxel Data

This addresses the problem of template-based 3D pattern recognition in volumetric data [Van05]. Volumetric data sets arise many applications, including medical imaging (e.g.

X-ray tomography, magnetic resonance imaging, positron emission tomography) and confocal microscopy. The goal of this effort is to take template-based pattern recognition, long a staple of 2D image processing, and extend it to three dimensions. This is exactly the problem addressed by the accelerator for rigid molecule interactions, with somewhat different interpretation placed on the juxtaposition of voxels in the image and template. It solves several problems that would otherwise make correlation unattractive for 3D pattern recognition:

- **Support for multispectral data types.** Confocal images, for example, commonly produce data on multiple fluorescence channels. The generality of this computing structure adapts readily to application-specific voxel data types not foreseen in the original design, and to scoring functions involving cross terms between elements of tuple values. The architecture also supports different data types for the template and image, allowing additional flexibility in specifying the pattern to be matched and the volumetric data acquired by the collection instruments.

- **Recognizing multiple instances of a pattern.** Many applications allow the possibility of more than one instance of an object appearing a 3D image, with the goal of finding all instances. Naively, that would require examination of the entire correlation result, in order to find multiple peaks indicating multiple matches to the template. This computation pipeline, however, contains a filtering stage that reduces the volume of result data by two or more orders of magnitude, but still reports multiple maxima in the result.

- **On-the-fly generation of rotated images**. Two-dimensional template matching applications often precompute rotated forms of the template image, and repeat correlation at each of the rotated orientations. Precomputation of rotated templates quickly becomes infeasible in three dimensions, where there are three axes of rotation instead of one. For example, there are 36 one-axis rotations at 10° steps, but more than 10^4 three-axis rotations at 10° intervals. There are 72 one-axis rotations at 5° resolution, but roughly 10^5 three-axis rotations. The pipeline used eliminates storage of all the rotated images, and eliminates transfers of the rotated images to the accelerator.

- **Support for spatially oriented data types**. Diffusion tensor tomography, for example, generates volumetric images representing 3D fields of diffusion vectors. This computation pipeline provides architectural support for spatially oriented voxel values, where three-axis rotation of oriented voxel elements is needed in addition to rigid rotation of voxels with respect to each other.

- **Direct handling of anisotropic sampling grids**. Scanning technologies commonly have finer resolution within a scanning plane than between scanning planes. As a result, the volumetric image is captured with non-cubical voxels. Traditionally, *"Any deviation from cubic voxel shape causes serious problems for … template matching*

> *operations.”* [Rus95] Normally, this would require a separate resampling step to convert the anisotropic axis steps into uniform ones. The address rotation logic, however, performs an arbitrary linear transform when converting the image traversal coordinates into array indices in the three-dimensional array that stores voxel values. Non-uniform scaling of axes can be combined with rotation through proper choice of transform coefficients, allowing the use of arbitrary parallelepipeds as voxels.

Addressing logic converts indices for images traversal into indices for access to image voxels using an arbitrary linear transformation. This implies that searching for matches to a template’s mirror image is also possible, with no change of accelerator hardware. Rotation and isotropic or anisotropic scaling have already been mentioned as helpful in template matching. Shear transformations, the remaining kind of linear transformation, are also possible for the indexing hardware, but practical applications have not yet been proposed.

This application makes it clear that some accelerators are applicable to apparently dissimilar applications. In fact, only one part of the application requires reconsideration. The rotated addressing logic in the rigid molecule application assumes isotropic axes for the 3D grid in which the image is traversed and for the 3D grid that stores the voxel data. This assumption affects the number of bits of precision needed for the linear transformation arithmetic that converts traversal indices into storage indices. Anisotropy of axes increases the precision required. It is striking that such different application areas require so little reconsideration of the acceleration engine.

2.1.7 Iterative optimization

Many problems, including many in bioinformatics, can be phrased as some kind of optimization: finding the best (or near-best) solution among very many. Search spaces in bioinformatics problems can be huge. Finding common motifs among k length-N strings covers a search space of size proportional to N^k, for example. The search spaces are often non-differentiable, so traditional optimization for differentiable variables is not applicable. Iterative, heuristic techniques can locate high quality solutions for many such problems, when exact optimality is not a requirement. Common techniques include:

· **Hill climbing**. This is a basic search technique, often used by itself or as a refinement step in more complex optimization algorithms. Starting from some point in the search space, a simple form of hill climbing creates one or more points nearby, along one or more search axes, and tests the function value at each point. Depending on the function value[s] found, the algorithm should step to the neighboring point with the highest value, adjust the step size and try again, or declare the search to be ended.

· **Simulated annealing**. This is just the hill-climbing algorithm, with different criteria for accepting a solution. It samples neighboring points, and accepts improved solutions unconditionally. If, however, the new solution gives a lower function value, it is accepted with some non-zero probability, usually dependent on the amount by which the new solution is worse. Occasionally accepting inferior solutions is intended to prevent the search from getting stuck at a local maximum, hopefully allowing the search to restart near another local maximum. The analogy to metallurgical annealing refers to gradual reduction over time of the chance of accepting an inferior solution, somewhat the way reduction in temperature gradually stabilizes a metal's crystal structure. Although usually phrased in terms of successive samples take one at a time, it is quite reasonable to evaluate multiple samples at each time step.

· **Gibbs sampling**. This technique searches a many-dimensioned space one dimension at a time. Each iteration samples a different axis of the search space, possibly taking many samples along that axis. As with simulated annealing, inferior solutions are sometimes accepted in order to avoid getting stuck at local maxima. This technique has been especially useful in locating functional and regulatory motifs in biological sequences [Hug00].

· **Genetic algorithms**. This technique holds many possible solutions at any one time. Each step generates a large number of candidate solutions by randomly combining parts of two solutions, or by modifying ('mutating') parts of one solution. Many different strategies are available for creating new candidate solutions and for selecting among them for the next iteration.

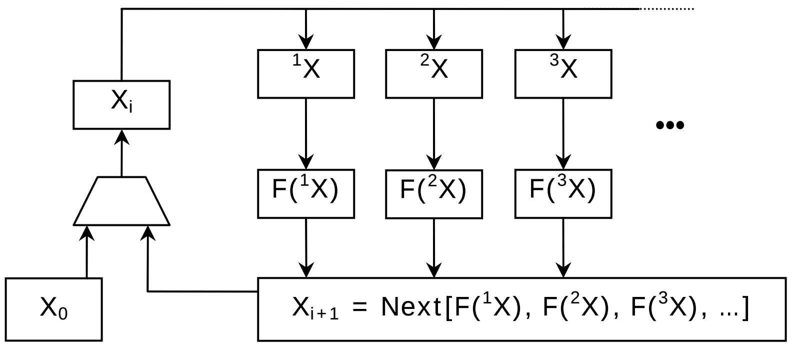


Figure 16. Computation structure for iterative optimization

All of these techniques, and others, follow the same computational schema, illustrated in Figure 16. The system starts with some initial estimate of a solution, X_0. At each iteration, new proposed solutions ^1X, ^2X, ^3X, … are generated from the intermediate solution X_i, according to an application-specific rule. Each of the newly proposed is scored, using separate instances of the scoring function `F()`. The scores are evaluated relative to each other, and used to select one of the proposed solutions as the next intermediate solution, X_{i+1}. Iteration continues until some termination condition is met.

The units labeled ${}^1X$, ${}^2X$, ${}^3X$, … in Figure 16 each differ in some parameter setting, so as to generate different updated ${}^NX$ values from a given X_i. Some rules for creating proposed solutions behave differently at each iteration, e.g. where Gibbs sampling explores different axes or in where simulated annealing applies different random perturbations. It is assumed, at least in initial implementations, that the scoring function F() is relatively simple.

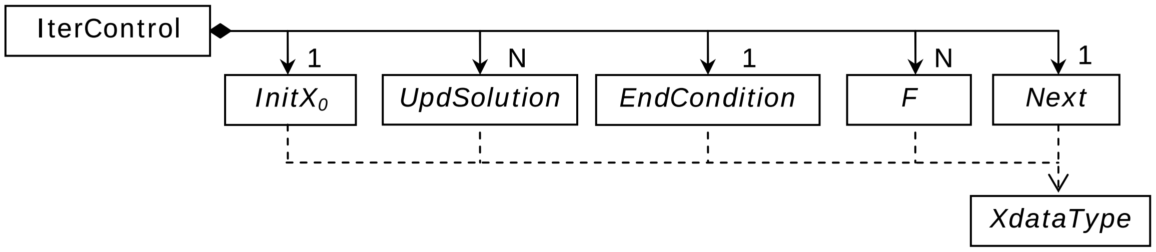


Figure 17 . Class diagram for interative optimzation

Despite its potential for complexity, this application has the simple logical structure of Figure 17. Difficult aspects of the design come from potentially complex data elements and functions. As a result, this application suggests the need for multiple levels of specialization. For example, common features of the simulated annealing computation could be phrased as a subclass that partially implements the abstract interface, with application-specific features phrased as a second level of specialization.

2.2 *Features common across applications*

Although very different in structure, these case studies have a number of features in common. The following list summarizes these similarities:

· **100-1000× acceleration is possible.** FPGAs accelerate some classes of applications by orders of magnitude, compared to a PC. They offering attractive performance without the expense, installation, and management penalties of clusters and MPPs.

· **Applications come in families.** Strings have different data types with different kinds of character comparisons, rigid molecule interaction testing allows many different scoring functions, and so on.

· **Performance requires rethinking the application.** Every level of an algorithm has the possibility of changing when recast from PC-compatible terms into an FPGA implementation. At the lowest level, fixed point data types might IEEE floating point; higher level changes include correlation by transforms on a PC, but by direct summation on FPGA platforms.

· **Application accelerators use scalable computation arrays.** Many accelerators of interest use repetitive arrays of processing elements, so that application performance depends on the degree of concurrency. Larger arrays are assumed to be better, for higher throughput, for finer resolutions of modeling grids, or for handling problems of larger sizes.

- **Communication and control are reusable components.** Although clear from these examples and familiar to software developers, data exchange and parallelism structures are not generally considered reusable hardware components. Existing logic tools handle this kind of reuse poorly, if at all.

- **Hardware expertise and application knowledge are both required.** The application specialist that who has the need for acceleration can't be expected to have logic design skills, and logic designers aren't biologists or chemists. Both skill sets are needed, to ensure that the accelerator solves the specific problem at hand and to apply the idioms of hardware design for best performance.

2.3 Conflicting demands in accelerator design

Systems for automating logic design have been under development for at least forty years, and the fundamental problem of easy specification of complex and efficient logic has not yet been solved. It is not the goal of this work to create the solution that has eluded so many other researchers for so long. Instead, the goal is to identify the conflicting forces that appear to create ineradicable tensions, and to address each conflict separately. The first contradiction is that an accelerator's logic design must be highly tuned to its specific application in order to achieve the 100-1000× speedups desired, but must still be general enough to handle wide ranges of applications. The second is that end users of application accelerators must be able to modify algorithms at will, but efficient accelerator design requires significant hardware expertise. The third is that accelerator users want to use the full capacity of the most powerful FPGAs available, but do not want to change their accelerator designs when porting to FPGAs with larger capacity.

2.3.1 Application tuning vs. general usefulness

An FPGA application must be highly tuned in order to realize the full potential speedup in a candidate application. Application specificity creates problems at two levels, however. At the lower level, users require hundreds or thousands of variations any one applications, each optimized for the unique version of the problem it addresses. At the higher level, a generally useful toolset must address problems of widely different structure and resource requirements.

The higher level of application flexibility addresses the desire to handle problems of very different character, not just variations within one family of closely related problems. Block generators of various types have existed for years, and are still and important part of commercial logic design. They tend to be highly specialized, however, and unable to handle functions even a little different from the ones originally targeted. Such systems sacrifice generality for optimized performance in a single problem domain. General purpose HDLs represent the opposite end of the spectrum: ability to handle any kind of application, with no explicit support for any one application domain. Domain-specific languages represent an intermediate level of application specificity and optimization, but addressing multiple different problem domains would require multiple different languages.

At the lower level, consider variations on the Needleman-Wunsch string matching algorithm. There are three major ways in which variant algorithms differ from each other. First is the component that defines the character rule. This embodies the type of each character in the string. It also defines the substitution matrix that rewards exact or near matches and penalizes mismatches between two characters, which creates another level of customization after the string data types are chosen. Substitution matrices are usually symmetric, but not necessarily, and may be procedural functions rather than declarative lookup tables. Note that the two strings need not have the same type: in one example, a query string consists only of the four nucleotides in DNA, but the reference string consists of IUPAC ambiguity codes (wildcards), requiring a different number of bits for representation. The second difference between algorithms is the matching cell, the component that implements one unit of the 2D recurrence relation by which whole strings are compared. Any matching cell can work with any string rule, since the recurrence relation depends on matching scores and not on the type of the strings being matched. Some matching cells allow variant behaviors, such as the 16 possible end-overlap rules in the Needleman-Wunsch algorithm. The highest level component is the sequencer, which controls the basic flow of string data and matching results through the system.

One solution would be to create a different and highly tuned accelerator for each variant on the problem. There are thousands of variations, though. Creating a new accelerator for each one, in advance of demonstrated need, would be repetitive and tedious. It would be wasteful, since combinatorial completeness would create systems that never get used. It would also be limiting since biologists create new substitution matrices all the time, and a novel matrix is inherently impossible to implement in advance.

Another solution would be to create a highly generalized matching engine, able to handle any of the known variations of the problem. Substitution matrices could be implemented as RAM lookup tables, data paths would be allocated for the largest character size, and so on. This would be inefficient for any one application, however. DNA strings require only a two-bit alphabet, but codons require six bits. If an accelerator had the capacity to support codons but was used for nucleotides, two thirds of the character string data path would be wasted. Also, RAM tables are slow compared to logic, and may become the limiting resources in an FPGA implementation. Exact-equality comparisons between strings can be implemented efficiently in logic. That would allow faster circuitry, would release RAM resources for other use, and would allow more character PEs by evading the RAM resource limit. And, no matter what fixed-size data path were chosen, it would preclude some applications of interest. A six-bit limit would prevent the string processor from handling ASCII text; an eight bit limit would lock out internationalized Unicode applications. Any fixed solution would, ironically, be too general for maximum performance at the same time that it was too specialized to handle all applications of interest.

Creating a highly tuned accelerator on demand is the ideal. Although the number of distinct applications in a family is large, the number of choices to make is modest. Intelligent tools should recommit resources according to application requirements. For example, a narrower data path generally requires fewer logic elements per PE. Reducing the number of gates per PE not only reduces the logic delay, in many cases, it also frees logic resources that can be used to create larger numbers of PEs. Likewise, a generalized PE would require resources for choosing between its potential modes of behavior. A hard coded application can recommit those resources away from configuration control and towards the application's payload logic, and also reclaim any logic resources used only in other operating modes.

Even without considering configuration resources, experiments have demonstrated a 4:1 performance difference between complex comparisons of protein sequences and simple nucleotide comparisons [Van04a]. The most complex of these string comparison accelerators is also the most general, and could be used to implement the simpler operations. If that were done, however, the simplest application would run at only ¼ of its potential speed.

All of these approaches presume one step of abstraction, from the language or design system primitives to the synthesized system. Two or more steps of abstraction are sometimes more effective: first, a highly generalized set of primitives for system specification, then one or more levels of increasing application specificity and domain knowledge, then finally the synthesized result. This, in fact, is the basis of any OO language with inheritance: specialization of general utilities by means of application-specific subclassing, with as many different levels of specialization as the programmer desires.

2.3.2 Application customization vs. hardware-based implementation

Application specialists have varied and changing computation demands, even within a single application family. It seems clear that good application performance demands a hardware implementation tuned to the application specifics, but very few application specialists have the hardware design skills needed for making changes to the logic design of an FPGA-based accelerator. Accelerator design tools create this conflict by offering bit-level implementation tools to developers working at the algorithmic level.

The *semantic gap* is the name given to the conflict between high-level representation of computational problems and high-efficiency implementation in some technology [Sni01]. Traditionally, this has meant the gap between a high level language's semantic definition and the instruction set of the machine into which a program is compiled. The semantic gap has long been recognized in the software development world, and has led to fruitful cooperation between compiler writers and instruction set designers.

The semantic gap between application accelerators and FPGA fabric is even wider. The application-specific logic of an accelerator operates at a higher level than the logic of a programming language, because it embodies so many more assumptions about the meanings and basic techniques implied by the application. At the same time, the FPGA fabric operates at a lower level than the processor's instruction set. For example, FPGA hardware requires selection of data word size, something fixed in a standard CPU, and often requires new word size choices at many points in the design.

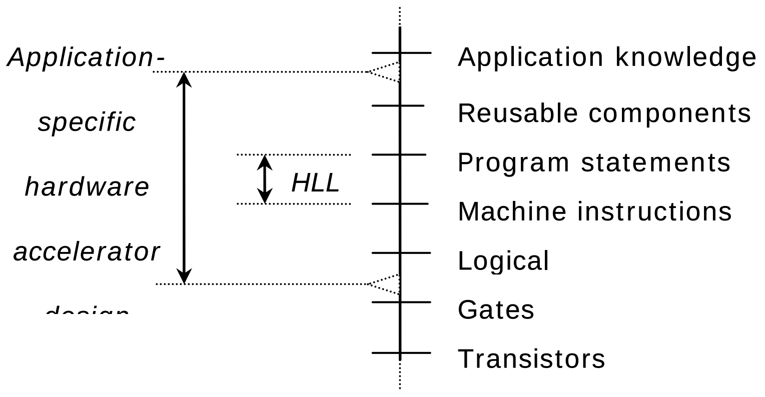


Figure 18. The semantic gap: high-level design vs. low-level implementation

It seems safe to assume that the semantic gap will remain a fact of life throughout the visible future. Despite the optimistic claims of some tool developers, a clever logic designer can often create a computation structure unimagined by the tool developer, and therefore inaccessible to the tools. Biologists, chemists, and other potential users of application accelerators can not be expected to equal a logic designer's creativity in using logic resources, or even basic competence in creating systolic arrays, reduction networks, and other native kinds of hardware solutions. As a result, one commercial accelerator builder asserts that it is necessary to "*[adapt] established algorithms to run on our FPGA Accelerator Arrays by reinterpreting from first principles*." [Tim05] There is both practical and theoretical reason to believe that direct compilation of standard algorithms does not generally synthesize into the most effective hardware implementations.

Since the semantic gap can not be closed, in the general case, it is necessary to address application development in terms of that gap. This acknowledges the application specialist (with high-level knowledge) and logic designer (with low-level design skills) as different individuals. They differ in their needs and responsibilities, and must work together to create the application accelerator. It follows that the design tools' responsibility is not to *convert* application specifics *into* a logic design, but to *connect between* application specifics and an efficient logic design.

Skilled logic designers are relatively scarce, and are not likely to become more common in the near future. As a result effective tools must not require logic design skills for making minor, routine changes to an application accelerator. The logic designer's contribution must be highly reusable, so it can serve many application specialists without the designer's intervention for each one. Finally, in order to decouple the logic designer from the application specialist, it must be possible for them to work at different times.

2.3.3 Exploiting the fabric vs. reusability

Whatever the computation array and FPGA capacity, the accelerator designer generally wants one thing that no current design tools are able to state explicitly: *as many PEs as possible*, given the resource utilization per PE and FPGA capacity.

Current design tools require that exact numbers of PEs be specified as inputs to the design; they have no direct support for open-ended parallelism. The desired number of PEs, the largest number possible, depends on other system inputs: the amount of logic required for each PE, the allowable sizes of the computing arrays, and the logic capacity of the FPGA platform. If the application details change so that a different amount if logic is required to implement the repeated PE, then the number of PEs could change, also, up to the limit defined by the FPGA's capacity.

Given the rapid pace of FPGA product development, it seems likely that useful FPGA-based applications will outlast one or more generations of acceleration hardware. Users of PC applications expect performance improvements when they recompile their applications for new processors; users of FPGA applications will surely expect the same. FPGA accelerator speed depends largely on the degree of parallelism, so performance improvements on larger FGPAs depend largely on the increased number of PEs that the new FPGA can support.

Although the highest possible degree of parallelism is a consequence of other design inputs, current design tools require the developer to provide that number as a system input. That, in turn, requires gate-level knowledge of the PE and accelerator design, the kind of knowledge an application specialist is expected not to have. Instead, accelerator design tools should automatically expand an accelerator to the maximum size allowed by the computing structure, application details, and FPGA capacity. For any one application family's computing structure, that size should change automatically in response to changes in application-specific details or in the FGPA platform.

3 Previous Work: BCB Computing and FPGA Design Tools

This chapter examines existing systems for high performance computing in order to characterize the advantages of FPGA accelerators relative to competing technologies. The first section within this chapter examines the kinds of hardware systems that have been used in high performance computing, showing a variety of advantages of FPGA-based accelerators. It is neither feasible nor useful to present the complete history of high-performance computing systems, nor of the applications areas in which they have been applied. Instead, section 3.1 offers representatives of the kinds of computing hardware systems that have been used for BCB problems. The focus on BCB is not meant to exclude other application areas, but to focus the kinds of systems currently in use for the application areas that have been examined in detail.

Next, section 3.2 presents a survey of logic design systems that have been used for specifying the structure and behavior of FPGA-based systems. As in section 3.1, the goal is not to present a complete history of the field. Instead, this section shows how traditional kinds of logic design tools have evolved to meet needs vary different from those in accelerator design. Many different conceptual bases have been proposed for logic design systems, but it will be seen that none address the full range of demands placed on them by the applications described in chapter 2.

Finally, section 3.3 examines object orientation as it applies to hardware description languages. This shows that object orientation in fact covers a wide range of linguistic concepts, many of which must be reconsidered when applied to hardware description languages. It also presents a range of alternative and complementary semantic constructs available for the design of OO hardware design tools.

3.1 Hardware systems

BCB problems have huge commercial and scientific value, but also huge computation loads. As a result, just about every kind of processing architecture has been applied to BCB problems, including standard processors and supercomputers, ASICs for specific problems, graphics processing units (GPUs) retargeted to general computation, and FPGA-based configurable logic. There have been so many different efforts within each category that it is impossible to list them all. Instead, this section describes a few systems that should suffice to represent each group.

Hardware systems have been built for computations other than BCB applications. Even within the BCB field, there are many omissions. Excessive detail would not meet the current need, however: to examine the kinds of hardware solutions that have been implemented and to determine which factors in each one define its strengths and weaknesses. Similar systems have been omitted in the interest of brevity and clarity.

3.1.1 Serial processors, clusters, grid computing

PCs based on Linux, Windows, or other operating system, are cheap, distributed, and flexible. Large amounts of PC-based software are available, freely, commercially, or for academic use, and PCs probably contribute the largest number of CPU hours to the BCB total. The drawback is computational power: it is common that algorithms of up to only $O(N^2)$ complexity (for common data sets) are run on large data sets. This means that PC applications make liberal use of assumptions and heuristics to limit computational complexity. Also, the complex higher order computations described above are completely out of reach for PCs.

In 2002, only five of the world's top 500 supercomputers [Top03] were said to be dedicated to biology, pharmaceutics, and the life sciences. This statistic probably under-represents the number of supercomputers in bioinformatics. About half of the systems listed had no assigned application area, and over 200 were listed as research or academic machines – the load mix in those machines could well include BCB applications. Other applications (e.g. chemistry or databases) could also support BCB computing.

IBM's Blue Gene project was originally targeted for protein folding applications, but can be used for any parallelizable computations. Blue Gene/L's maximum configuration is intended to provide 360 TFLOPs, using 65,536 PowerPC processors connected in a 3D torus, and a total of 32TB RAM [Gar05].

SIMD arrays have also been successful on some problems [Bla90, Bor94, Gra01]. To date, none have had lasting commercial impact. Still, new SIMD processors continue to appear. For example, ClearSpeed's PCI-compatible accelerator uses the CSX processor, with 96 "poly" (SIMD) processors. CSX-based CSX600 boards have been put to work in

GROMACS molecular modeling systems [Cle05] and in Bristol University's Dockit [Cle05a].

Cray has showcased AMBER's molecular modeling as a successful application [Cra00]. Silicon Graphics Inc. addresses biocomputing with their Origin supercomputer family. They publish benchmarks and free executables for a number of popular applications, including BLAST and CLUSTAL W [Sil01].

Clusters represent a second approach to massively parallel computing, not always distinct from multiprocessor computing, except that some clusters support heterogeneous processor types. Compared to typical multiprocessors, cluster architecture exacerbates three problems: system administration and management, load distribution and scheduling, and IO sharing and data distribution. Cluster vendors address these issues explicitly, and often with the same tool suite [Sun02a]. Administration issues are more complex than traditional LAN management, because of the need to harness many systems to one task. For many installations consist of multiple clusters, geographically distributed cluster management covers several ranges of administrative control and communication cost [Sun02b]. IBM's cluster control software, for example, includes HACMP for local cluster management, Workload Manager for load distribution, GeoRM for synchronizing data at different locations, and HAGEO for resilience against cluster loss [IBM02, Sun02].

Cluster performance is up to *P*-times better than a PC for *P* processors. For small clusters (fewer than 16 nodes), the hardware cost can be slightly less than *P*-times that of a PC. Larger clusters, however, e.g. those with more than 64 nodes, often have special housing, network, power, and cooling requirements. Those costs, plus network administration, add to the cost of ownership without adding system capability. Fujitsu Computer Corporation and Orion Multisystem have both addressed the need for office-environment processor clusters. Orion's DS-96 offers 96 processors in a deskside box [Ori05], and Fujitsu's BioServer offers 128 processors deskside or up to 1,920 processors rack mounted, without need for special power or environmental control [Fuj05]. Still, there is far less cluster software available than uniprocessor software, and programming for clusters is a relatively rare skill.

The many varieties of Unix clusters include supercomputers and MPPs, such as those at national centers and commercial server farms [Adi02, Cel00, Cra00, Sil01, Sun02]. As with PC clusters, performance is up to P times better for P processors. Because of tight coupling and homogeneous processors, scalability and ease of programming are often better than in heterogeneous grids and irregular clusters. MPP costs are high, though, both in acquisition and system maintenance, and the convenience often low. Researchers must apply for supercomputer time, scalability studies are often required, programs must be submitted batch with substantial turnaround times, and users are rarely permitted more than a small fraction of the resource.

Special software addresses the problems of multiple servers, replicating data, striping a single file across multiple servers, and managing redundant paths between files and clients [IBM02a]. Higher-level services organize disparate types of biological data into coherent “warehouses”. Whole companies exist selling only software for load distribution and system management [Pla02]. Even acceleration hardware is sold as a cluster component [Tim02], rather than an isolated solution.

Grid computing extends the ideas of cluster computing to a much larger scale. Where clusters normally consider of physically close, heterogeneous processors, grids often span geographic regions, administrative domains, and processor types. This introduces new problems not seen in local, homogeneous cluster configurations, including relatively large and unpredictable networking latency and bandwidth issues. The Globus Toolkit is under development by a non-profit consortium of industry and academic users. It works within the Open Grid Services Architecture to standardize APIs and services for distributing computation [Glo02]. Other groups are working to standardize grid practices for management, scheduling, etc. [GGF02].

A related concept has variously been called “utility computing”, “pay-per-use computing” [Sun05], or “deep computing capacity on demand” [IBM03]. It’s based on a conventional server farm or MPP, but is generally owned by one organization for use by other organizations. The system’s owner sells CPU time to clients that have only occasional need for massive computation. This isn’t really a difference in technology, however, but a difference in ownership and business models around the computation hardware.

3.1.2 Graphics Processing Units – GPUs

GPUs, driven by the multibillion dollar gaming industry, were originally developed for fast rendering of scenes containing complex geometry, color, and other features. Current GPUs have dozens SIMD and MIMD computation pipelines [3Dl04], specialized for different kinds of calculations, supported by on-board memory busses up to 512 bits wide. Their primitive data types commonly include booleans, 32-bit integers, 16- and 32-bit floating point, and possibly a fixed-point format [Ros04]. Typical GPUs have hardware support for vectors to length four [Eng04, Ros04, NVI05], though larger vectors and arrays can be defined. Despite their non-standard and profoundly constrained computing capabilities [Gra03], their raw speed and modest cost have made them attractive as general purpose computation engines - GPGPUs. To help ease the task of creating GPGPU applications, the Brook language has been proposed as a vehicle for encoding general computations for GPGPUs and similar systems [Buc04].

GPGPUs have been programmed to perform phylogenetic tree-building [Cha05], matrix operations [Mor04], computation of Voronoi regions [Hof99], database operations [Gov04], distance fields [Sud04], statistical summaries [Gov05], 2D FFTs [Mor03], and other operations. GPGPU surveys [Man05, Owe05, Tra05] list these applications and a variety of others.

GPU clusters have been built [Fan04], but most research has been done in the hardware context of a typical PC graphics accelerator. PC configurations with multiple

GPUs have been marketed, however, with the goal of improved graphics performance. According to the company product documentation, 3Dlabs' Realizm 800 graphics card contains two GPUs, called "visual processing units." This configuration includes a processor ('vertex/scalability unit') with multi-GPU coordination responsibilities [3Dl04a]. ATI sells a two-GPU PC motherboard based on the PCI Express bus standard and their Crossfire bus communications chip [ATI05], similar to Nvidia's two-GPU motherboard built around their "scalable link interface" (SLI) and "media and communications processor" (MCP) [NVI05a]. Evans and Sutherland's simFUSION simulator product uses up to four ATI GPUs [Eva03]; their RenderBeast product line puts up to 64 GPUs in one server [Eva03a].

As demonstrations of GPGPU interest, day-long courses on GPGPU processing were presented at SIGGRAPH 2004 [Har04] and 2005 [Har05], and at IEEE VIS 2004 [Lef04]. Term GPGPU courses have been presented at The University of North Carolina at Chapel Hill, University of Aarhus (Denmark), CalTech, and the University of Pennsylvania.

3.1.3 ASIC-based systems

Special-purpose computers have a long history, even within bioinformatics. Some research systems have been dedicated specifically to BCB computations. Other research systems (e.g. Kestrel) have had general-purpose capabilities, and have addressed BCB problems as targets of opportunity. Yet other systems have been highly specialized for particular computations with BCB applications, but sometimes work in other application areas as well. Hardware systems for self-organizing maps (SOMs) fall into that last category.

Self-organizing maps represent one family of unsupervised clustering algorithms, noted for heavy use of matrix multiplication and comparisons across vectors. SOMs are not exclusive to bioinformatics, but have found some use in the field. Kohonen [Koh01] reviews a number of hardware accelerators for SOM algorithms. Kohonen notes that, to make implementation feasible, some hardware implementations accept limits: maximum size of data handled, suboptimal communication paths, limited precision, or simpler but less-desirable algorithms. Although SOM speedups were impressive, ASICs implemented as late as 1997 were still limited to 16MHz clock speeds. It is also interesting that all applications reviewed used fixed-function ASICs or general-purpose processor ensembles, not FPGAs.

Shortly after publication of the Needleman-Wunsch (NW) DP algorithm for DP AM in 1970 [Nee70], it became the *de facto* standard technique for AM in biological sequence matching. Because of the computation's regular structure, limited data types, and simple computation element, it has been a target for hardware acceleration for at least two decades [Lip86, Lop87, Cho91, Hoa93, Bor94, Blu00, Yu03]. It also became the target of many variations, including the Smith-Waterman technique for substring alignment, “end space free” variants [Gus97], Smith-Eggerton repeated alignment, and a theoretically unbounded number of different gap-penalty strategies [Dur98].

Customized implementations of general-purpose processor arrays have also been applied to bioinformatics applications. The USCC Kestrel project [Dah99, Hir96, Hug95, Mes01] is one such system. It is a SIMD array of eight-bit processing elements (PEs), 64 to 512 of them in different implementations. Each PE shares a register bank with two neighboring PEs. This effectively connects the array into a bidirectional ring. The architecture is reportedly designed for comparisons like Smith-Waterman alignment, but executes hidden Markov model and other algorithms as well.

SAMBA (Systolic Accelerator for Molecular Biological Applications) [Lav96, Lav98] is another experimental processor tailored for Smith-Waterman string comparisons. It claims an array of 128 twelve-bit processors. Although the processors themselves are fixed-function ASICs, their interconnection appears to be a reconfigurable FPGA.

These are just some of the recent architectures designed or adapted for bioinformatics applications. They have many predecessors, though, including:

· P-NAC (Princeton Nucleic Acid Accelerator), one of the earliest well-known LSI implementations aimed specifically biological computations [Lop87]. This 1987 system used multiple chips, each with 30 PEs, and implemented a dynamic programming for of string comparison.

· BISP (Biological Information Signal Processor) followed a few years later, in 1991 [Cho91]. It also accelerated dynamic programming for string comparison. Initial configurations used 16 chips, each containing 16 PEs. Even at 12.5MHz, the system outran a Cray-2 by a factor over 100. The authors claimed a cost/performance ratio well over 104× better than the Cray. The system could, theoretically, have been expanded to 218 chips with near-linear speedup for long comparisons.

· RFDH-1 [Fai93], able to compare up to 2048 pattern characters to an input string. Although this implemented only shift-and-add comparisons, it was able to handle several patterns at a time, and had some ability to handle regular expressions.

· The BioSCAN processor [Sin93], consisting of 16 ASICs with 812 processors in each. Although it handled parallel comparisons against patterns up to 12992 characters, it had no explicit support for gapped comparisons. It did, however, support comparisons with different matching scores for each possible pair of characters. 16-bit scoring values are more than adequate for representing common BLOSUM or PAM matrices.

· The MGAP (Micro-Grain Array Processor) claimed over 0.75 teraop of performance when coupled to a 1995-era desktop workstation [Baj94, Bor94]. This unusual architecture consisted of a rectangular mesh of 214 one-bit processors, configurable as words of various sizes. Dynamic programming was only one of the applications demonstrated on this system.

· The RAPID-2 processor [Arc94, Fau95], a SIMD processor. This was based on an associative memory model of computing. This allowed matching algorithms in the dynamic programming style, as well as many other kinds of computations, but required an unusual programming model.

· String-matching accelerators, often designed for non-biological applications, cover more than two decades of hardware technologies [Muk79, Fos80, Lip85, Muk89, Du94, Sas95, Ran96, Lee97, Ran7, Meg90, Smi90, Tim02, Che03, Mac04, etc.]. String matching has been a favorite target for acceleration because of the bounded nature of the problem and high regularity of elementary computations.

· Stanford's Merrimac [Dal03] is being built around custom "stream processing" chips. Although not specifically a BCB engine, it is being applied to a customized version of the GROMACS molecular modeling application [Ere04].

Paracel was one of very few companies to commercialize ASIC BCB accelerators. Its GeneMatcher systems performed matching operations using dedicated ASICs configured in chains [Ull00], claiming over 1000× speedup [Par01]. Paracel systems were *"Designed as a high-performance network server,"* to be shared by multiple members of one work group. It is interesting that Paracel was spun off from TRW to commercialize the fast data finder (FDF) chip, originally a classified weapon in the NSA's Cold War communication intelligence arsenal [Yu88].

The MD GRAPE family of ASICs has also been commercialized. The original GRAPE processors of the 1980s were designed for handling force calculations in astronomical N-body calculations. Current generations are intended for molecular dynamics (MD), and appear in commercial, four-chip PCI-bus accelerator cards [Con04a] rated at 64 GFLOPS. Related chips are also being built into other systems, including the ambitious Protein Explorer, a 128-node PC cluster expected to contain 5,120 MD GRAPE 3 chips and to reach 10^{15} FLOPs performance levels [Tai02].

3.1.4 Reconfigurable processors

FPGA coprocessors are not the only way to combine configurable logic with a processor of fixed structure. The Garp processor architecture integrated a MIPS-based processor with configurable logic [Hau97]. Its configurable elements were aligned with the fixed-function data path of the MIPS CPU, in order to ease insertion of new functions into existing logic structures. PipeRench [Gol99] is similar, in containing a configurable logic array tuned for data path applications, but without a fixed-function processor. The Tensilica [Ten04] processors, with their Tensilica Instruction Extension (TIE) features, and Philips' discontinued TriMedia TM1100 [Phi99] processor follow similar strategies, but differ dramatically in the amounts of reconfigurable resources available.

The Xilinx Virtex-II Pro FPGAs are somewhat opposite the Garp structure: they consist of a fabric of 10^3-10^5 fine-grained configurable logic cells and a few dozen to a few hundred dedicated multipliers and RAM buffers. One or two PowerPC cores [Xil04] are embedded within that fabric, normally for high-complexity and low-speed control or computation functions. Members of the newer Virtex-4 product family [Xil05] vary in logic capacity, but also in the balance between fine-grained configurable logic, dedicated arithmetic functions, IO, and hard PowerPC cores.

Soft CPU cores can also be implemented on most modern FGPAs, and customized as desired. ARC International's product line [ARC05] consists entirely of synthesizable processor cores. The distinguishing feature of the ARC products is that they are designed with the specific intent that the customer modify the cores by adding application-specific instructions.

Many other commercial and experimental combinations for fixed-function CPUs and configurable logic have been proposed, far too many to enumerate here [Har01, Com02a]. As shown, they span a wide range of system- and chip-level architectures, representing many kinds of configurability and relationships between fixed and configurable computing elements. To date, they have generally addressed high-volume applications, especially multimedia and communications. They are, however, well suited to BCB applications as well.

3.1.5 FPGA systems

Reconfigurable accelerators for fixed-configuration processors have been studied for over forty years, starting with systems based on logic modules built from discrete transistors and diodes [Est63]. In the early 1970s, the Macromolecule Modeling System was built, using MSI logic modules, specifically to handle BCB applications using reconfigurable logic attached to a LINC processor [Ell73]. The author noted that "*experiments were carried out largely by temporary hardware modifications which were easier to effect than program changes.*"

Since shortly after their commercialization, the reconfigurable logic in FPGAs has attracted attention as an engine for general computation [Gra89]. Today, the list of FPGA accelerators includes commercial PC coprocessor boards [Ann05, Gid05, Nal05], fixed configurations of PCs and multi-FPGA coprocessors [Sta04], FPGAs in clusters [Bar04, Tom05] and large-scale clusters of FPGAs [Cha03, Gid05, Nal05a, Syn05]. There is also a long history of research-oriented systems. The Splash-1 VME board set [Gok91, Hoa93] was an early and well-known example, based on 32 Xilinx XC3090 FPGAs in a serial systolic array, and was specifically intended for sequence comparison algorithms.

Turn-key BCB accelerators are available in the TimeLogic [Tim05] product line from Active Motif. These consist of clusters with dedicated FPGA-based accelerators, claiming 100-1000× speedup, depending on the application [Tim02a]. These have had only moderate success because of high cost and low flexibility. They are packaged as self-contained units, with dedicated processors and storage [Tim02]. Perhaps even more importantly, these systems can only run those applications supplied by the vendor – BLAST, HMMer, and Smith-Waterman. Only the vendor or close business partner can modify the algorithms being used. Biocceleration BioXL and Bioccelerator product lines (formerly sold by Compugen Ltd.) are comparable [Bio05], and also FPGA based. Like Paracel's products, Time Logic and Biocceleration market their accelerators as shareable, networked resources. Since these systems are based on commodity FPGAs, they have the advantage of being able to track improved fabrication technologies more readily than ASICs [Tim02b].

Commercial processors from SRC [Boh04] and MPPs from Cray and SGI now have FPGAs tightly integrated into their processor nodes [Cra05, Sil04]. In addition, the FPGA High Performance Computing Alliance (FHPCA) was launched in May 2005, with the goal of creating a TFLOP computer based on reconfigurable logic [Cla05].

3.1.6 FPGAs vs. other computing systems

Single, serial processors are by far the most common computers available, but simply do not have the capacity to perform many important BCB computations in reasonable time. Clusters of N nodes offer (at best) N-fold faster computation. Cluster costs are also typically N-fold as high as for single nodes, and often more: they often incur additional costs for controlled environments, power supply, and system administration. MPPs and supercomputers, often clusters themselves, have many of the same drawbacks. The largest supercomputers now require special building construction, multi-megawatt power supplies, and air conditioners rated upwards of a hundred tons of cooling capacity. Total costs must also include obsolescence and replacement every few years. Although SPMD systems give good speedups for a wide range of applications, programmers able to exploit their full capacity are relatively scarce and often unfamiliar with the application areas being addressed.

PC-class processors have shown and continue to show impressive performance gains over time. Many of these gains have come from larger on-chip caches, branch prediction, speculative execution, and related additions to the processor core. Although helpful, it is interesting that these features are not directly related to the CPU’s nominal task: carrying out logical and arithmetic operations. Likewise, storing more cache lines on chip suggests that proportionally fewer cache lines deliver payload code and data values per memory cycle. Even on-chip, cache access is throttled by the von Neumann bottleneck. Both factors mean that processor performance per transistor had plummeted as more transistors have been added to the processor chip. Successive generations of FPGAs add transistors also, but a larger percentage of the added transistors are visible as added logic elements, communication resources, and independently accessible memory elements. FPGAs come much closer to the ideal of linear increase in computing capability with increase in transistor count.

FPGAs vs. GPGPUs

GPGPUs are emerging as affordable processors delivering performance on the order of 10^{10} FLOPS today and 10^{11} within a few years, for selected applications. They are readily available in PC boards with retail prices of a few hundred dollars – on the order of US$10 per GFLOPs and falling. The GPU programming model is severely constrained, however, and GPGPU programming is a narrow sub-specialty within the specialty of GPU programming. Relatively few non-graphics applications have been mapped into GPU programs, and those usually appear to be isolated acts of cleverness. The GPU's basic programming model is of interest, however. The user provides relatively simple leaf-node computations to the processing system, roughly corresponding to a scalar computation to be performed at each element of a vector or array. This allows the GPU complete freedom in organizing high degrees of parallelism and in accessing operand and result memory for best throughput. It also makes performance improvements automatic when a better-performing GPU becomes available, since system-specific scheduling and data access are managed separately from the application logic, and can be handled differently according to the GPU's different internals.

GPGPUs offer exceptional performance for some kinds of numerical computations, and offer floating point capabilities that current FPGAs lack. Still, GPUs achieve their demonstrated levels of performance by constraining the kinds of computations that can be supported. Clever implementers have put GPUs to use in a number of surprising non-graphical computations. In some cases, it appears that the GPUs' raw computational power makes even indirect and inefficient implementations run well: a 10 GFLOPs processor run at 20% capacity still beats a 1 GFLOPs processor 2:1. Despite impressive demonstrations, GPGPU applications are still isolated rarities, achievable only by programmers with unusual skills.

FPGAs offer advantages over GPGPUs in many respects. The biggest is their potential for generality: the FPGA application is free to use any data types, pipeline structure, memory access pattern, and internal communication network desired. Multi-GPU configurations exist, but are rare. Even two-GPU configurations require specialized processor boards, creating doubts about the scalability of multi-GPU solutions. Multi-FPGA configurations exist commercially, as add-ins to existing host hardware, and current FPGAs come with numerous serial links to 10Gb/s, plus hundreds of general-purpose pins that can be configured for communication. Their hardware, at least, is readily scalable far beyond existing multi-GPU configurations.

Also, although skilled FPGA developers are hardly common, there is an existing base of logic designers with FPGA skills. It's possible that there are larger numbers of programmers with GPU experience than there are FPGA developers, but GPGPU skills are still extremely rare.

FPGAs vs. ASICs

ASICs deliver the best performance for a given amount of silicon. They offer wide flexibility in the kinds of computations available, up to the moment of manufacture – once built, however, the function is fixed permanently. ASIC design is a rare skill, and nearly never found among BCB application developers. Non-recurring costs for a new chip are high, also: said to be about US$10M for high-end ASICs, and US$1M for an all-new mask set [Bur05]. These costs have to be amortized over the production run of the chip, and must be spent again, at least in part, each time a new ASIC or variation is required, including changes to take advantage of fabrication process improvements. As a result, ASICs make sense in applications:

· where incremental performance improvements justify large costs, as in some defense applications,

· where huge production runs amortize initial costs over many units, as in many consumer products, or

· where exotic devices are required, as in some research efforts.

None of these descriptions match the cost-sensitive and fast-changing computing systems needed for commercial use.

FPGA-based systems offer many advantages of ASIC-based systems, but at far lower cost, partly because one FPGA mask set is reused by many customers. FPGA designs can be simpler than ASIC designs, partly because only the application logic needs to be specified, where ASIC design requires attention to many more circuit- and device-level details. FPGA designs can be simpler than ASICs in some case, because any errors or omissions in an early version of a design can be rectified later. The initial design does not need to anticipate all future requirements. FPGAs are reprogrammable at run time, so one chip can be time-multiplexed to accelerate many different applications. Designers with FPGA skills are rare, but somewhat more common than ASIC designers. The current trend towards reconfigurable logic in mainstream computers makes it likely that application development will become easier, as the computer developers create tools to make their products attractive.

3.1.7 Summary: advantages of FPGA accelerators

In summary, FPGA-based accelerators seem to offer the preferred combination of benefits:

· super-computer or cluster levels of performance without the expense and administration,

· ASIC-like compactness and application-specific efficiency, but with lower costs in tracking technology and with higher flexibility, and

· GPGPU-like performance and integration with familiar PC environments, but higher flexibility in the kinds of computations supported.

FPGA accelerators now cost more than commercial GPU hardware, but those costs should drop if they reach the manufacturing volume of graphics cards. In all other ways, however, FGPA hardware offers attractive advantages over other kinds of computing platforms.

Application developers nearly never access the hardware levels of their computing platforms, however. Processor hardware is accessed through compilers for high-level languages. GPUs are used with elaborate graphics libraries and device drivers. For FPGA hardware to be attractive as a system for implementing applications there must be comparable kinds of development and application-support tools. That issue is addressed in the section that follows.

3.2 Logic design tools

This section summarizes the current state of software tools for analyzing and designing FPGA-based systems. The primary goal of this survey is to examine the conceptual bases of logic design tools at a coarse level, in order to learn how tools for creating application accelerators might need to differ from logic design tools.

First, this gives a brief summary of the kinds of tools commonly used in creating FPGA designs, and how different tools are combined for handling different parts of the logic design process. Next, this presents a review of the HDLs used by the large majority of commercial logic designers. The remainder of this section gives a brief survey of the kinds of design tools used, at least experimentally, in recent years. These include numerous HDLs without object orientation, standard programming languages used as logic specification languages, design based on commercially available libraries, mixed-language approaches, visual design entry, tools for partial reconfiguration of FPGA logic, and object oriented design systems.

3.2.1 Typical design flow

A suite of logic design tools is often described as a *flow*. The same term often applies to the practice of using those tools. Figure 19 outlines the data and tools in a typical, possibly simple tool flow. The logic design is specified using one or more different inputs and tools, including hardware design language (HDL) source code, circuits entered visually using schematic capture tools, outputs from special-purpose block generators, and black box instantiations of commercial IP blocks.

Details vary between different design flows, but the next step typically converts the input format into an abstract logic design. Many optimizations are possible at this point, including constant propagation, elimination of redundant and unused logic, etc.

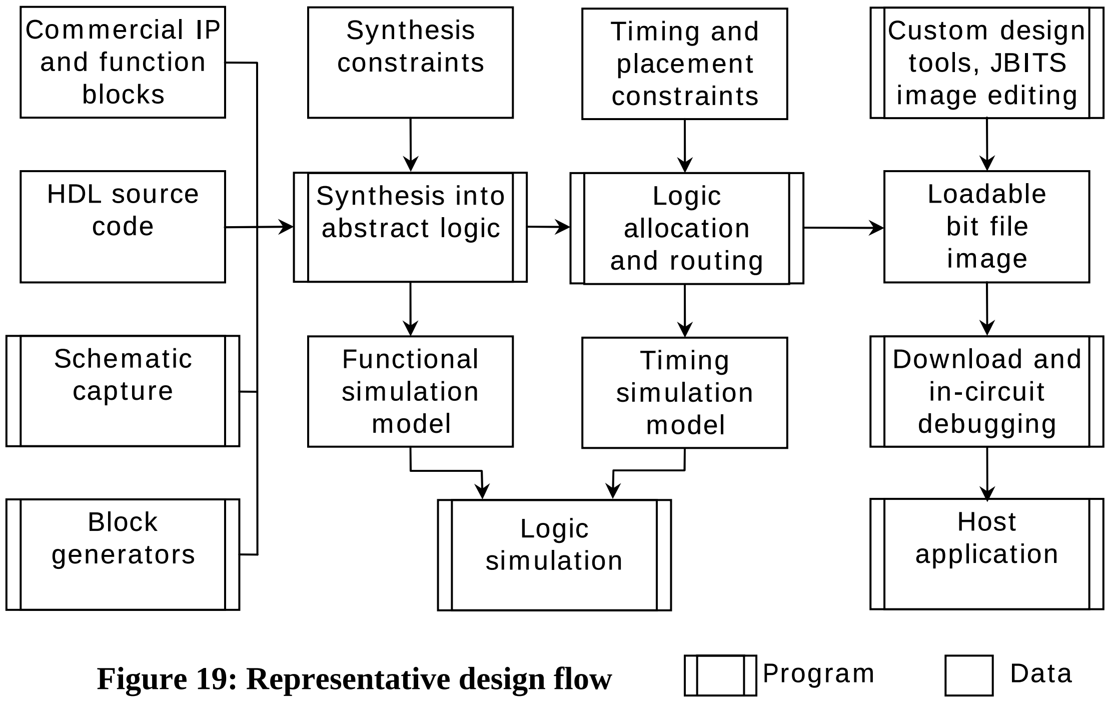

Commercial IP and function blocks
HDL source code
Schematic capture
Block generators
Synthesis constraints
Synthesis into abstract logic
Functional simulation model
Timing and placement constraints
Logic allocation and routing
Timing simulation model
Logic simulation
Custom design tools, JBITS image editing
Loadable bit file image
Download and in-circuit debugging
Host application
Program
Data

Figure 19: Representative design flow

Additional inputs to this step guide the process of logic synthesis, for example different ways of encoding finite state machines or fanout constraints. Output from this step commonly includes preliminary performance estimates, including timing and hardware utilization, and functional models, suitable for behavioral simulation.

Once the logic is phrased in terms of chip-specific primitives, design elements are allocated to actual hardware resources and wiring (routing) is created between them. This is the placement and routing (PAR) phase. Designers often provide additional constraints at this step. Floor planning, for example, provides broad guidelines for location of design elements within the FPGA. User constraints state that particular logic signals must be connected to particular pins on the FPGA package. Timing constraints require that particular wiring delays stay within specified limits. After placement, connectivity resources are allocated. PAR is typically an iterative process. Placements and wiring allocations are tried, tested against user constraints, and successively refined. It is possible for a user to specific exact placements and routing resources, but this is highly chip-specific and rare in practice.

Output from PAR includes accurate estimates of circuit performance, based on particulars of the FPGA technology and on the exact logic and routing resources used. The post-PAR model includes accurate timing information, so can be critical for finding subtle errors in timing relationships. The most important PAR output is the *bit file*, the exact data image that will be loaded into the FPGA. This includes all lookup tables, switch settings, and other state required for the FPGA to implement the design.

Xilinx offers a Java API for some of its FPGAs, JBits [Xil03], that manipulates the bit file directly. That can be useful for highly skilled developers who create custom design tools, or who want to make minor changes (such as RAM initialization or some arithmetic constants) to the finished design. The JBits API is highly specific to the chip for which the file is intended, and is not available for all chip types. As of this writing, it is not available for the Virtex-II Pro chip family, the intended target of the prototype system.

FPGA vendors offer tools for downloading the bit files into FPGAs, or the user may load the data into ROM or Flash memory. Once loaded, debug tools built into the logic design help the designer analyze and debug actual system performance in the running circuit. Debug tools, possibly including JTAG serial data access, would have been incorporated into the design at early stages, typically as IP blocks provided by the FPGA vendor. The analogy to debug compilation of C code is loose but meaningful.

Some times, the entire flow is one unified package provided by one vendor. More commonly, though, a user's flow will include schematic capture, IP or block generators, HDL processors, synthesis tools, and simulators from different vendors. As a result, commercial and research tools for logic design vary widely in their input and output formats, depending on the subset of design responsibilities addressed by a particular tool or tool set.

The flow is also open to many variations. Some features of Figure 19 have inner detail, not show – graphical or compiler-based design entry need not be the single unit suggested by the illustration, but a family of interacting tools. The HIDE tools, for example, generate specific placements of logic elements within an FPGA, but use the vendor's tools for routing [Bel03]. It is also common for tool vendors to define vendor-specific pragmas in their HDL compilers for various kinds of synthesis constraints. Figure 19 summarizes the kinds of inputs provided to typical design tools, but the specific sources of different kinds of data vary widely.

3.2.2 Verilog/VHDL

The two dominant HDLs are Verilog [IEEE01] and VHDL [IEEE97, IEEE02, IEEE02a]. Although popular within different design communities, these languages offer similar levels of abstraction. Practical application of these languages is generally at the RTL design level, no matter what the claims of each language's proponents.

These offer modest isolation from the idiosyncrasies of the underlying FPGA's fabric of logic primitives. For example, the design could specify addition of an eight-bit and a ten-bit value yielding a ten-bit result. The language and synthesis tools automatically sign-extend the one value, hide use of specialized hardware resources (e.g. carry-propagation logic), and allocate enough bit-level resources to create the multi-bit values, logic, and registers. These HDLs also support division of the application into multiple modules, allowing hierarchical decomposition of problem statements and designs.

These languages share two main strengths: some independence from the logic primitives offered by any particular implementation technology, and a high degree of control over the allocation of logic resources. Platform independence is often illusory, though. Real systems introduce non-portable pragmas [Alt03], maybe hundreds of them [Xil02], to instruct that specific HDL constructs be synthesized in particular ways. Vendor-specific support libraries [Act01, Xil02a] also expose the underlying primitives. This supports the traditional style of logic design, in which the best designer was the one who could use the idiosyncratic capabilities of the hardware to the fullest extent.

Because an HDL permits fine-grained control over logic resource allocation, it generally requires users to exercise that low level of control, too. High-performance applications commonly wish to replicate a computation as many times as will fit into the hardware resources available on a specific chip – no current system allows that kind of open-ended request for parallelism. Also, HDLs tend to be constrained to a subset of the optimizations used in standard programming languages [Xil02b]. They may be able to extract common sub-expressions, for example, or reduce logic through constraint propagation. More aggressive optimizations, such as loop unrolling, are generally missing from HDL compilers, though. This is reminiscent of the 1970s programming languages like K&R C [Ker78], which relied less on optimizing compilers and more on optimizing programmers.

System Verilog [Acc04] is emerging as a successor to Verilog, but has not (as of this writing) gained wide acceptance. It has many OO extensions that are said to be useful for simulation and modeling. The OO features appear not to lie within System Verilog's synthesizable subset, however. System Verilog includes standard Verilog as a subset, and its synthesizable subset is expected to be System Verilog's as well.

3.2.3 Non-OO design languages

Many HDLs have been proposed and implemented; a few have even been commercialized. Each one represents a somewhat different conceptual basis, different logic specification philosophy, and set of design tradeoffs. Non-object oriented languages predate OO systems, so they have a longer history. Also, partly because of relatively slow infiltration of software design concepts into hardware design practice, non-OO HDLs continue to hold a strong position on HDL research. Although they vary widely, recent HDLs seem to cluster around four major conceptual centers:

· **Customizations of existing languages**, typically C, for behavioral specification. In these HDLs, familiarity of the base language is generally considered an asset, as way of easing the involvement of standard programmers in hardware design. This also lets the language designers focus on their novel aspects, without having to solve basic language design issues all over again.

· **Geometric design**, confusingly called "'structural" design by some authors. In these systems, it is taken as a premise that the 2D arrangement of logic elements on a chip is as fundamental as the behavior of the logic elements, usually for performance reasons. Most geometric HDLs do not require absolute 2D addresses. They do, however, require design in terms such as "*A* to the right of *B*", "*C* north of *D*", and so on.

· **Structural design**, as distinct from geometric. In structural design, hierarchical composition of logic components is the central concept. The hierarchy reflects logical structure only, and does not have any direct effect on the placement of logic elements within the FPGA fabric. Behavioral or functional specification need not even be part of the language, but relegated to primitive components defined outside the system. Verilog and VHDL can be used this way, in terms of vendor-specific component libraries, but are more often used as a combination of behavioral and structural HDLs.

· **Interface-based design**, often an extreme form of structural design. Here, the externally visible features of a logic component are central, and automate connections within or across levels of hierarchy.

Of course, real systems generally emphasize one conceptual base while including others, in varying degrees.

Language customizations for behavioral specification

HardwareC [Ku90], SpecC [Dom98, Dom02], Bach [Kam01], Handel-C [Cel03], dbC [Gok97], Streams-C [Ags95], Transmogrifier C [Gal95], Mitrion-c, [Mit05], Impulse C [Imp05, Pel05], and SA-C [Rin01] represent a category of behavioral languages with syntax based more or less closely on ANSI C, but with extensions representing specific hardware concepts. In HardwareC, those concepts include block composition, fine-grained parallelism, and tri-state busses. In SA-C, a set of common image processing kernels (2D convolution, histograms, etc.) have been identified, then represented as a fusion of loop control constructs and array handling. Other mainstream programming languages been modified for use as HDLs, as well [Sni01].

One category of FPGA tools based on sequential languages "*is designed with embedded processors in mind*" [Imp05], or implements applications in terms of "*virtual processor*" [Mit05]. Perhaps this technique is meant to appeal to programmers with typical backgrounds, or to offer some benefit in compilation and scheduling of operations. At an extreme of this style, one finds the X-Acute Tcl [Acr05] and JOP Java [Sch05] bytecode processors implemented in FPGAs. Processors inside of FPGAs, especially soft cores, typically run at clock speeds one to two orders of magnitude lower than those of typical CPUs, so this technique has trouble exploiting much of an FPGA's inherent parallelism.

Celoxica's Handel-C is said to be "*based on the syntax of conventional C*" [Cel03]; it is very loosely based on C, and introduces a number of incompatible constructs. It adds language elements for parallelism, for replication of logic, for reset operations and for Occam-like communication channels between elements executing concurrently. It omits the `union` declaration, but adds declarations of memories and signals. It also adds novel syntax similar to bit fields in a C `struct`, allowing explicit bit allocation to particular variables. It allows pointers, but places severe constraints on the ways they are allowed to be used. It redefines `main()` to be a clock domain – an application with more than one `main()` entry point implicitly has more than one clock domain. As with Forge, Handel-C keeps large parts of the syntax of the base language (C in this case), but creates semantic differences that make it quite unlike its parent language.

SA-C [Boh01, Dra00, Ham99, Ham01, Rin01] differs from the Forge, Handel-C, and System-C in two significant respects. First, it is a research vehicle, not a commercial product. More importantly, it does not claim to address the general problem of compiling any application into any hardware engine. It addresses a specific set of computations that are known to occur in image processing. Its authors claim it to be a single-assignment variant of C. If a C "variant" is allowed utterly novel control flow, array indexing syntax, declaration types, and syntax for variable references, then that could be true.

SA-C embodies a number of concepts that appear useful in contexts other than image processing. Its single-assignment rule simplifies data flow analysis, but is almost obviated by other language features. It eliminates normal looping constructs in favor of a few custom constructs. These have been chosen for broad utility within the application domain. One loop type implicitly iterates over all indices of a two-dimensional array, as needed for typical whole-image operations. Other loop organizes 2-D window operations or 1D row and column operations within the 2D image. Yet another co-indexes several vectors, as needed for sums, dot products, etc. SA-C invites the application writer to fill in the loop bodies with application logic. By enforcing the use of well-understood loops, SA-C creates a highly predictable flow of data. That flow is amenable to many optimizations that can not feasibly (or at all) be derived from highly generalized language constructs.

Since SA-C prohibits ordinary hand-coded loops, loop bodies must be basic blocks (except for possible conditionals). This dramatically simplifies automated analysis of the code. The single-assignment rule, in particular, makes it possible to analyze all expressions fully – there is no possibility of open-ended growth of a sum, for example, because of open-ended iteration of a loop construct. It is worth noting that “shaders” in ATI’s Radeon, NVIDIA’s GeForce, and similar GPUs [ATI02] are programmed using a similar construct: the application writer provides vertex or pixel computation code, without control constructs or with profoundly limited ones, and that code is iterated over pixel ranges of interest by the GPU’s control logic. Limits on shader control constructs include “static branching” and “static looping,” where branch selection and number of iterations are decided at compile time [Eng04], allowing compilation to branchless instruction sequences. Some GPUs also imposed limits on the number of instructions in a shader, 96 for pixel shaders and 256 for vertex shaders in Microsoft’s DirectX 9, shader 2.0 [NVI05]. Other constraints, such as limits on specific kinds of instructions (e.g. texture vs. arithmetic), also simplified the system controller [NVI04]. The OpenGL shading language is built around the same separation of computation and control [Ros04]. Another analogy to SA-C is that GPUs control all access to application data by the application-specific code. In SA-C’s case, this allows advanced memory optimizations and data sharing between loop iterations, but GPUs use this to hide task scheduling and the presence of parallel evaluation pipelines.

Geometric design

For current discussion, a design is *geometric* if it is phrased in terms of the spatial placement of logic elements with respect to each other, whether or not an exact coordinate system is used for placing design elements. This has been called “structural” design by some authors, but current discussion reserves a different meaning for that term.

It has been asserted that *“[geometric] design techniques often still result in circuits that are substantially smaller and faster than those developed using only behavioral synthesis”* [Hut99]. It is also widely realized that the place-and-route (PAR) step of FPGA design can be time consuming – overnight runtimes are common for large, dense designs. Geometric design often constrains the placement problem and eases the routing problem enough that PAR is omitted or minimal, a fact that appeals to people impatient with hours-long PAR times.

Many languages and systems make geometric features first-class parts of the design statement, including μFP [Luk89], Pebble [Luk98], JHDL [Hut99], and HIDE [Ben02, Bel03]. Languages including Ruby[1] [Jon90, Guo95, Sin95] express geometric constraints in less explicit terms. Lava [Bje99] assigns additional geometric semantics to features that already existed in the base language from which it is derived.

Geometric information is not present in standard HDLs. It can, however, be added in the form of floor-planning directives to the place and route steps of standard tool flows. Other design constraints can force logic assignment to exact resources within the FPGA, define relative placement between design elements, or create connections to specific IO pins [Xil02]. These constraints can be inserted into VHDL using custom values for the language's standard `attribute` mechanism, or into Verilog using specially formatted code comments. Extensions like these do not qualify either VHDL or Verilog as truly geometric design tools, but as structural or behavioral tools with non-standard geometric semantics added.

---

[1] Not to be confused with a more recent scripting language by the same name [Tho04]

Structural design

A purely *structural* design specifies logic elements as blocks composed of other blocks and connections between them, down to black-box primitives. Mainstream HDLs including VHDL and Verilog combine behavioral and structural design styles, but can be used in a completely structural way, when combined with vendor-specific libraries of primitives [Xil02c]. Systems such as JHDL [Hut99] and Balboa [Dou03] support only connection of design blocks, assuming that leaf blocks will be created externally to the design system. Vendors like Xilinx provide extensive libraries "cores" as simple as adders or as complex as network MAC protocol handlers. Other research has proposed systems for block generation [Che90, Meg01], without direct reference to connection between blocks, or to minor customizations of complex algorithms [Chu98]. Block-generators such as PAM-Blox have been useful for packaging specialized designer knowledge about CORDIC algorithms [Men00], boolean satisfiability [Men99], or floating point [Lia99]. Such systems often address varying kinds of configurability [Giv00].

Interface-based design

Throughout this discussion, the term interface is intended, at least loosely, in the following sense [OMG03 p.3-50]: "*An interface is a specifier for the externally-visible operations of a class, component, or other … without specification of internal structure. … Interfaces do not have implementation*". Although this definition is most commonly applied to software systems, it is compatible with the `component` definition in the VHDL specification [IEEE02 sec4.5]: "*A component declaration declares an interface to a virtual design entity that many be used in a component instantiation statement*" – implicitly, without knowledge of the component implementation. It is also compatible with the Java language's `interface` construct.

When design blocks are created or bought as black-box units, only the block's interface is visible to the designer. This has prompted interface definition as a task in itself [Neb96, VSI97, Row97, Vah98, Len00, Dou03, OCP01, Oli02, Sin02]. As will be seen in later sections, well structured interfaces are central to both object oriented design and division of responsibility within the design system.

3.2.4 Synthesis from standard programming languages

A number of different approaches have been taken to increasing the conceptual level at which an HDL represents a design by basing the HDL directly on a familiar programming language. Xilinx, for example, has created the Forge programming language. It is syntactically derived from Java, but with a number of constraints described in the language's *Source Style Guide* [Xil03a]. This *Style Guide* states the restrictions that make Forge a subset of Java. The restrictions, rather than being points of programming "style," create such profound constraints that normal, object-oriented Java programming style is largely impossible.

System-C [OSC03] is another derivative language, based on C++. Since it supports templates, operator overloads, and user defined type-casts, it can implement complex semantics with syntax that looks superficially straightforward. Like Handel-C, it supports interfaces (a bit of syntax that might descend from Java.) Although Synopsis advertises support for System-C, it appears not to have gained broad acceptance within the logic design community. This language appears to rely on a sophisticated front end, possibly more complex than the input phases of standard C++. That complexity makes it unattractive as a research vehicle.

Other systems use standard programming languages as a means of exploring the space of hardware/software partitions. They vary widely in the degree of automation they provide. ASC [Men03, Men06] starts with a C++ compiler, but adds libraries of macros and hardware-oriented data types. Under the control of macros, operator overloads, and coding features, the programmer specifies a subset of the application to be synthesized. Approaches differ, but ASC uses a well-established library of logic blocks [Men02] for generating synthesizable output. HLLs with parallel constructs have also been used or adapted, including Ada [Bar85], parallelized Prolog [Gre85], Occam [Man85, Pee00, Pag95], and Haskell derivatives [Odo02].

To raise the level of abstraction of HDLs, several projects have integrated useful compiler functions such as generating graph representations and performing optimizations [Gal95, Sod98, Gok00, Mar00, Kam01, Moi01, OSC03, Xil03a]. Others compile efficiently to particular hardware models [Hau97, Wai97 Gol99, Ye00].

However, direct compilation of standard C code to hardware, even when possible, does not overcome the basic problem – that *"concurrency is fundamental for efficient hardware, yet C-like languages impose sequential semantics and nearly demand the use of sequential algorithms"*. Further, *"automatically exposing concurrency in sequential programs is limited in its effectiveness*" [Edw04], and *"compilers are generally able neither to detect enough of the necessary parallelism nor to reduce sufficiently the inter-processor overheads"* [Str05]. This is due to unintended introduction of unnecessary dependencies, which are very difficult to remove [Sny86, Cul99]. Even without unnecessary dependencies, *"two designs synthesized from two semantically equivalent but syntactically different descriptions may differ significantly in quality,"* with optimal designs potentially being unreachable [Voe01]. As a result, it is both correct and misleading to assert that "*system design is essentially a programming activity*" [Gup97]. Reconfigurable system design is programming, to the extent that it uses keyboards and compilers as its main development tools. This statement disregards the fact that FPGAs represent a fundamentally different kind of computing fabric than any fixed processor, a different *type architecture*, [Sny86]. As a result, that statement gives the impression that standard, sequential programming languages can represent massively parallel FPGA-based implementations as effectively as they represent sequential programs – a questionable assertion, at best.

It has long been known that superscalar ILP (Instruction-Level Parallelism) provides only modest speedups on typical applications [Ris72, Jou89, Wal93]. Some mathematical applications are amenable to loop unrolling and other aggressive optimizations, but non-vector code derives little benefit from *N*-way parallel instruction dispatch, for *N* greater than about 4 to 10. Given FPGAs' relatively slow clock rate, massive parallelism must be a major factor in achieving high speedups over processor-based implementations. These results provide little hope that automated analysis of existing applications can extract the level of parallelism required to make FPGA acceleration attractive. This leads to the statement that "*… the C-to-gates approach … doesn't work because of the parallel nature of hardware. This shouldn't be too much of a surprise, because optimizing a sequential program onto a parallel machine has long been known to be computationally intractable …We might hope there would be compiler algorithms to do a "good enough" job despite the complexity. … Unfortunately, and despite 40 years of parallelizing compilers for all sorts of machines, these algorithms don't work terribly well.*" [Pag04]

3.2.5 IP libraries

There is burgeoning industry built around intellectual property (IP) blocks for use in logic designs. Xilinx, for example, offers a large library of "cores," designed by Xilinx or by other vendors. Some are as simple as parameterized accumulators, or as complex as JPEG decoders and whole microprocessors. Actel offers over 100 cores [Act03] covering a similar range capabilities. Altera has a similar library, and offers a "DSP Builder" product [Alt03]. This offers several dozen computation-, memory-, and filter-oriented blocks, parameterized in a variety of ways, with semi-automatic aggregation into dedicated data paths. Licensing arrangements vary widely, according to the vendor and complexity of each block.

These cores have a number of obvious advantages: they offer complex functions (including Reed-Solomon codecs, FFTs, and PHY-level network protocols), already debugged, and ready to add to the standard logic design flow. It would be difficult to use many of these functions in BCB applications, though. First, the functions offered have little to do with BCB or other general computations. Even blocks that look promising at first, such as Viterbi decoders, may have very different meanings in applications like hidden Markov models [Kos01] than they do in the context of error-correcting convolutional codes [Swe02].

3.2.6 Mixed-language approaches

Logic designs are commonly defined using more than one kind of representation. The purely structural design style of VHDL or Verilog is one example, where the leaf IP blocks, cores, and primitive components [Xil02c, Act03] are defined outside the language. Several vendors allow DSP functions to be modeled in Matlab, then converted into logic [Xil02d, Alt03]. Component generators [Luk98, Men99] and domain-specific compilers [Men01] represent other ways of using different specification languages for creating parts of a logic design. The question is rarely whether multiple representations should be used in a logic design, but the way in which the different representations are integrated.

The conceptually simplest way to combine logic representations in one logic design is to define components separately, using the different representations. Components are then integrated at a late stage in synthesis or PAR. This is analogous to the software modules written in different HLLs (e.g. Fortran and C), converted to a object code format, and integrated in their converted forms by a linker. In the normal Xilinx tool flow, EDIF is the common low-level representation, output by compilers, graphical schematic capture tools, block generators, etc. This is the most flexible way of combining different representations, since it eliminates any dependencies between the different formats.

A related approach uses different representations at different levels of hierarchy. Balboa [Dou03] is built around the idea of integrating leaf components from undefined, external sources. This uses at least three different representations: one or more formats for creating the leaf blocks, C++ as a Split Level Interface (SLI) adapter from external format to a common representation, and a Component Integration Language (CIL) for composition of leaf blocks into complete designs. The CORBA Interface Definition Language (IDL) [OMG01] is another integration language. IDL is similar to Balboa's CIL in providing a unified mechanism for establishing communication between components of differing interfaces. IDL differs in that it creates adapters to enable communication but does not create communication pathways.

IDL also differs from CIL in being an OO language, of an odd sort. It supports classes, polymorphism, and inheritance, but does not itself support objects. Instead, it uses OO concepts to describe objects created using standard programming languages, like Java, Ada, or C++.

Other language combinations used mixed representations within a single body of source code. The C language's macro preprocessor, cpp [Ker78], is one familiar example. Although macros are commonly treated as part of the C language itself, the macro preprocessor typically runs as a separate compilation phase. Macros in C code always contribute to the C application, and are eliminated from the source code at an early stage of compilation.

Javadoc annotation [Fri95], including an HTML subset, is embedded in Java source code as comments, so has no effect on the Java application. Instead, Javadoc annotation is processed separately into hypertext that describes a Java implementation. This is related to Knuth's Literate Programming [Knu92], in which alternative processing would extract either Pascal code or $\TeX$-formatted text from a file containing both. This differs from Javadoc a number of ways, but most obviously in that a literate Pascal program is not a valid Pascal program itself, but must be "tangled" before compilation.

Other language combinations include HTML with embedded JavaScript code [ECMA99], PHP programs with embedded HTML [Gil01], SQL embedded as string data in other programming languages, tool constraints embedded in VHDL attribute declarations [Xil02], and so on. A full catalog of language combinations and relationships between combined languages is far beyond the needs of current discussion. It suffices to say that multiple different languages can successfully be combined in a single source file, allowing each to take responsibility for a different aspect of the whole system's definition. The possible combinations and ways of extracting the different elements are limited only by an implementer's creativity.

3.2.7 Visual design entry

Design entry systems are the point at which a programmer or logic designer states the intended behavior of an FPGA-based system. Two styles of interface dominate contemporary commercial practice: register transfer level (RTL) programming languages, described above, and schematic capture [Xil02a].

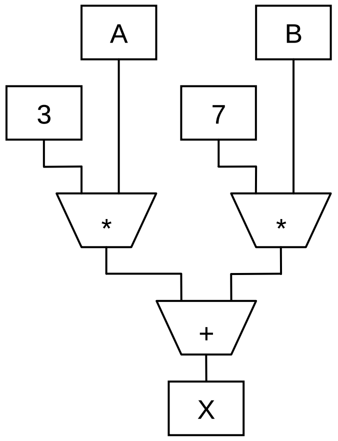


Figure 20. Graphical form of expression X = 3*A + 7*B

Schematic capture uses a visual editor, with the familiar glyphs for AND gates, OR gates, inverters, and other design elements. The design consists of various instances of the selected primitive objects and blocks representing other design components in hierarchical aggregation. Connectivity is strictly visual, consisting of lines representing wires connecting the circuit elements. Although natural to some hardware designers, particularly those experienced at the transistor and discrete component level, schematic capture is not suitable for representing BCB applications. The problem is not that visual representation is necessarily inappropriate to BCB, it is that the design primitives available are poor matches to BCB applications. More than just the logic elements offered, the techniques for creating design hierarchy, control constructs, and logical relationships are not suited to high-level software design.

In addition, the need to create a legible spatial arrangement adds to the number of design criteria and constraints, even though that spatial arrangement is irrelevant to the computation. Figure 20, for example, illustrates a typical representation of the phrase `X=3*A-7*B`, nine typed characters. The visual representation requires manipulation and placement of eight blocks representing signals, constants, and operations, plus seven connections between them. It also includes the task of placing blocks and wires in visually logical arrangements. This visual placement is generally not geometric design, in the sense of section 3.2.3, since it does not affect the placement of logic elements within the FPGA fabric. That arrangement is work required of the user by the design tool, but that is irrelevant to the logic design.

Star Bridge's Viva system [Sta05] can operate at a similar or even lower level than typical schematic capture, exposing the structure of the Xilinx FPGA's Configurable Logic Block (CLB), the finest unit of configurability in the FPGA. Applications built from such device-dependent libraries are difficult or impossible to port, even to other products from the same vendor. The Viva tool set does, however, contain higher-level blocks, such as adders and multipliers, parameterized for number of bits. At higher functional levels, its feature set is comparable to that of other box-and-wire visual design tools. Annapolis Micro Systems has also announced its CoreFire design system, based on a library said to contain over 1000 block types. This library is heavily oriented towards signal processing functions, and its usefulness in general computation is not clear. It also appears to be locked to Annapolis hardware, further limiting its usefulness.

Xilinx has created a set of libraries, the System Generator for DSP, that work with MatLab's SimuLink [Xil02d, Mat03]. These allow a user to define signal processing data paths using point-and-click, drag-and-drop interface. The basic units of the design library include elements to generate constant values, multiply by a constant, add two values, or compare two values. There are more complex blocks as well, up to the complexity of FFTs or microprocessor cores. This is different from schematic capture systems, since the blocks have no direct correspondence to logic functions, and since synchronization and buffering are handled automatically. Other tools, such as Accelchip [Acc03] and Match [Per99], convert Matlab applications to FPGA-compatible logic.

Other visual tools exist, including floor planning tools and circuit-level design tools. These in fact have a close relationship between an element's position in the visual field and it position in the FPGA fabric. Floor planning has little or no effect on the logical function of a system. Given 100K available gates, design at the level of individual gates can not be effective except in rare, isolated, and small parts of a design. As a result, neither kind of tool is considered important for the functional design of large computations.

UML for visual representations

OMG [Sel98, OMG03, OMG03a], ObjecTime Ltd. [Sel94], and many others have proposed visual representations for software designs. These systems tend to share several characteristics. First, they address the general problem of representing all possible software systems, or vast categories of systems. Although sometimes applicable to hardware design, they tend to address hardware concepts tangentially or in unfamiliar terms. Second, they tend to be complicated, using a variety of notations to represent different aspects of system behavior. Third, they tend to lack clear relationships between the graphical and textual encodings of different parts of the system specification. In particular, the division of responsibility between visual description and source code representation tends to depend on specific CASE tools rather than the notations themselves. Finally, these representations are often vehicles for specific design methodologies [Sel94, IBM04]. There is a growing realization that these methodologies address projects involving thousands of work-years of development effort, but do not always suit smaller designs [Bec00].

Still, UML has gained wide acceptance as software design and even implementation tool, so it is no surprise to see UML applied to logic design, too. It includes a variety of notations for representing state machines, object composition, object interactions, and other useful concepts. The YAML[2] system [Sin00] uses UML's "component diagram" notation as the input format for synthesizing control structures. In order to strengthen the design entry metaphor, YAML also uses some geometric design. Others have used UML's activity diagrams for specifying state machine behavior [Bjö02], generating synthesizable code using tools reminiscent of the Model Driven Architecture's transformations between abstraction levels [Kle03]. Another effort uses UML class diagrams to automate creation HDL code skeletons, which are then filled in by hand [Dam04]. This work also parameterizes component definitions (called *metaprogramming* by those authors), using UML's template class notation. The authors note that UML's containment and physical composition map neatly into structural hardware design, but use the term "logical composition" to map multiple inheritance to a component that exports more than one interface. Although the conceptual mapping is meaningful, the

[2] Distinct from *YAML Ain't Markup Language*, proposed as a simpler alternative to XML. (http://yaml.org/spec)

authors have created a new definition for the term “composition” in a way likely to cause confusion to people familiar with UML’s composition in other contexts.

Newer features in the UML family are also appearing in the hardware design literature. MDA (Model-Driven Architecture) is based on automated conversion of PIMs (Platform-Independent Models) to PSMs (Platform-Specific Models), or automated adaptation of high-level system specifications to technology-dependent implementations [Kle03]. UML’s OCL (Object Constraint Language) [War03] is designed to add semantics constraints a system description, so serves some of the same purposes as the `assertion` statement of SystemVerilog [Acc03].

3.2.8 FPGA reconfiguration tools

FPGA reconfiguration, and especially partial reconfiguration, is not properly a kind of design tool, although tools have been built to support it. Instead, partial reconfiguration is a technique for reloading part of an FPGA’s programming pattern while leaving other parts untouched. Some FPGAs and accelerator implementations allow reconfiguration of part of the FPGA while other parts continue to deliver service. Since partial reconfiguration has the potential for modifying small functional elements within a larger body of logic, it is worth considering as one possible way of creating customized accelerators.

Standard FGPA development combines IP blocks and circuit-generator outputs with custom logic designs, possibly based on graphical circuit capture or text-based languages like VDHL and Verilog. This design technique generates a single bit file representing the entire definition of the FPGA's programmable state. A manufacturer can release new bit files to reconfigure an FPGA-based system, to correct errors or to enhance features – *compile-time configuration*, as some authors call it. Some systems choose between multiple bit files at run time, loading new ones as needed to switch between operating modes. This is called *dynamic* or *run-time* reconfiguration, sometimes called *temporal partitioning* of logic [Hut95]. It allows the use of smaller FPGAs, since only the active subset of application logic needs to be held on chip at any moment, reducing logic costs and power consumption. Said differently, it can "*increase the functional density*" of a system [Eld96].

Developers have also realized that bit files could be customized on the fly, to make application- and context-specific changes to functionality or data values. This added flexibility allows hard-coded efficiency, for example in string-matching applications [Lem95] or in filters with variable coefficients [Wir97]. When Xilinx published its JBits API [Guc99a] for the XC4000, descended from the JERC6K tools for the XC6200, it allowed full access to the bit file. This enabled new uses for application-specific customization of bit files [Guc02]. Other, similar APIs also enabled application composition on the fly, from libraries of fixed components, resolving placement mismatches between input and output ports at composition time [Guc99]. Still these approaches all loaded or re-loaded the entire FPGA programming pattern.

As early as 1988 [Kea88] it was realized that subsets of the FPGA's state could be reprogrammed without changes to the rest of the FPGA logic definition – *partial reconfiguration*, occasionally called *local run-time* configuration. This offers several advantages over complete reconfiguration, including faster time to reconfigure, preservation of state in untouched parts of the FPGA, and the possibility of continued operation during reconfiguration [Cur05]. The feature was first commercially implemented in Xilinx's XC6200, and has since been used in many FPGAs from Xilinx and other manufacturers.

Current tools for partial reconfiguration address different parts of the problem. PARBIT [Hor01] takes a full bit file and extracts the region to be loaded. The swappable component must be built using synthesis constraints to define the region of chip in which it is to be loaded, and the chip coordinates at which connecting wires must be located [Hor04]. JRTR [McM00] analyzes two different bit files in order to determine which FPGA regions differ and must be reloaded.

Design techniques for partial reconfiguration also address different aspects of the problem. Early papers covered basic concepts, such as kinds of reconfigurations most amenable to partial reconfigurations [Had95]. Other authors address issues of two-dimensional fit between logic elements that change at different times, whether rectangular [Baz00] or not [Com02]. Coverage of partial reconfiguration issues is erratic, though. It is still true that "*Currently available CAD tools are a very poor match for local RTR [run-time replacement] implementations*." [Hut95]

3.2.9 Object Oriented Hardware Design Languages

As noted above, Java and C++ are two mainstream OO programming languages that have been adapted for hardware specification. Despite numerous proposals for other OO HDLs over the years, they not had wide acceptance. The IEEE 1577 standards committee was formed to study OO extensions to VHDL, but was terminated without producing a standard.

Lola [Wir95] was a relatively early and stripped-down HDL. Like Pascal (and by the same creator), it was intended as an educational language. Although Lola uses the idea of objects for representing hardware elements, it lacks other features generally associated with OO languages, notably classes and inheritance.

SystemVerilog [Acc04] is an object oriented extension to standard Verilog. At this writing, it has some industry support, and is in draft form as IEEE standard 1800. Acceptance of the final IEEE standard is not complete, however. A number of companies have announced support for SystemVerilog, most often as a design verification language. SystemVerilog's extensions beyond the original Verilog language OO include classes, class objects, and inheritance. According to the SystemVerilog standards committee, however, a "*class is not intended to be synthesizable but useful for testbench and system level modeling*." [Ger03] If the OO language features are not synthesizable, current discussion does not consider this an OO HDL.

It has been claimed that VHDL is an object oriented design language [Eck96], because of the static form of polymorphism supported by the `configuration`/`architecture` language construct. The claim that VHDL, in its current form, supports inheritance requires a notion of inheritance that very few would accept.

At least one design system [Dam04] uses UML notation to create OO descriptions of logic designs. The system described, however, uses OO techniques only for the design hierarchy and for defining the ways in which components are parameterized. Leaf-level implementation of the design is left to unspecified "generators." Also, the examples and discussion do not demonstrate significant use of UML's OO subset – using an OO-capable design tool without using its OO features is difficult to consider OO.

3.3 Object Orientation and HDLs

Various authors have described at least three meanings for an "OO design system:"

· The design system itself is implemented using OO technology [Gup89, Men99, Men01], irrespective of how the resulting hardware is modeled or specified.

· The simulatable model of the hardware system is phrased in OO terms [Hut01, OSC03, Acc04], irrespective of how the logic is implemented or how the hardware design is specified.

· The synthesizable hardware design is specified in OO terms [Wir95, Bje99, Hut99, Xil03a], at least in part.

The first definition is not of interest for current purposes. It refers to the implementation technology used by the tool-builders, and says nothing about how the hardware's structure and function are defined.

The second definition, the hardware model, is not necessarily of interest either. The hardware model is the design specification only if it can be translated directly and automatically into a logic design. Not all modeling systems have this property. Such systems require at least two representations of the hardware design, one for modeling and simulation, and a second for synthesis. This includes oddities, such as versions of SystemC that use C++ with inheritance for modeling, but exclude inheritance from the synthesizable subset [Gim03]. It also includes methodologies that separate OO architectural design and functional representation for logic specification [Jan00].

OO systems in the third category actually use OO techniques to specify all or part of a hardware design. Lola, an early object-based HDL [Wir95], used objects to represent components, and object state to represent component state. One Java-based design system mapped Java's component model, called JavaBeans, to hardware components [Kuh00], and otherwise maintained the semantics of large parts of the Java language. JHDL, another Java-based effort [Hut99], used Java syntax for connecting leaf components, defined outside of the system. It is worth noting that, although JHDL allows all of Java's semantics for compile-time configuration, its synthesizable mechanism for passing data (the `Wire` class) supports only untyped bitstrings. Other research have described fixed-depth [Ver00] and more general [Neb96] inheritance hierarchies for component reuse. Despite lengthy discussion in the 1990s, Objective VHDL [Rad98] and other OO extensions to VHDL (e.g. [Sch95]) never entered the mainstream, and IEEE efforts at standardization have been discontinued.

3.3.1 OO language features

Definitions and categories of object oriented (OO) programming languages have varied over the course of time [Weg87, Liu00], and even now differ between authors. Whatever the other differences in their definitions, writers seem to agree on these features as central to object orientation [Cab95]:

· Use of objects, discrete entities that define some set of operations and, optionally, manage some internal state. Different instances of an object type have identical operations, but each has its own replica of at least some of the state values,

· Inheritance, a set of policies for defining new objects in terms of one or more existing objects or interfaces, and

· Polymorphism, rules that allow objects with a given interface to be handled the same way, irrespective of the object's type and of how the interface is implemented.

· Classification, or presence of language constructs (commonly called *classes*) that create a distinction between an object interface and object implementation.

This consensus admits large numbers of variations. Classes (including Java interfaces [Gos05], etc) define object types in terms of methods, data values, etc, but do not define instances of the type, and are not a necessary part of an object-oriented type system [Bru02]. Object oriented languages without classes are often called object-based. JavaScript [ECMA99] and Self [Ung97] are object based languages that support inheritance. Some early object based languages, including Modula [Wir82] and the educational HDL Lola [Wir95] had only objects but not inheritance. Since the distinction between *object oriented* and *object based* is not important for current discussion, the distinction will not be mentioned again.

Genericity is also known as *class parameterization*, *parametric polymorphism* [Bru02] or *let-polymorphism* [Pie02]. Despite historical debate [Mey86], genericity and inheritance-based polymorphism are generally considered complementary rather than competitive, and genericity is not a requirement for object orientation. Genericity mechanisms appear in C++ templates [Str97] and in the somewhat richer mechanisms in CLU [Lis84], Eiffel [Mey92], and Java 5.0 [Gos05], for building complex types in terms of other types. Ada [Ada95] defines the `generic` language construct, which allows datavalues well as types for class parameters. VHDL [IEEE02] borrows Ada's `generic` implementation, but allows only data values as parameters, not types, and the same is true of SystemVerilog's `module`, `interface`, and `class` parameterization [Acc04]. As a result, the VHDL `generic` and SystemVerilog `parameter` constructs affect only range limits on values and array indices. For purposes of current discussion, this is not considered a significant change to the parameterized class; only types passed as generic parameters qualify for genericity.

These language features can appear together or separately, in different combinations, as suggested by Table 1, adapted from [Liu00]. HDLs in this table are considered to support objects, classes, or inheritance only if those language features appear in the language standard's synthesizable subset. As a result, Table 1 reports both SystemC [Gim03] and SystemVerilog [Ger03] as lacking inheritance. VHDL and Verilog both have published standards stating their synthesizable subsets. Some vendors include more features in the vendor-specific synthesizable language subset, but the published standard is used for this categorization.

Examples of objects include instances of a Java `class`, a VHDL `architecture`, a Verilog or Modula `module`, or a CLU `cluster`. Java's `class` and `interface` constructs count as classes, because they both specify object interfaces, as does VHDL's `component`.

Verilog's `module` and SystemVerilog's new `interface` constructs can be instantiated to create hardware objects. In both cases, the interface definition is implicit in the

Table 1: Comparison of Object Oriented Features for Selected Languages

Language	*Objects*	*Classes*	*Inheritance*	*Type generics*
C, Fortran, Pascal	-	-	-	-
Modula, Lola[a], Verilog[a], SystemVerilog[a]	*Yes*	-	-	-
VHDL[a]	*Yes*	*Yes*	-	-
Ada83, CLU	*Yes*	*Yes*	-	*Yes*
JavaScript, Self	*Yes*	-	*Yes*	-
Java 1.4	*Yes*	*Yes*	*Yes*	-
CORBA IDL	-	*Yes*	*Yes*	-
SystemC[a]	*Yes*	*Yes*	*([b])*	*Yes*
Java 1.5, Beta, C++, Eiffel	*Yes*	*Yes*	*Yes*	*Yes*

([a]) This language is an HDL. All others are HLLs.

([b]) Yes, if the construct is in the compiler's synthesizable language subset, no otherwise.

implementation. Although these languages have hardware objects, they do not have classes. In VHDL, however, the `component` (object interface) and `architecture` (implementation) are different. Using the `configuration` construct, one `component` reference can be made to use different `architecture` implementations in different contexts. It's a crude but valid form of polymorphism.

Code references allow another kind of code sharing, and even a limited form of interface definition, but are not, but themselves, polymorphism features. They have the form of function pointers [Ker78], procedure parameters [Jen91], C# delegates [ECMA05], or other similar constructs. These refer to individual functions, which allow creation of different implementations of a given function signature, but do not refer to whole objects. Another system, Star Bridge's Viva, claims to be object oriented because it implements operator overloading and polymorphism [Sta05]. Operator overloading, though convenient, has little to do with the definition of object orientation: Java, a strongly OO language, lacks operator overloading; VHDL, a non-OO language, implements it. One could also argue that ANSI C's `void *` references and Java 1.4's collections of `Object` elements implement a form of polymorphism, since they are compatible with arbitrary reference types. Viva's "polymorphism" is somewhat more expressive than `void *` or `Object`, but not enough to qualify the system as OO, despite the vendor's claims.

OMG's CORBA Interface Definition Language (IDL) has the distinctive feature of being an "object oriented" language without having objects of its own [OMG01]. IDL is not intended as an application development language by itself. Instead, IDL is meant to enable component assembly in systems built from heterogeneous processors, networks, operating systems, and languages. IDL defines object interfaces, and defines methods that pass object references, but does not create object instances itself. In this, it closely parallels the intent of hardware design tools that focus on component assembly with relatively little regard to component content [Dou03].

3.3.2 Type and object resolution: static vs. dynamic

Polymorphism is one of the hallmarks of OO design. It is the property that objects of different underlying implementations export a common interface, and can be accessed interchangeably through that interface. In languages like C++ and Java, this implies that a given instance of some variable can refer to different objects at different times during execution. Those objects may be of different types, and therefore have different code implementing operations on those objects.

Software systems implement new object instances by allocating storage for the object's data, including references to any data shared across classes and to the code that implements the object's behavior. It is generally assumed that all instances of a single object type all share one body of code, but have unique data. This differs significantly from component instantiation in hardware design systems. Component instances do allocate storage, in the form of registers or on-chip RAM, for the component data unique to that instance. Hardware component instances normally create new instances of the component's logic, as well. That requires allocation of gate-equivalents, connections, and possibly other kinds of FPGA resources. This is roughly the behavior one might expect if the C++ `inline` declaration applied to each instance of a C++ class, instead of applying to each reference to the function. An `inline` function has the possibility of generating different code at each reference, due to local optimizations. Likewise, different instances of a component's logic can generate somewhat different circuitry, according to the circuit environment in which it is instantiated.

Typical software systems also pass references to application data (other than primitive types), but hardware systems pass the data values themselves. The choice of data instances on which a hardware component operates typically involves multiplexers and demultiplexers for collection of operands and distribution of results. RAM indexing can be used for both, but runs into problems when concurrent access is needed to many different objects, since multiported RAMs or careful assignment of objects to different RAMs becomes necessary.

At least one author has proposed a hardware mechanism that acts as if different objects (possibly of different types) are selected at hardware run time [Sin02]. Others have attempted to determine a fixed list of data items that a C pointer can refer to, and emulate that selection using hardware multiplexers [Sém01]. It has been much more common, however, for HDLs to require that types and object references be completely resolvable at compile time [Neb96, IEEE02, Xil03a].

Although VHDL supports some notion of classes and objects, in the form of `component` usages that each instantiate an `architecture` of some `entity`, it lacks object references. Each instance is statically bound to its inputs and outputs at the point where it is instantiated. Instance names can only accessed only by `configuration` statements, which can not affect the association of signals to instance inputs and outputs, and those instance symbols can never refer to another instance. Verilog does support object references, in the form of hierarchical names based on named `module` instances, but does not allow one reference to be assigned to another. There is a permanent correspondence of references to instances; names are considered constants that can not be reassigned. Also, Verilog does not support polymorphism at all, and VHDL resolves the implementation of a `component` instance under the manual control of `configuration` statements and at compile time. Neither language requires run-time resolution of object references or of object types.

C++ and Java support variable object references; popular HDLs (Verilog and VHDL) have either constant object references or none at all. The C++ and Java language type systems enforce elaborate rules ensuring that variables (including the implicit references in parameter passing, return values, and thrown exceptions) are never assigned object references of incompatible type. It is therefore worthwhile to reconsider the type system of any new HDL with object instantiation like that in VHDL or Verilog. Many of the C++ or Java type system's rules are predicated on ensuring type safety of variables that reference different object instances of different types. Those rules could be overly restrictive for any new HDL that lacks variable object references.

3.3.3 Object instantiation and hardware design

Object oriented programming languages generally share one major assumption: that an object instance is a new allocation of data storage, and that one instance of executable logic is shared across all objects. This certainly makes sense in the strict model of a sequential processor, where one instruction at a time executes, and the code is reused sequentially across references to potentially many objects. It also makes sense in a system with multiple hardware threads, cores, or processors and with shared memory – all execution contexts, at least in principle, have access to the same instruction store. Copies of code in one or more cache memories don't invalidate the basic model, because caching generally maintains the illusion of a single memory. The model even works, given a loose interpretation, in clusters and network-based multi-processors. Java's remote method invocation (RMI) extends the image of a single thread of execution to multiple homogeneous virtual machines, and OMG's CORBA includes features for maintaining the appearance of homogeneous processing on heterogeneous multiprocessors.

Logic design, including FPGA design, achieves parallelism through multiple instances of the executable logic. Many times, the data (registers) and code (logic gate equivalents) will be instantiated together. Other times, one instance of executable logic sequentially handles multiple data instances. No fixed relationship exists between code instances and data instances. The main stream of OO language theory [e. g. Aba96, Bru02, Pie02] appears unaware of this basic fact of logic design and implementation.

4 FPGA ACCELERATOR DESIGN TOOLS: REQUIREMENTS AND IMPLEMENTATION

Chapter 2 summarizes a set of case studies that expose many issues in FPGA-based accelerator design. The lessons of these studies suggest a number of requirements for tools intended for designing FPGA-based application accelerators. Chapter 3 exposes weaknesses in existing design tools and techniques. These experiences and analyses combine to suggest requirements for a new generation of design tools, presented in section 4.1. Implementation choices follow from the specifications, so section 4.2 presents the major architectural decisions made for creating new tools to implement the requirements of section 4.1.

The LAMP tools implement the choices of section 4.2 in order to meet the requirements of section 4.1. Chapter 5 describes the techniques used in selected parts of the LAMP compiler. Many different input representations could have been used to implement these semantic features.

Since the LAMP language syntax follows from and supports the more important design stages, it is presented in Appendix A. Next, Appendix B describes theoretical aspects of how computing arrays can be sized to give maximum parallelism for any specific application and FPGA platform. Finally, Appendix C presents the important deatures of a case study in using the LAMP tools.

4.1 Requirements

Case studies of applications, described in Chapter 2, suggest a number of basic features that meet two basic demands: exploitation of the FPGA's performance potential, and accessibility of that potential by the application specialists with the computing needs.

- **Families of applications.** Applications tend to appear in families of related computations more often than as isolated point solutions. Application families differ from each other significantly in their basic structure, however. Design tools must be able to define families of applications where families differ widely in their basic structure, but members of any one family differ only in the details embedded in the family's common structure.

- **Multiple participants: logic designer and application specialist.** At least two different skill sets, not generally found in any one person, are needed for creating successful application accelerators: a logic designer and an application specialist. All participants must be able to work with a reasonably familiar notation for their part of the system, but not necessarily the same notation. Dependencies between the participants must be reduced so that routine changes to the accelerator can be made by the application specialist alone, without direct support by the logic designer.

- **Reuse of non-leaf components.** Traditional logic components are solid black boxes, with no accessible inner structure. The case studies show, however, that structures for memory access, communication, and parallelism are often the reusable component. Differences between family members are generally differences of leaf functions and communicated data items. Accelerator design tools must be able to treat customizable communication and control as the reusable component.

· **Automated sizing of computation arrays.** Performance of FPGA accelerators generally depends on the degree of parallelism that the FPGA can support, which can be a complex function in terms of the application family's structure, the details of the application family member, and the resources available in a given FPGA platform. Tools must support automated choices of computing array sizes, in order to maximize each family member's performance on a given platform.

· **Reusability across platforms.** Although major computer vendors have only shipped one generation of FPGA-based accelerators, it seems certain future FPGA-based products will appear, and will offer increased computing capability. Tools must be able to exploit the additional resources, without the need for changes to the application logic.

The rest of this section presents justification for each requirement, and adds detail to the broad points mentioned above.

4.1.1 Families of applications

The case studies show that, although the application areas have computation structures that differ significantly from each other, any one application area covers a wide range of related applications. String alignment and scoring applications differ in the string data types they address, the scoring functions, local vs. global alignment, and many variations within each category. Rigid molecule interaction applications differ in the rules used for grading interactions. Iterative optimization applications different in objective functions, halting criteria, algorithms for selecting proposed solutions, and rules for selecting new tableaux from scored proposals.

It is not effective to construct accelerators that address individual members of the application family. No one family member is assumed to be of interest to many users. It is not feasible to create a range of point solutions intended to span the application family's range; it is not even possible to anticipate all possible members of many application families. At the same time, highly general, programmable solutions are inefficient at handling any one member of the range of applications. As a result, tools for developing cost-effective accelerators must allow developers to create highly tuned computation structures that are also highly reusable and customizable.

These examples do not cover the whole range of computation types, or even the whole range for which FPGA-based computation is a good match. They do, however, cover enough of a range of computations for several conclusions to be drawn. The case studies each have a distinct pattern of memory usage and a distinct pattern of communication, parallelism, and synchronization. They have different ways of using the various FPGA resources, and different rules for creating arrays of processing elements. Accelerator development tools need not handle every possible computation to be widely useful, but must be able to handle many applications of widely differing character.

4.1.2 Logic designer vs. application specialist

The biggest barrier to acceptance of FGPA accelerators may be the dual skill sets required for creating an effective application accelerator: “*engineers conversant with programming in C or assembly language are often unfamiliar with digital design*” [Hwa01]. An application specialist, such as a biologist or chemist, is the end user who defines the problem to be solved. Efficient hardware designs, however, require a logic designer’s specialized knowledge. Perhaps some of the computing structures examined in chapter 2 could have been inferred using aggressive compiler-based optimization. The case studies of sections 2.1.1 and 2.1.3, however, are counterexamples; structures like these require the logic designer’s insight into each unique design problem. Design idioms like 1000-bit data words, 1000-stage computation pipelines, systolic arrays, and dozens of memory busses are needed for exploiting the FPGA’s potential, at the same time that the application specialist must worry about the details of molecular evolution or biochemical interactions.

There is also cost associated with finding and applying reusable components, leading to the claim that “*... if the time to reuse a part is greater than 30% of the time required to design the part from scratch, design reuse will fail.*” [Gir93]. When the intended users are biologists or computation experts with little skill in logic design, it is reasonable to expect them to take longer "design the part from scratch" than a logic designer. Suppose it were only two or three times as hard for a computational chemist to create an effective application accelerator as it would be for a logic designer. Then 30% of the time for a user to design the part from scratch would be 60-90% of the time it would have taken a logic designer. The original statement assumed an apples-to-apples comparison of implementers' logic design skills, which is not likely to be the case for application accelerators. Other estimates of the development cost saved by component reuse, possibly as much as 85% [Reu99], also assume that the people reusing a component are also logic designers, and therefore underestimate the value of reuse in application accelerators.

For the foreseeable future, both logic designers and application specialists will be required for development of FPGA accelerators. There is no reason to believe that logic designers will become more common any time soon, however. Together, increased demand for logic design skills and [near-] constant availability of logic designers suggest that new multipliers are needed, to enable reuse of one logic designer's output. Many different application specialists should all be able to use a given logic design, for their different applications, without involving the logic designer in the details of each application. This creates two requirements for FPGA-based accelerator tools. The first is that the tools acknowledge the two categories of developers, with their different design responsibilities and different ways in which they prefer to state their design. The second is that the tools allow the two to contribute to the design independently of each other, even at different times.

4.1.3 Reuse of non-leaf components

The fixed, reusable component in each application is its high level structure, including parallelism, communication, memory access, and synchronization. Reuse consists of replacing leaf components within the fixed communication structure: application-specific data types and functions.

This runs contrary to the general style of reusability in hardware components. The reusable component, such as a ROM or digital filter, is generally treated as a black box with no accessible internal structure. Components are parameterized in many ways, as in the case of a filter accepting signed or unsigned inputs, adjustable ranges of values, and selectable coefficients and numbers of taps. Still, it is virtually unheard of to have a reusable component that contains other components, to be chosen by the application developer.

Design and documentation of reusable leaf components is recognized to be more difficult than designing components for one-time use [Reu99]. A widely applicable application accelerator can, however, meet the criteria for successful reuse. If it offers a 100× performance improvement over a serial processor implementation, then the economic value, compared to a PC cluster of similar performance, is high. If the accelerator is used by large numbers of people, then the cost is amortized over many instances. As a result, effective tools must address reuse and parameterization directly.

Part of the problem stems from the design tools available. Languages such as VHDL and Verilog make it easy to specify the interface exported by a component. Such languages, however, have no constructs that make it practical for a component to specify an interface that it imports, i.e. that must be provided to it by some other component. Common object oriented techniques solve all of these problems, however. The requirement is that data types and functions be parameterizable, not just singly but as sets of related items, and existing language technology offers many ways to fulfill that requirement.

4.1.4 Automated sizing

Traditional logic applications have well defined numbers of processing elements and data values. For application accelerators, more processing elements generally give better performance or allow larger problems to be addressed. The desired number of processing elements is “as many as possible.” Computation pipelines generally consist of several stages or subsections. The computation array in each section is subject to size constraints, according to the structure of the computing array and the FGPA resources available for its implementation. Interconnection between subsystems creates interlocked constraints of the connected subsystems. Chapter Appendix B addresses these issues in detail.

The concept of automated sizing is familiar to programmer who program grid or MPP applications, in distributing an application across however many processors are available. It is not, however, supported by current FPGA design tools. Repeating structures are supported, but the numbers of repetitions must be specified explicitly. Given the observed complexity of sizing issues, because of "magic numbers" in permitted sizes, interlocked structures, and application-dependent resource utilization, sizing of computation arrays is a non-trivial task, with answers that are likely to change at almost any change of application-specific data or functions. Clearly, tools for accelerator design must support automated sizing of computation arrays, so that each unique application accelerator can make the most of the FPGA hardware available to it.

4.1.5 Reusability across platforms

New generations of FPGAs appear every year or two, offering successively larger amounts of computing resources and higher potential performance. Between generations, FPGA vendors offer smaller performance increments within FPGA families, including larger family members, faster clock speeds, and lower cost per unit of computing resource. Often, an FPGA with larger capacity will be "footprint compatible" with a smaller one, allowing use of the new chips in old hardware systems, unchanged. It seems safe to assume vendors of FPGA accelerator hardware will track these technology changes as eagerly as PC vendors track changes in processors and system integration technologies. Rapid evolution of FPGA platforms must be taken for granted. Applications must be highly reusable across FPGAs, and tools must support that reuse.

Ideally, an old processor-based application should be binary-compatible with new processors, while still taking advantage of most or all of the new processor's performance improvements. Another solution, usually considered acceptable, is to recompile and relink the old code to take advantage of instructions and new libraries. Source code changes for new platforms are generally considered undesirable, and increasingly undesirable with increasing complexity of changes required for the new platform. When special personnel are required for ports to new platforms, logic designers in particular, acceptance of new platforms will be problematic. Should a logic designer's effort be necessary for porting an existing application to a new platform, those changes should be made once and reused by all of the application clients. Tools for FPGA-based applications must support the kind of inter-platform portability that application developers have come to expect.

Reusability has a second dimension, too: not just within product families, but between product families. Multiple vendors now offer incompatible FPGA hardware, and more seem likely. If an FPGA-based application is popular, however, it should be able to run with minimal changes on the different platforms. If changes are required, they should be made only once per application family, applied generally to all family members, and incorporated through normal compilation and relinking (or analogous operations).

This differs in basic ways from traditional FPGA-based logic design. System builders have generally specified a particular FPGA part early in the design cycle, and taken that choice as an application constant. There has been some reuse of logic designs across successive generations of FPGA-based products, but that reuse has generally been accompanied by major redesign of the application, to make the most cost-effective use of hardware resources. As a result, existing FGPA design tools do not support the kind of application portability needed for widespread acceptance of FPGA accelerators. New tools must offer that kind of portability.

4.2 Implementation decisions

4.2.1 Bridging the semantic gap

The *semantic gap* is the name given to the conflict between high-level representation of computational problems and high-efficiency implementation in some technology [Sni01]. Traditionally, this has meant the gap between a high level language's semantic definition and the instruction set of the machine into which a program is compiled. The semantic gap has long been recognized in the software development world, and has led to fruitful cooperation between compiler writers and instruction set designers.

The semantic gap between application accelerators and FPGA fabric is even wider. The application-specific logic of an accelerator operates at a higher level than the logic of a programming language, because it embodies so many more assumptions about the meanings and basic techniques implied by the application. At the same time, the FPGA fabric operates at a lower level than the processor's instruction set. For example, FPGA hardware requires selection of data word size, something fixed in a standard CPU, and often requires a new word size choice at many points in the design.

It seems safe to assume that the semantic gap will remain a fact of life throughout the visible future. Despite the optimistic claims of some tool developers, a clever logic designer can often create a computation structure unimagined by the tool developer, and therefore inaccessible to the tools. Biologists, chemists, and other potential users of application accelerators can not be expected to equal a logic designer's creativity in using logic resources, or even basic competence in creating systolic arrays, reduction networks, and other native kinds of hardware solutions. As a result, one commercial accelerator builder asserts that it is necessary to "*[adapt] established algorithms to run on our FPGA Accelerator Arrays by reinterpreting from first principles*." [Tim05] There is both practical and theoretical reason to believe that direct compilation of standard algorithms does not generally synthesize into the most effective hardware implementations.

Since the semantic gap can not be bridged, in the general case, it is necessary to address application development in terms of that gap. This acknowledges the application specialist (with high-level knowledge) and logic designer (with low-level design skills) as different individuals. They differ in their needs and responsibilities, and must work together to create the application accelerator. It follows that the design tools' responsibility is not to *convert* application specifics *into* a logic design, but to *connect between* application specifics and an efficient logic design.

An effective tool suite is built around the idea of two different kinds of user: the logic designer and the application specialist or end user. One application specialist would be the computational chemist interested in using the molecule interaction application domain for pharmacological research. That is the user with a computational task in need of acceleration, and the one who needs control over the application-specific details of the computation. The application specialist needs to modify the details of the computation being performed, but is assumed not to have logic design skills.

The logic designer is responsible for creating an efficient memory, computation, and communication structures for each application domain. The logic designer is assumed to have little knowledge of the application area, but is able to paraphrase the critical computations in terms amenable to efficient FPGA implementations. The design assumptions allow application specialists at different sites, and allow application specialists to use the accelerators long after the logic designer has finished the generic logic model for the application domain, so it is assumed that the logic designer can not participate in tailoring the accelerator to the application specifics.

Any solution must also recognize that skilled logic designers are relatively scarce. An effective solution must not require logic design skills for making minor, routine changes to an application accelerator. The logic designer's contribution must be highly reusable, so it can serve many application specialists without the designer's intervention for each one. Finally, in order to decouple the logic designer from the application specialist, it must be possible for them to work at different times.

4.2.2 Need for new design representation

Two questions summarize the most important issues in deciding whether new design concepts require new design tools:

· Why create a new design language when so many exist already?

· Why not create a language that addresses the whole issue of accelerator design?

The LAMP system requires semantic constructs and combinations of constructs that do not exist in current HDLs, so new language features are certainly required. Lack of semantic features is even more compelling reason to move away from existing languages. It is certainly possible to define a new language as a subset of an existing one, but the result tends to cause confusion. Well designed languages are built around interlocking semantic concepts. Removing any one from the language can leave gaps that can not be bridged in any natural way. Adding and removing semantic concepts while leaving a viable language is especially difficult when the base language is as complex as, for example, C++. A logically consistent whole would be impossible within an existing language, if the new semantic requirements diverge too much from the semantic content of that base language. LAMP semantics are described in Appendix Appendix A, showing how LAMP's conceptual base differs from that of existing languages.

In answer to the second question, LAMP is built around the idea that there are at least two groups of designers involved in creation of an application accelerator: a logic designer and an application specialist. The logic designer is responsible for the reusable control and synchronization features of an accelerator, and the application specialist defines the features that customize the accelerator to a specific application. The logic designer must be given the greatest degree of access to the FPGA platform's logic resources, and is responsible for interfacing the accelerator's logic core to the hardware environment defined by the FGPA accelerator board and host system. An application specialist, on the other hand, is assumed not to have logic design skills. The full capability of an HDL would not just be useless to the application specialist, it would be a positive source of confusion and difficulty. The application specialist is responsible for "filling in blanks" left by the logic designer; it is important that the application specialist not "color outside the lines" of the blank spots in the accelerator structure. Natural representation and semantic control both argue that the application specialist and logic designer use different representations for their different parts of the accelerator's design.

It must also be understood that LAMP is a research vehicle. It explores a bounded set of design concepts, without attempting to create the level of generality demanded by industrial logic design. All parts of the tools must be open to modification, without license encumbrance. The tools must express new semantic concepts as needed, and must also be able to omit features that have no place in the LAMP framework. Creating new tools ensures freedom from legal encumbrance, and obviates the problems of semantic mismatch between the new tools and existing bodies of source code. A disadvantage is that new code requires recreation of features already present in existing tools, but it avoids the problems associated with retrofitting legacy code to novel use.

New code also allows clear separation of the novel concepts from established technology. Current EDA tools, especially compilers and PAR tools, have been developed over the course of decades. It would not be practical to recreate the huge amount of work embodied in tools that are readily available from EDA and FPGA vendors. It is also a goal of the LAMP project that it be able to use newer, larger FPGA fabrics as they become available. Tool and chip vendors are already committed to supporting new FPGAs as they become available. By integrating their work, LAMP gets the full benefit of new technologies in synthesis, PAR, and FPGAs, but without changes to the LAMP tools themselves. Instead, use of existing tools allows LAMP research to deal exclusively with its novel content.

4.2.3 Event-driven systems and inverted flow of control

The classic C program starts when its execution environment hands flow of control to the `main()` function in the program. Until the `main()` function executes a `return` statement, normal flow of control is explicitly defined by the application. Applications relinquish temporarily when a system service is requested, but even that is an explicit request by the application.

Event-driven systems invert that flow of control. The execution environment dispatches service requests to the application whenever significant events occur. Programs based on graphical UIs (GUIs) are almost always written in an event-driven way. The system owns the display and input devices. When input an event occurs, including mouse activity or a keystroke, the system records any data items (e.g. character input from the keyboard) and determines which visual element is associated with event's the screen coordinate. The system manages many routine UI tasks, such as changing the cursor image when it enters specific parts of the screen, or highlighting active elements in the GUI. The system also determines whether the application has associated any activity with that event type at that UI element. If so, it passes data describing the event to the event handler. Web servers work in a similar way, dispatching network requests to various scripts, servlets, web applications, and other network event handlers according to the URLs in the arriving HTTP requests, and many other application domains adapt the technique to their various kinds of system events and data elements. In any case, the application has little or no explicit control over the order in which event handlers are invoked. Unlike the `main()` program, code fragments of the event-driven program occur in an order specified by the execution environment.

This style of programming offers many advantages to the application developer. GUIs, for example, tend to be highly standardized and complex systems. The `main()` paradigm would require every GUI-based application to recreate the major GUI responsibilities all over again: managing the display, interpreting input events, and dispatching the event’s data content to the application logic responsible for its handling. Other event-driven systems, such as Sun’s Java 2 Enterprise Edition (J2EE) [Sun05b], take on even more of the overall system’s behavior, including transaction integrity assurance, audit trails, authentication, migration between service processors, and more. Pushing such responsibilities into the execution environment does a lot to simplify application development and to ensure uniform, reliable handling of routine but complex application duties.

Inverted control in GPUs

Graphics processing units (GPUs) implement a form of this software paradigm. They implement a well-defined computation pipeline, starting with geometric objects, and lighting models, and ending with pixels drawn in display memory. The computation pipeline has a fixed structure, with two points at which application-specific logic can be inserted. Graphics programmers create vertex shaders and pixel shaders, small pieces of code to be inserted into that pipeline to represent the geometry and the visual presentation of the object.

The first major feature of this programming model is that the application writer has little or no control over the order in which code elements are executed or data items are processed. This gives the GPU designer many choices about memory structure, scheduling, pipeline structure, and parallelism, all of which contribute to GPUs' impressive execution speeds. The inputs and outputs to shaders are so tightly constrained that data dependencies have limited and explicit form, and control dependencies are non-existent or nearly so. Standard CPUs deal with uncertainties about data and control dependencies using clever tactics like speculative execution, branch prediction, register renaming, and instruction reordering. GPUs, because of their tightly controlled dependencies, can replace all of that logic with additional arithmetic capability. Caches in standard CPUs reduce the effect of unpredictable access to program and data memory. In GPUs, highly predictable reference patterns allow most memory access costs to be hidden by pipelines or by scheduling of runnable tasks. GPUs move most decisions about code and data reference patterns out of the application and into the processor architecture, allowing a different distribution of processor resources than in standard CPUs.

GPUs also constrain the set of operations available to shader writers. They provide a rich set of arithmetic operations on vector data. Early shaders, however, were constrained to some small number of instructions, 8 to 64 [Gra03]. The instruction set did not include conditional execution; even tests and loops based on compile-time constants were innovative when they were introduced. Newer generations of GPUs allow longer programs and some conditionals (though they may be discouraged for performance reasons), but still offer a severely constrained programming model compared to what a C programmer is used to. These constraints are considered acceptable, however, because they are still well matched to the specific graphical tasks being performed.

Applicability of inverted control in FPGA accelerators

This paradigm offers an attractive way to involve hardware-naïve users in the design of application acceleration hardware. Within any one application family, large parts of the behavior and structure of the application are common across all family members. The common subset of an application family generally includes data access, system interface, synchronization of multiple parallel processing elements, interface to the host system, hardware resource allocation, and handling of communication between processing elements. Some of these, such as data access, are dramatically different in von Neumann or Harvard architectures than in an FPGA with hundreds of independently addressable memories. Communication and task allocation are also very different in a typical MPP or computing cluster than in an FPGA. Systems with multiple processors generally favor large units of parallelized work with infrequent, low-volume communication. FPGAs offer the opposite, fine-grained parallelism, down to individual arithmetic operations, and massive on-chip communication bandwidth –broadcast operations at every clock are often feasible. Even if a programmer is skilled in parallel application development, that skill is probably tuned to MPP and cluster programming, the opposite in most ways of what works well in FPGAs.

The event-driven paradigm consists of two parts, system control that acquires and dispatches application data to the proper handlers, and the application-specific handlers themselves. This dichotomy is a good description of an effective FPGA based application accelerator. Data access, synchronization, resource allocation, and dispatch are tasks well suited to a hardware designer's skills. The problem, then, is to phrase the application specialist's part of the accelerator as an event handler. In a software system, that would be a "callback", a function of defined interface to be filled in by the application developer.

There are many ways to describe the application specialist's additions within the framework created by the hardware designer. Using an analogy to the standard C library, one treat the application specifics much the same way as the comparison function parameter to the `qsort()` function [Pla92]. That allows one sorting framework to sort any data type, according to any specified order, by phrasing the ordering function as a parameter. This, however, violates the belief that parameterization is defined in terms of "*… feature[s] that can be modified … without affecting the application's essential functionality*," [Giv00] where common examples include buffer sizes or ROM dimensions. The communication subsystem can also be considered as a reusable communication component, with empty slots inside in which communicating sub-components are instantiated. This also goes against the common belief that reusable components are necessarily leaf components, and that "*Reuse is in the first place a matter of reusing functionality, not structure.*" [Sch99] Others have claimed that "*parameterization of functionality … has taken the form of operating modes. … This allows optimization to automatically remove unused circuitry as appropriate*." [Gir93] It is implicit in this school of thought that the component creator must anticipate every possible set of behaviors for the component, and allow the component user to select among subsets of that choice. Despite the general trend towards treating components as leaf components with more or less fixed inner function, some authors have addressed the idea of reusable communication components [Ver00].

Reuse of non-leaf components, with parameters that embody functions, is well established in the software development world. Non-OO languages like ANSI C, as noted above, support function parameters. OO languages offer other mechanisms, as well. The *Template Method* and *Strategy* design patterns [Gam94] suggest techniques quite similar to each other. Both assume that the missing inner component is defined by an abstract class interface. The *Strategy* pattern is built around composition of an undefined object into the fixed application logic, and *Template Method* assumes compile time resolution by subclassing. Given the static nature of object binding in logic design, the line separating the two patterns becomes indistinct.

5 LAMP TOOLS: DETAILS OF OPERATION

This chapter describes the LAMP tool flow and the basic features of LAMP tool operation. It starts with descriptions of the responsibilities of the two participants in accelerator design, the logic designer and the application specialist. Next, it places each one's responsibilities in the context of the LAMP tool flow as a whole. Section 5.3 then presents the LAMP *model*, the set of files that defines the accelerator family its customization to a particular application.

Those sections define the input to the LAMP tools. The remainder of the capter describes how that input is processed. Section 5.4 describes the process that connects abstractions in the LAMPML input language to actual instances of VHDL logic. Section 5.5 outlines the connection between LAMPML functions or type definitions and their VHDL representations, taking into account LAMP's type polymorphism and VHDL's lack of it. Section 5.6 shows how LAMP declarations are integrated with the model's annotated VHDL code. Finally, section 5.7 addresses the synthesis estimation algorithms in the initial version of LAMP, including some of the factors that cause accuracy problems in the estimates.

This chapter uses parts of the XML-based LAMPML markup language and the CLAMP high-level notation defined in Appendix A. Although different in appearance, they are alternative representations of largely the same semantics. Because of complications caused by integrating LAMPML with HDL code, CLAMP notation can not be used for HDL annotation. LAMPML can be used to represent all parts of a LAMP accelerator design; CLAMP is a semantic subset of LAMPML with enhanced readability.

5.1 Logic designer and application specialist

Normal use of LAMP tools has two major phases, corresponding to the two classes of developer involved in accelerator design. The first phase is entirely the job of the logic designer, or the group of people working together to create the logic design. The second phase comes when the logic designer hands off the accelerator model to the application specialists. The application specialist and logic designer are assumed to be independent of each other. In practice, however, application specialist will likely advise the logic designer about algorithms, features, and options needed in the accelerator.

The logic designer creates the parameterized structure of the accelerator during the first phase. That includes the annotated HDL for the accelerator, the application abstraction used by the application specialist, and the FPGA resource descriptions. This job is essentially that of a logic designer in any other context, but with additional responsibilities. One is that the logic design must be parameterized using LAMPML markup, so that application-specific data types and leaf computations can be replaced. Normal HDLs allow only parameterization in terms of data values but LAMP allows types and functions as design parameters, so this is more complex than normal parameterized design. Still, it's just an extension of normal HDL parameterization.

The logic designer has a fundamentally new responsibility, however. It is assumed that most accelerators are built around memory and computing arrays that can vary in size, and that larger arrays give better application speedup. As a result, the logic designer must phrase the various computation and memory arrays in terms of their sizes and FPGA resource estimates, in order to determine the values of structural parameters that give the most desirable accelerator possible. Section B.1 states the general mathematical formalism for sizing the computation arrays optimally, section 5.7 describes LAMP's technical support for synthesis estimation and sizing, and appendix B gives an example of sizing logic represented in LAMP code.

This is not as radical as it might sound at first. Logic designers always have to deal with FPGA resource limitations, and with getting the most capability out of the most economical hardware. The difference is that the sizing knowledge must be phrased algorithmically, parameterized for all the features that affect and constrain accelerator sizes, and packaged in LAMP code as part of the accelerator model.

The second phase of the accelerator's life comes after the logic designer has verified the accelerator's LAMP model. At that point the logic designer has no further role, unless additional features are required or porting between FPGA platforms becomes necessary. The model is turned over to the application specialists, who put it to use in their computations. Although logic design is generally the job of one developer (or development team), there is no limit on the number of different application specialists that use any one model. In some cases, the person who customizes the accelerator model will not be the person who uses the accelerator, and any number of end users could share one customization of an accelerator. It simplifies discussion to assume that the application specialist both customizes and uses the accelerator, and that application specialists are generally independent of each other.

5.2 LAMP tool flow

The following discussion traces construction of an accelerator through the LAMP design flow. Each of the following subsections corresponds to one of the items of Figure 21. At this writing, Figure 21's *LAMP programming environment* is being planned. *Application specifics* are LAMPML or CLAMP text. *LAMP Compilation Tools* exist as prototypes.

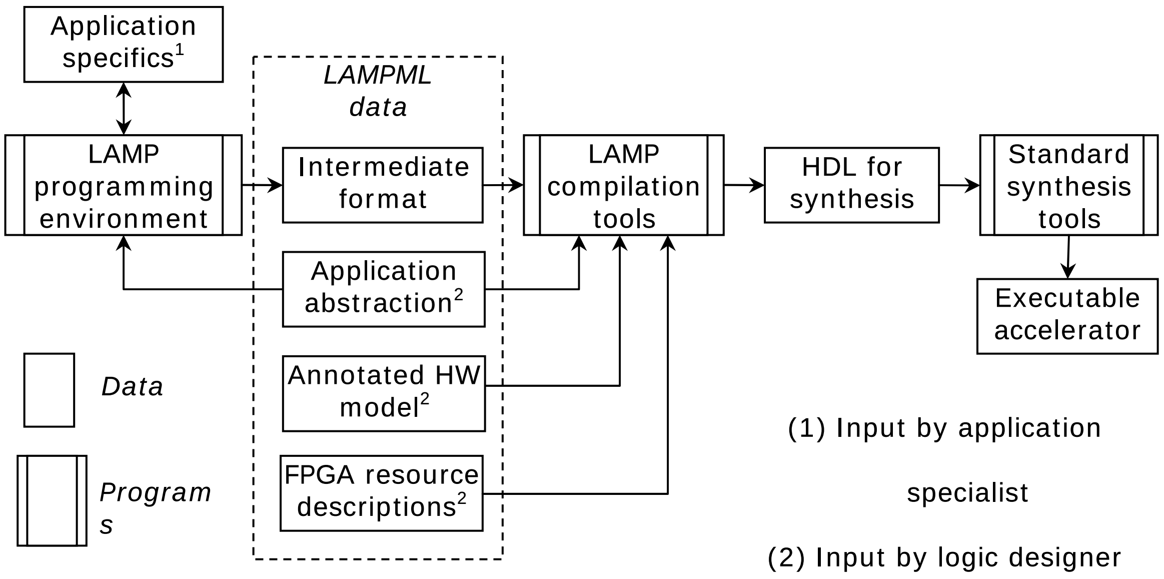


Figure 21. LAMP tool flow

The *Standard Synthesis Tools* are any of the available VHDL compilers, along with placement and routing tools and other tools needed for converting VHDL code into a bit file.

Logic designer's input

The first inputs for any accelerator are defined by the logic designer, using XML-based LAMPML (LAMP Markup Language). The Application Abstraction is the first of the logic designer's inputs, an abstract description of the data types and functions that that are left undefined by the *Annotated Hardware Model* (or just model). The FPGA Resource Description describes the amounts of each computing resource available on a given FPGA.

Assuming the Rigid Molecule Interaction application of section 2.1.3, the application abstraction contains formal definitions for interfaces to the functions and data types that describe and score molecule voxels, not tailored to any specific scoring strategy and voxel representation. The *Annotated HW model* of Figure 21 specifies the communication paths, control mechanisms, and host interface items that are constant across all members of the application family. Section 5.3 discusses the application abstraction, model, and resource definitions in detail.

LAMP programming environment

At present, the LAMP programming environment is under consideration but has not been implemented. Its place is currently held by the text editor that the application specialist uses for creating LAMPML or CLAMP specializations of the application abstraction.

The foreseen programming environment will accept the application abstraction as input. This abstraction identifies the functions and data types that are used but undefined in the model. Based on the application abstraction, the programming environment will create a semi-graphical user interface for supplying the information needed by the model. Extensions to LAMPML (and possibly CLAMP) will probably become necessary or helpful in preparing the model for GUI presentation.

Application specialist's input

The application specialist makes configuration decisions at the programming environment's UI. Application specialists enter data type definitions and function definitions in a simple C-like representation, similar to the one described in section A.3, filling in text fields within the GUI. The programming environment saves the human-readable form of input for UI purposes, but converts it into validated LAMPML for further processing. If the user needs to make further changes, the programming environment re-opens the user input data, edits the data according to user input, and creates new copies of the intermediate format output.

LAMPML intermediate format

Although LAMPML is editable text data in XML format, very few users would find it a natural or convenient way of expressing application logic. The XML intermediate format is readily machine readable, however, and separates the input and parsing issues from the compiler's internal processing. This corresponds to the compiler's intermediate code, a step between a compiler's front end and back end that is normally not visible to the user.

Because LAMPML is a text-based XML format, it can be edited manually. In order to focus on the central LAMP design issues, this is currently used as the primary input format. A more complete system would hide this level from nearly all users, and allow a more natural means of expressing the application-specific logic of the accelerator. The Convenience LAMP (CLAMP) language is one such representation, described in Appendix A.3. It is not, however, a functionally complete representation of LAMPML.

LAMP compilation tools

LAMP accepts the hardware model, defined in terms of the application abstraction, and combines it with the LAMPML from of the user input. This phase includes synthesis estimation and scaling, allowing the largest possible computation array for the logic resources available and required. The output from this compilation is Figure 21's *HDL for synthesis*.

These tools do the real work of combining the application-specific code with the hardware model. This stage integrates user logic with the HDL application framework. It also performs synthesis estimates for the code provided by the user, and evaluates estimation functions provided in the hardware model. Together with the FPGA resource information, these imply the largest number of PEs that the available FPGA fabric can support.

The compilation phase should also generate interface code fragments for the host programming environment. This small amount of code in ANSI C establishes the common data formats shared between the host application and the hardware accelerator. These code fragments supplement the vendor's device drivers, and are not intended as replacements. Generation of the interface modules form LAMP code has not yet been implemented, however.

Synthesizable HDL output

LAMP tools do not perform low-level synthesis directly. Instead they create output in a synthesizable HDL, and hand synthesis, placement, and routing to other tools. This has the advantage of keeping as much as possible of the LAMP tools and hardware model independent of specific FPGAs, allowing easier porting to new accelerator hardware or to FPGAs with larger amounts of resources.

For users familiar with standard programming environments, this corresponds roughly to the assembly-language output from a compiler, before conversion to binary format. In this case, however, the output is in a synthesizable HDL. Although this form of the application is human-readable, it is not intended for human use or modification. Still, some logic designers seem more comfortable with design tools when they have at least the possibility of hand-tuning the code generated automatically, just as some programmers have been known to hand-tune a compiler's assembly output.

Standard synthesis tools

The LAMP tool set works with any available synthesis tools; it has no architectural dependency on any particular HDL or compiler. That said, it is understood that tool-dependent pragmas or VHDL attributes in synthesizable HDL code are sometimes critical for achieving performance and resource allocation goals. To date, those tool-dependent features have all been managed within the annotated HDL model.

LAMP tools interact with an HDL and generate HDL code, but do not depend fundamentally on any particular HDL. VHDL has been chosen for the initial LAMP implementation, and parts of LAMP are written speicifically with VHDL in mind. LAMP's architecture identifies and isolates those dependencies, allowing a new HDL to be supported without basic changes to the LAMP tool structure.

Executable accelerator

The final product is a bit file for implementing the accelerator in the target FPGA.

The applicaiton user treats this roughly as executable code to be run on the hardware accelerators. This is analogous to a graphics shader programs in compiled form, ready to load into graphics accelerator hardware. It is also possible for the user to create and reuse multiple bit files that serve different purposes, much the way a graphics programmer writes shaders for different purposes.

It is not generally necessary for each application specialist to recompile the accelerator for each usage. Most application specialists are likely to try a few combinations, then find the one (or few) that best serve their use of the application. It's been assumed that application specialists act independently of each other. Sharing of application data or compiled accelerators is likely, especially between members of one research team, but is not relevant to the basic idea of how the application accelerators are most likely to be used.

5.3 The LAMP "Model"

Figure 22 shows the parts of the LAMP *model*. All of the blocks in that diagram represent LAMP design input, in LAMPML. CLAMP offers convenience, but serves the same purpose. HDL input is contained in and annotated with LAMPML elements.

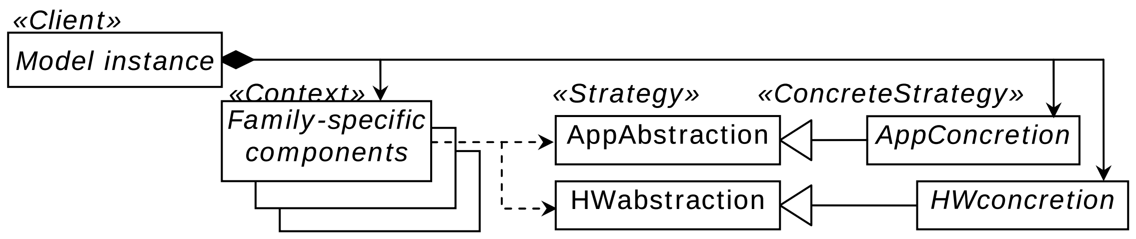


Figure 22. Structure of LAMP's accelerator model

Stereotype labels are roles in the *Strategy* design pattern [Gam94]. A *design pattern* is a widely used and very generic solution to a recurring problem in system design. Although design patterns arose in the software development literatures, they are gradually becoming accepted in discussion of hardware systems. In this case, the Strategy pattern defines a specific set of relationships between the design components that implement the fixed high-level structure and the swappable low-level steps of an algorithm.

5.3.1 Accelerator model components.

Figure 22 contains the following blocks:

· **Model instance**. This is the aggregate of all the LAMPML design blocks. When the *AppConcretion* blocks for a specific member of the application family is provided, this can be processed by the LAMP compilation tools to generate HDL for synthesis. This normally does not contain any application logic itself, but acts as an index to the other blocks in the accelerator as a whole. A generic model instance describes the accelerator in the abstract, and specific instances modify for the *AppConcretion* blocks and bindings that define a particular member of the family of applications.

· **Family-specific components**. These are the annotated HDL, parameterized so that specific data types and functions can be provided to tailor the model to a specific member of the application family.

· **HWabstraction**. This contains the definition of the salient FPGA resources in abstract form. This isolates the rest of the model from the exact numeric values for each resource

· **HWconcretion**. A different *HWconcretion* represents each different FPGA by creating concrete definitions for the resource abstractions defined by the *HWabstraction*. The logic designer creates one of these sets of definitions for each FPGA platform of interest, but only one can be used at a time. In the ideal case, where different hardware accelerators differ only in being built from different footprint-compatible members of one FPGA family, a the *HWconcretion* in the *Model instance* would be changed to represent the new FPGA's resource amounts. The model would be recompiled with no further source changes.

· **AppAbstraction**. This abstraction isolates the common logic of the application family (in the *Family-specific components*) from the unique logic of the family member (in the *AppConcretion*), and vice versa. It defines the set of data type, function, and value abstractions needed to customize the model for any one member of the application family.

· **AppConcretion**. This is the only design block that the application specialist modifies, unless lack of tools requires hand-editing of the *Model instance*. The *AppConcretion* contains the application-specific implementations of the definitions called for by the *AppAbstraction*.

Only the *Model instance* and *AppConcretion* change for different members of the application family. The *AppConcretion* comes from the application specialist, and is the only block in that diagram that is not properly part of the model. The *Model instance* is largely indexing information that must be customized to each application's specific set of design blocks. All other blocks represent data provided by the logic designer.

In LAMPML terms (Appendix A) the *Model instance* is an `application` element, the *Family-specific components* are `entityDef` and possibly `class` elements, and the others are all `class` elements.

5.4 Object binding

Although LAMP uses a standard HDL for synthesis, it is not limited to that underlying HDL's capabilities. In particular, LAMP allows more flexibility in component signal typing than does VHDL. This section addresses a VHDL example using LAMP's semantics for object instantiation, showing how it extends the capabilities of VHDL. Code samples in this section are fragmentary, but adequate to show the semantics of LAMP language constructs.

Because LAMP uses two linguistic levels, LAMPML for integration and VHDL for logic implementation, there are two meanings for a component *instance*. VHDL component instances are the usual, familiar instantiations of synthesizable entities. LAMP adds the notion of a *logical instance*, a LAMP entity with its `import` symbols bound to appropriate type names and values. It does not by itself create anything synthesizable, but provides a definition that can be instantiated by the HDL code. Loosely speaking, a LAMP entity with unbound type parameters corresponds to a C++ template class. A logical instance is one binding of type parameters to that template class, but does not imply C++ objects of that type-parameterized template. The VHDL component instance corresponds to a C++ object of that filled-in template type.

Code sample 2 shows the original VHDL statement of the delay line's interface. Data type `Datum` is assumed to be defined elsewhere in the application, and is not relevant to current discussion. For concreteness, assume that this defines a reusable component that implements a fixed delay of some number of clock cycles (set by `delayAmt`) Any `Datum` value presented at the `datIn` port will appear at the `datOut` port `delayAmt` cycles later, where a cycle is a transition from logic level 0 to 1 on the `sysClk` input. The actual function of the component does not matter, however. Current discussion addresses only the instantiation logic.

Code sample 2. VHDL component declaration

```
entity DelayLine is
    generic(delayAmt:   natural)
    port(sysClk:        in std_logic;
        datIn:          in Datum;
        datOut:         out Datum);
```

The problem is that VHDL allows only one definition of the `Datum` data type, limiting the reusability of the component. The designer could, instead, have achieved reusability at the cost of typing by requiring the application data to be flattened into a `std_logic_vector` bit representation, which effectively defeats strong typing. LAMP allows both – strong typing and reusability, using a mechanism related to the template construct of C++ or Java.

5.4.1 LAMPML entity definitions

Component definitions for LAMP are written in LAMPML, an XML-based format. The top-level element of a LAMP component is an entityDef element, which starts as shown in lines 1-7 of Code sample 3.

Code sample 3. LAMP entity declaration

```
1. <entityDef name="DelayLine">
2.     <typeImport name="Datum" type="baseType"/>
3.     <symImport name="delayAmt" type="natural" />
4.     <typeImport name="FpgaContent" type="FpgaResource" />
```

```
5.    <symExport name="entName" type="symbol" >
6.       <uniqueID prefix="delay_" />
7.    </symExport>
```

Line 1 is the top-level element. It continues until `</entityDef>` appears in the file, like the closing parenthesis around the group of definitions. Line 2 states that a data type must be imported into this definition. The `type` attribute in this declaration constrains the data types that can be imported into this symbol – any valid import must be a subclass of `baseType`. Since `baseType` is the root of every inheritance hierarchy, any data type is allowed to be imported into this symbol.

Line 3 takes the place of the `generic` declaration in Code sample 2. The `delayAmt` variable has been moved from VHDL to LAMP so that LAMP logic can use this value in synthesis estimation and sizing. Line 4 imports information about the FPGA platform, so that estimation and sizing can match resource utilization to resource availability. Details of estimation and sizing are described in detail in section 5.7, so will not be discussed further here.

Line 4 states that symbol `FpgaContent` is used locally as a class name. The specific class to which is refers is defined elsewhere, but that class mut be a subclass of `FpgaResource`. This lets the current context use the symbol names defined by `FpgaResource`, but defer actual symbol definitions until the entity is used.

Lines 5-7 generate a data value in this entity definition, scoped to be visible outside of this compilation unit. The value assigned to is it a globally-unique symbol to be used as externally as the name of any logical instance of this entity. This is required by the HDL code, since different instances of this entity can have different bindings to the `typeImport` symbols – as if VHDL allowed `generic` parameters to specify data types, instead of just values. When LAMP generates an actual VHDL entity definition for this component, it will have the unique name specified by `entName`. This will be demonstrated in detail later in this example.

VHDL code is interleaved with the LAMPML markup. The following code fragment appears later in the compilation unit of Code sample 3. Other VHDL code would also have been present in a real code file, but has been omitted for clarity.

```
8. entity <varRef name="entName" /> is
9.    port(sysClk:              in std_logic;
10.         datIn:               in <symRef name="Datum" />;
11.         datOut:              out <symRef name="Datum" /> );
```

This shows how closely XML-based LAMPML markup is tied to the VHDL code. Rather than using actual VHDL symbol names for the VHDL entity name and signal data types, LAMPML symbols are used. The way, the VHDL code can be generated differently for each logical instance of the entity created in the LAMP code.

Line 8 replaces an actual entity name with a reference to the `entName` symbol generated at line 5. Because this symbol has a different value for each logical instance of the entity, the VHDL code can be reused in ways that are not normally possible for VHDL. In particular, lines 10 and 11 show signals without explicit type definitions. Instead the type definition is refers to the symbol `Datum`. The `Datum` symbol resembles a formal parameter to a function. Its exact meaning is not known to the source code that uses the symbol. Instead, meaning is assigned by the binding of an actual type definition to the `Datum` symbol in a logical instance of this entity.

The `Datum` symbol used in lines 10-11 is a LAMP definition, but must be converted into a HDL type definition accessible within this entity and the entities that pass signals to these ports. When the type is one of LAMP's primitives, `boolean`, `integer`, or `natural`, conversion to VHDL is straightforward. Ranges of `integer` and `natural` values also have obvious VHDL translations. LAMP also allows tuple definitions in `typeDef` declarations. If a `symRef` element makes use of one of those types, as in this example, LAMP generates a corresponding VHDL `record` declaration. Tuple elements may themselves be tuples, just as fields of a VHDL record may also have some record type. In those cases, one `symRef` triggers multiple VHDL type declarations, for as many levels of fields within fields as necessary. Recursive declarations are errors, so all LAMP tuple types can be cast into VHDL.

5.4.2 Logical and synthesizable instances

Lines 1-11 of Code sample 3 are just a small part of the whole LAMPML file. It continues with more VHDL and LAMPML code. Only the parts relevant to LAMP component instantiation have been shown. The example continues below, with code taken from a separate file where the `DelayLine` entity is actually used.

```
1. <entityUse entity="DelayLine" name="audioDelay">
2.    <bindType importSym="Datum" bindTo="audioSample" />
3.    <bindType importSym="FpgaContent" bindTo="FpgaContent" />
4.    <bind importSym="delayAmt">
5.       <literal type="natural" /> 100 </literal>
6.    </bind>
7. </entityUse>
```

Code sample 4. LAMP entity usage, creating a logical instance

The `entityUse` element at line 1 creates a logical instance of the `DelayLine` entity. The logical instance is named `audioDelay`.

Import symbol `Datum` of `DelayLine` is bound to the locally defined data type symbol `audioSample` at line 2. The `audioSample` symbol need not have any meaning within `DelayLine`. Instead, the type referred to by `audioSample` should be thought of as an actual parameter to formal parameter `Datum`. The `bindType` element creates the association between the `audioSample` type and the `Datum` import symbol.

Line 3 performs a similar binding. This `bindType` element deals with two different uses of symbol name `FpgaContent`. The first, in the `importSym` attribute, refers to a symbol defined internally to the `DelayLine` entity. The second, in the `bindTo` attribute, identifies a symbol in the entity that is instantiating `DelayLine`. The two attributes use different symbol resolution scope, so the two different uses of `FpgaContent` refer to different things.

Lines 5-7 assign a numeric literal value, 100, to `audioDelay`'s `delayAmt` symbol. Any natural-valued expression could have been used here, but details of LAMPML expression handling are deferred to Appendix A.1.4.

This completes the definition of logical instance `audioDelay` of entity `DelayLine`. At this point, the LAMP tools resolve the meanings of all unbound symbols in `DelayLine`. The unique value of `entName` (line 5, Code sample 3) is assigned, and LAMP creates a synthesizable VHDL file with the symbol bindings replacing the symbol references in that file.

The `entityUse` declaration does not, however, create a VHDL component instance. The VHDL code must do that explicitly, using a reference to the logical instance like the one below:

```
8. AudioLag: component <varRef name="audioDelay:entName" />
9.    port map(sysClk, digitizerInput, laggedSample);
```

This is exactly a normal VHDL component instance, except that the entity name has been replaced by a LAMP variable reference. The `varRef` element fetches the value of the named variable. In this case, the name is scope-qualified. The colon ":" separator in LAMPML serves the same syntactic purpose as a period "." separator in a C++ or Java program. It orders that later parts of the colon-separated sequence be resolved in the context defined by the first symbol. In this case, it requests the `entName` variable defined within the entity that `audioDelay` instantiates. That is the symbol uniquely generated for that logical instance of `DelayLine`. Definitions of the `digitizerInput` and `laggedSample` signals are not shown, but are assumed to be of type `audioSample`.

Once the `audioDelay` logical instance has been defined, VHDL instantiation can create any number of synthesizable instances. LAMP's extension to VHDL semantics, however, comes from the ability to create multiple logical instances of a single entity and to create synthesizable instances of them. For example, Code sample 4 could continue as shown below:

```
<entityUse entity="DelayLine" name="pipelinedControl">
   <bindType importSym="Datum" bindTo="controlPacket" />
   <bindType importSym="FpgaContent"
                               bindTo="FpgaContent" />
   <bind importSym="delayAmt">
      <literal type="natural" /> 2 </literal>
```

```
16.       </bind>
17.    </entityUse>
18.    CmdBypass: component
19.                 <varRef name="pipelinedControl:entName"/>
20.       port map(sysClk, cmdIssue, cmdPerform);
```

The `entityUse` element follows the same pattern as lines 1-7, but binds a different type to the data ports of the delay line and a different unique name to the entity definition. This logical instance also has a different name, `pipelinedControl` rather than `audioDelay`. As shown in lines 18-20, that lets the VHDL developer choose which logical instance is to generate a synthesizable instance.

5.4.3 HDL output for logical instances

This generates a different VHDL entity definition than the `audioDelay` logical instance. Generated code for the two would have the following forms:

```
entity delay_000139 is
 port(sysClk:     in std_logic;
      datIn:         in controlPacket;
      datOut:        out controlPacket);
      ...
```

and

```
 entity delay_000076 is
port(sysClk:    in std_logic;
     datIn:        in audioSample;
     datOut:       out audioSample);
     ...
```

The entity names (delay_000139 and delay_000076) are only representative, because they are generated by the uniqueID expression at line 6 in Code sample 3. The prefix string delay_ was chosen by the developer, but the numeric suffix is chosen to guarantee global uniqueness of the generated symbol, and is not guaranteed to have any particular value or even a repeatable value. This LAMP mechanism allows a higher degree of type-safe reusability than is possible in the standard VHDL language.

Although VHDL generics are not shown in this example, they are still available to the developer. The reason for moving Code sample 2's delayAmt parameter from VHDL to LAMP was to make the value accessible to calculations (not shown) in the LAMPML markup, for evaluation during compilation. LAMP does not restrict the VHDL features available to the developer. Instead, LAMP adds additional features, which were used in this case.

5.5 Function synthesis

One of LAMP's goals is to be able to define functions that represent application logic and to integrate those functions into the synthesized HDL code. This problem has several parts: defining the expressions that represent application logic, passing those executable functions as parameters to the HDL code, and instantiating the functions as synthesizable HDL.

5.5.1 LAMP functions and expressions

As with all of LAMP input, expressions in LAMPML use XML-based syntax, described in Appendix A.1.4. This format is easily machine-readable, but unwieldy as a human interface format. For example, the expression *X * 9 + Y * 7* would be written as:

```
<functionCall name=”add”>
 <functionCall name=”multiply”>
   <varRef name=”X” />
   <literal type=”natural”> 9 </literal>
 </functionCall>
 <functionCall name=”multiply”>
   <varRef name=”Y” />
   <literal type=”natural”> 7 </literal>
 </functionCall>
</functionCall>
```

Convenience LAMP (CLAMP), described in Appendix A.3, presents syntax more familiar to C or Java programmers. A translator has been written to convert CLAMP source text into LAMPML, but CLAMP can not handle HDL text. Rather than force the reader to wade through lengthy XML, this section uses CLAMP syntax for clarity. The reader must understand that this isn't the actual format used by LAMP, but can be converted into the XML format.

LAMP allows synthesizable functions to be written into the annotated HDL directly, or to be passed through a type import. LAMP processes the synthesizable form of the function the same way in either case. A LAMP function is not processed into synthesizable form if it is used only during LAMP processing, i.e. if its parameters are all LAMP constants or expressions that can be evaluated at compile time. Such function references generate control values and aid in decisions about LAMP's processing of the HDL code. Once their values have been used, however, the function body does not need to be preserved.

Generation of a functions synthesizable HDL form is triggered by the LAMPML construct that allows HDL text to be passed as a parameter into a LAMPML function, the `passParam` element, described in Appendix A.1.4, page 281. LAMP does not process the HDL text other than to place it verbatim into a function call, so error checking and type compatibility can not be verified for any use of this construct. In use, such a function call might look as shown in the following example of VHDL with LAMPML annotation:

```
1. constant vhdlSym : natural := someVar + otherVar;
2. constant symA : natural := 5 +
3.    <functionCall name=”lampFunction”>
4.       <varRef name=”lampSym” />
5.       <literal type=”natural”> 9 </literal>
6.    </functionCall> ;
7. constant symB : natural := 2 *
8.    <functionCall name=”lampFunction”>
9.       <varRef name=”lampSym” />
10.         <passParam> vhdlSym * 9 </passParam>
11.      </functionCall> ;
```

Code sample 5. Use of `passParam` in LAMPML function call

Line 1 of Code sample 5 is plain VHDL code. It shows a simple expressions that evaluates to a constant when the VHDL compiler runs, using VHDL symbols only. The assignment at line 2 performs an assignment that combines VHDL and LAMP expressions. In this case, the LAMP expression is wholly in terms of LAMP values and constants, so it is evaluated entirely within LAMP. The generated HDL code replaces that `functionCall` element with the HDL constant to which it evaluates. This does not trigger generation of HDL code for the `lampFunction` definition.

The assignment at line 7, however, does cause HDL code generation for `lampFunction` because of the `passParam` element at line 10. The `functionCall` element is replaced with a call to a VHDL function, and the definition of `lampFunction`'s body is instantiated as a function definition local to this `entityDef` element. This, in turn, triggers HDL code generation for data type definitions used within that function. It also creates HDL code for any auxiliary functions invoked by the `lampFunction` body, and so on recursively. LAMP performs some optimizations having to do with constant values in expressions, but generally creates direct HDL representations of the LAMPML expressions.

There are some restrictions on the functions that are used for logic synthesis. In particular, recursion is not permitted in synthesis. LAMP, however, uses recursion as its mechanism for performing repeated operations, in place of C-like loop constructs. As in the definition of VHDL functions, the developer must make sure that recursive functions are used only for compile-time constant expressions, not for synthesis.

LAMP's function syntax is straightforward, and VHDL equivalents for most of the primitive functions and data types should be obvious. LAMP's `if-then-else` operator, however, requires generation of auxiliary functions. VHDL does include a similar construct, the `assign-when` form of signal assignment. This can only be used for VHDL signals, not variables, and can not be used in an arbitrary expression context. If considered as an operator, that conditional construct would have the weakest of all operator bindings, would be allowed on the right-hand side of assignments only, and would not be allowed inside of parentheses or function calls. This is far more restrictive than LAMP's conditional operator, which is allowed in any expression context.

Each auxiliary conditional function has the following form, in VHDL:

```
function sel_000013(tst: boolean;
   ifT, ifF: someType) return someType is
begin
   if tst then return ifT; else return ifF; end if;
end function sel_000013;
```

The type name used for the `someType` symbol is deduced from the types of the operands to the LAMPML `if-then-else`. As many differently-typed selection functions are created as needed for handling `if` expressions of different types, each with a different numeric suffix. Specific numbers for suffixes in the generated names have no significance and can not be guaranteed.

VHDL allows functions in signal assignments as well as constant expressions, so the HDL functions generated by LAMP can be used in signal expressions. This is the feature that connects the LAMPML or CLAMP input to the synthesized output. LAMPML function calls can accept signal inputs via the `passParam` element, that triggers HDL generation for the function called, and that in turn becomes synthesized logic.

5.5.2 Class bindings in logical instances

LAMP configurability is based on replacement of abstract superclass symbols by subclass concretions of those abstract symbols. The annotated HDL code defines a set of abstract data types, functions, and constants, and is customized by binding a concrete subclass to the abstraction.

This example uses a reduction array, where a vector of data values is examined and one value produced as output, based on performing some operation down the length of the vector. This include vector sums, AND of all elements, parity computation, arithmetic *max* or *min*, priority encoding, and other functions. Using CLAMP notation (see Appendix A.3) for convenience, the variable parts of this array could be written as follows:

Code sample 6. Reduction array abstraction

```
class Reduction {
   typeDef inputType;
   typeDef innerResult;
   typeDef finalResult;
   const innerResult resultInit;
   function innerResult reduceOne(
      natural    curIndex,
      inputType        curValue,
      innerResult resultSoFar);
      function finalResult makeFinal (
         innerResult   lastResult);
   };
```

This assumes annotated VHDL also exists to create synthesizable instances of reduction arrays based on these definitions. The VHDL code is assumed to expand to a structure resembling the one shown in Figure 23.

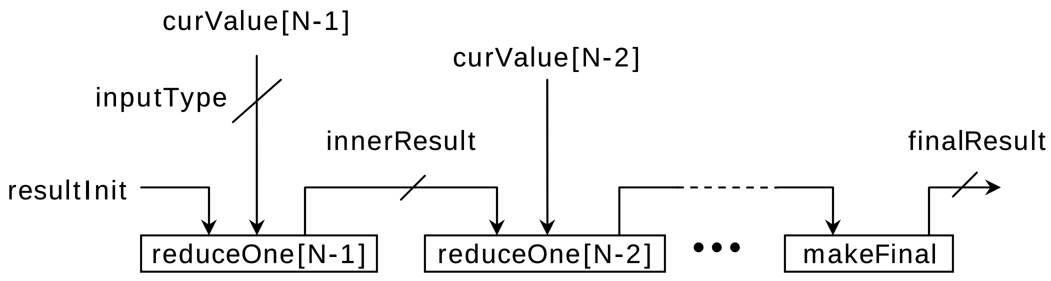


Figure 23. Sample application: reduction

Lines 2-4 state that the model uses some data type definitions for input values, for temporary results, and for results of the reduction operation. The types must be specified, and need not be the same. For example, a priority encoder would generate an integer result for binary inputs, so `resultType` could be defined as an integer range and `inputType` as a boolean value. Line 5 states that the reduction starts with some initialization value. That would be 1 for a product of all inputs, `false` for a boolean OR of all inputs, the lowest possible value for a *max* of all inputs, or any other value appropriate to the specific usage.

Lines 6-9 specify the calling interface for the reduction function. The `curIndex` value is a convenience, the index number of the current element in the reduction array. It must be present syntactically, whether it's used or not. If it is not used within the function body, however, it will be eliminated in synthesis. The `curValue` input represents the value of the current element of the array to be reduced. The `resultSoFar` value is the intermediate result in the running sum. This function combines the current value with the running sum to produce the next value of the running sum for the next element in the reduction array. Lines 10-11 capture the idea that the computation, in some cases, requires some book-keeping values that are not part of the final answer, and defines a function that converts the computation into a final result

Code sample 7 shows one concretion of this interface, for finding the array index that has the highest-valued element. This class would normally be stored in a file separate from class `Reduction`.

Code sample 7 . Concretion of reduction array – position of the largest value

```
1. class MaxPos extends Reduction {
2.    const natural inpBits := 12;
3.    const natural maxIndex := 31;
4.    typeDef inputType integer[inpBits];
5.    typeDef finalResult natural[#maxIndex];
6.    typeDef innerResult {
7.       finalResult index,     inputType  maxVal };
8.    const typeDef resultInit := new innerResult (
9.       index := 0, maxVal := 1 << (inpBits-1) );
10.      function innerResult reduceOne(finalResult curIndex,
11.         inputType curValue, innerResult resultSoFar)
12.      {
13.         innerResult thisIsMax := new resultType(
14.            index := curIndex,     maxVal := curValue);
15.         return if curValue >= resultSoFar.maxVal
16.            then thisIsMax else resultSoFar;
17.      }
18.      function finalResult makeFinal(innerResult lastResult)
19.         { return lastResult.index; }
20.   };
```

Lines 2 and 3 define constant values specific to this application – the number of bits in the signed input value, and the highest index value (starting from 0) to be used. These are not visible to parts of the system defined in terms of class `Reduction`. They could also have been expressions based on values defined elsewhere. Line 4 defines the input value to be a signed integer of some number of bits. Line 4 states that the input data value is an integer with specified number of bits. Line 5 says that the final result is an unsigned value with enough bits to hold the `maxIndex` value. Although the final result is an index value only, temporary results throughout the operation need to carry both the highest value to date and the index at which it occurs. Lines 8-9 create the constant that initializes the reduction: its index is irrelevant and set to zero, and its value is the most negative possible value. Although this uses a `new` operator like that in Java or C++, it does not result in any run-time allocation of memory. This executes at compile time, so it reduces to a set of constant bit values by the time it reaches synthesis. The function in lines 10-17 starts by creating the result that would needed if the current value were to be the highest to date in the reduction. Lines 15-16 decide whether this new value should be returned to the caller, or the previous best. Finally, lines 18-19 strip off the information needed for finding the highest value and return only the position at which it occurred.

Clearly, `MaxPos` is only one of many possible concretions of abstract class `Reduction`. A sum over the inputs might have used an integer value for `finalResult`, and the same value for `InnerResult`. The `makeFinal` function, in that case, would need to be present for architectural reasons, but would just pass the input directly to the output.

Assume that the system consists of (at least) two component types: `Level1` and a component that it instantiates called `Level2`. The `Level1` component is the top-level component instantiated by the system. In this case, the application as a whole might include the following files.

```
1. entity Level2 {
2.    import class AppSpecifics extends Reduction;
3.    . . .
4. }
```

Although class names and file names have no fixed relationship, the LAMPML form of this file would normally be named Level2.xml. This component makes use of class definitions using a symbol named `AppSpecifics`. It is known only that `AppSpecifics` exports all the symbols definitions found in class `Reduction`, defined above. All interface items are `Reduction` are abstract, however, so `AppSpecifics` can not actually be bound to `Reduction`. Class `Reduction` just defines the set of interface items accessible through symbol `AppSpecifics`.

```
1. entity Level1 {
2.    import class AppSpecifics extends Reduction;
3.    . . .
4.    instance Level2 innerL2 {
5.       class AppSpecifics := AppSpecifics };
6.    . . .
7. }
```

The file Level1.xml is assumed to have the text above as its content. As in Level2, symbol `AppSpecifics` represents some class, not yet identified, which exports the interface defined by class `Reduction`. Entities `Level1` and `Level2` are different symbol definition scopes. As a result, definitions of symbol `AppSpecifics` in the two have no relationship to each other. Choice of the same name for symbols in the two different entities was arbitrary, but convenient for the developer.

Line 4 in `Level1` defines `innerL2` to be a logical instance of entity `Level2`. The symbol binding at line 5 refers to a symbol defined in `Level2`, on the left of the assignment, and a different symbol on the right of the assignment, defined in `Level1`. It means that class symbol AppSpecifics in innerL2 actually refers to the same class that AppSpecifics in Level2 refers to. At this point, however, the actual class bound to symbol AppSpecifics has not yet been determined, so innerL2's class binding is not known either.

The last file in this example contains the application definition that follows:

```
1. application TestApp {
2.    useClass “Reduction.xml”;
3.    useClass “MaxPos.xml”;
4.    use “Level1.xml”;
5.    use “Level2.xml”;
6.    root Level1 appRoot {
7.       class AppSpecifics := MaxPos }
8. }
```

The actual file would be XML derived from this CLAMP text. Lines 2-3 state the names of the class definition files used by this application. Lines 4-5 state the names of files containing annotated HDL code. Line 6 states that the root component is an instance of Level1. The logical instance of the root component is called appRoot.

The binding of line 7 states that the symbol AppSpecifics in logical instance Level1 shall refer to class MaxPos. This is valid, because AppSpecifics must name a class that is a subclass of Reduction, and MaxPos is such a subclass. Because of the binding at line 5 in entity `Level1`, symbol `AppSpecifics` in `innerL2` also refers to `MaxPos`. Symbol binding is a recursive process, and propagates to any number of levels of structural hierarchy.

5.5.3 LAMP to HDL data conversion

Some synthesis tools require `std_logic` bits and bit strings for specific constructs to synthesize properly. For example, Xilinx recommends that only those standard data types be used in synthesizable code [Xil03b], and inference of block RAMs fails for any stored data types other than `std_logic` and `std_logic_vector`. LAMPML code, however, encourages use of tuple data types. Although LAMP's tuple types can be represented exactly as VHDL `record` types, such values cause problems where `std_logic` is required.

LAMPML supports the `std_logic` and `std_logic_vector` types directly. The LAMP tools also provide two function for every data type, *typeName*`:fromStdVec` and *typeName*`:toStdVec`. LAMP automatically provides these conversions to and from every data type. All conversion functions have the same names, so scoping is required to disambiguate them. The `a:x` LAMPML syntax specifies the instance of `x` that is scoped within `a`, so that `x` is distinct from `b:x`, `c:x`, etc. The functions implicitly have the following LAMPML signatures:

```
function std_logic_vector[synthSize(typeName)]
            toStdLogic(typeName)
```

and

```
function typeName
          fromStdLogic(std_logic_vector[synthSize(typeName)])
```

As shown, the `synthSize` (described below) function works with these functions to define VHDL data items of sizes that match the LAMP types.

These functions work the same way as any other, when being converted to synthesizable form. They respectively generate auxiliary VHDL functions named `toStdLogic_nnnnnn` and `fromStdLogic_nnnnnn`, where `nnnnnn` represents some unpredictable integer value. As many auxiliary functions are generated for a tuple type conversion as are needed to handle the types of each element of the tuple.

They type name that scopes `toStdLogic` or `fromStdLogic` may be a name in a LAMP entity's list of type imports. In that case the exact identity of the conversion function is not known until a logical instance of the entity is created by binding actual type definitions to the import symbols. Different actual types could be bound to a given import symbol in different logical instances. In that case, each logical instance has different conversion functions, according to the actual types bound to the import symbols.

5.6 Local and global definitions

Section 5.4 described how VHDL entities with different component interfaces are generated when symbol bindings create logical instances. Section 5.5 described how HDL code generation works for LAMP library functions and for LAMP-defined data conversion functions. Additional VHDL definitions are created in generating VHDL code for LAMP data types and symbolic constants. Some of these definitions must be available globally, across all HDL code in an application. Other definitions are local, of interest only within one of the application's compilation units.

Either way, VHDL demands that definitions of different types be placed at different syntactic locations in the source files involved. LAMP does not perform syntactic analysis on the HDL files, so it has no direct way to determin where the generated code is syntactically allowed. Three LAMPML elements are involved in correct placement of the definitions: `writeHdr`, `insertHdr`, and `insertLocal`. These elements appear only within LAMPML `entityDef` elements, the elements that corresponds to the VHDL `entity` definitions for synthesizable component.

5.6.1 Local definitions: the `insertLocal` element

The `insertLocal` component normally appears exactly once. It has no attributes, and does not contain any other elements or HDL text. It identifies the point in the VHDL `architecture` definition at which the local symbol definitions can be inserted, which must be between the `architecture-is` part of the entity definition and the `begin` statement that encloses the synthesizable logic of the component, and not within an incomplete VHDL declaration. Generated VHDL code has this element replaced by the set of automatically generated local symbol definitions. In the even that one `entityDef` element contains multiple VHDL `architecture` blocks, each one must have its own `insertLocal`.

5.6.2 Global definitions: the `insertHdr` element

The `insertHdr` element appears only once per HDL file, among the VHDL `use` statements that import library definitions. This creates a reference to a header file created by the LAMP tools, and used to share definitions required by multiple VHDL entities. The name of this file is not under programmer control.

5.6.3 Creating globals: the `writeHdr` element

The `writeHdr` element may be used as many times as desired within an `entityDef` element. The raw text and LAMP expressions in it are written to the global definition file referenced by `insertHdr`, and are not inserted into the HDL file for the current entity. Any expressions are evaluated before being written to the file. The main use of this feature is in creating new, globally visible VHDL `component` declarations for each LAMP logical instance. For example, consider the following LAMPML code fragment:

```
1. <entityDef name="EntExample">
2.    <symExport name="entName" type="symbol" >
3.       <uniqueID prefix="conv3D_"/>
4.    </symExport>
5.    <typeImport name="portData" type="baseType"/>
6.    entity <varRef name="entName" /> is
7.       port(datIn:      in <symRef name="portData" />;
8.          datOut:       out <symRef name="portData" />);
```

See Appendix A for full descriptions of the XML elements and attributes used in this example. Line 1 opens the definition of a new entity the same way the CLAMP `entity` declaration does. The next lines, 2-4, create a symbol to be used for the entity name. This can not be an HDL symbol, because LAMP could create multiple logical instances of this entity with different bindings. Each logical instance must have its own name, so that HDL instantiation can refer to them differently. Because this uses `uniqueID` to create a symbol, it is guaranteed to have a different and unique value in every logical instance of the entity. Line 5 imports some data type for use by the component.

Lines 6-8 are annotated VHDL code that starts the VHDL entity definition. Line 6 uses the `entName` symbol defined at line 2, rather than using some specific VHDL symbol for the entity name. The actual VHDL entity name is not predictable, because `uniqueID` does not guarantee any particular symbol as a result. The `entName` symbol is an export from this LAMP entity, however, so any client of this logical instance can use the actual VHDL entity name for creating VHDL instances of this component, no matter what is assigned by `uniqueID`. Lines 7 and 8 define the two ports for this component, both of some type defined by the `portData` import at line 5. This creates a new VHDL component with port parameter types that could not be predicted by the VHDL developer. If multiple logical instances of this element are created, then multiple VHDL components with different port types are created, where each requires its own unique entity name. Note that this is a significant extension to basic VHDL semantics, because there are no VHDL language constructs able to perform data type substitutions like this.

The VHDL developer can go on to create the rest of the VHDL architecture using any desired VHDL constructs, and using expressions based on the `portData` data type. The problem is that the global definition files, built around VHDL `package` statements, can not create `component` references to this VHDL entity. It was not known at the time this entity was defined what port data types would be used, or even how many different logical instances would be created. Because of the `uniqueID` entity name, even the client components that instantiate this one could not predict the entity name, so could not create their own `component` declarations. That would have been bad programming practice in any case. It would have distributed the declaration of the one logical instance across multiple different files. Deriving all definitions from one place reduces the possibility of "version skew" that occurs when the component definition changes, but the change is not propagated to all points at which it occurs.

Instead the VHDL developer should use this `entityDef` declaration to make visible all of the logical instances that derive from it. The following code continues the example above, in the same LAMPML source file:

```
<writeHdr>
      component <varRef name="entName" /> is
         port(datIn: in <symRef name="portData" />;
            datOut:  out <symRef name="portData" />);
      end component <varRef name="entName" />;
   </writeHdr>
```

The `writeHdr` element has no effect on the local file – it and its contents are removed when the HDL file for the logical instance is generated. It does, however, generate output in the global definition file referenced by the `insertHdr` element, once per logical instance of this LAMP entity. In addition, it generates a globally visible definition for type `portData`, if necessary, and for any other types on which `portData` depends.

5.7 Synthesis estimation and sizing

Synthesis estimation depends on three sources of information: the logic designer's understanding of the HDL for the reusable parts of the application accelerator, the application specific details of leaf functions and data types, and the FPGA-specific pool of resources and allocation strategies. This section defines the basic tools and techniques of synthesis estimation and sizing, and Appendix B includes a detailed example.

5.7.1 FPGA specifics

FPGA-specific features are handled within the LAMP language. The logic designer is required to determine what FPGA resources are important to the application, to state the limits of each resource, and to provide any other FPGA-specific knowledge required for reasonably accurate synthesis estimation. Code sample 8 provides one example of a generic FPGA description, and Code sample 9 gives a specialization of that interface, to represent the actual amounts of rersources in a particular model of Xilinx Virtex II Pro. Both of these examples use the CLAMP language, defined in Appendix A.3.

```
class FPGAresource {
   const natural logicCells;
   const natural ramBlocks;
   const natural hardMultiply;
   function natural nRams(natural wordSize, natural nWords);
}
```

Code sample 8: Abstract definitions of FPGA resources

This interface definition describes any FPGA in terms of three constant values and a function, all of which must be given concrete meanings for any particular FPGA. The abstract definition – without actual values for the constants and with an actual strategy for allocating physical RAMs to logical structures – gives a set of symbols to use in FPGA-specific allocation. By itself, however, it can not be used for allocation logic. All of its interface items must be given actual definitions for this to be useable.

```
class XC2VP70 extends FPGAresource {
   const natural logicCells := 66176;
   const natural ramBlocks := 328;
   const natural hardMultiply := 328;
   const natural ramBits := 16*1024;
   const natural minWords := 512;
   function natural nRams(natural wordSize, natural nWords) {
      natural effWords := if nWords > minWords
         then nWords else minWords;
         return (wordSize * effWords + ramBits-1) / ramBits
      };
}
```

Code sample 9. Chip-specific resource description

This provides implementations for all of the interface elements in Code sample 8, but also defines some values for its own use (`ramBits` and `minWords`).

5.7.2 Application-specific estimation

The LAMP tools define a function that reports the number of bits needed for a value of given type, or amount of logic for a function defined in the LAMPML language. This allows the developer to create synthesis estimation functions that do not depend on any particular choice of application-specific functions or data types.

The first form of the `synthSize` element accepts the name of a data type, defined by some `typeDef` element. In CLAMP notation, this appears as a function that has a type name as its input parameter, rather than a data value. When applied to a data type, the synthSize expression returns an integer value that specifies the number of bit in a value of that data type.

The second form of the `synthSize` expression accepts a user-defined function name as input. Polymorphic functions synthesize differently according to their parameter types, so this use of `synthSize` requires the names of the actual data types corresponding to the values each of its parameters. As an example,

```
synthSize(func, ptype1, ptype2)
```

performs estimation on function `func`. That function has two parameters. This expression considers the case where the first actual parameter to the function call has data type `ptype1`, which must be a subclass of `func`'s first formal parameter, and the second actual parameter type is `ptype2`, which must be a subclass of `func`'s second formal parameter.

LAMP's primitives are assumed to synthesize only into logic elements. As a result, no estimation is made for the number of block RAMs, dedicated multipliers, or other FPGA resources.

Precautions in using `synthSize`

Synthesis estimation can never be 100% precise. The amount of logic generated for a given expression depends on a large number of factors, not all of which are visible to the LAMP tools. Even the choice the synthesis tool's control options can affect the amount of logic generated. As a result, the user should be aware of the algorithms used by `synthSize`, including likely sources of estimation error.

If the `synthSize` element's `name` attribute refers to a data type, the expression returns the number of bits needed for allocating one value of this data type. This over-estimates the actual bit allocation if some or all of the value is constant or is never used. If any of the value's bits are constant, synthesis often performs optimizations that combine the constant bits with other logic, eliminating those from the executable logic design. If some of the bits are never read or, transitively, are inputs only to bits that are never read, then the synthesizer need not allocate resources for them.

When the `name` attribute of a LAMPML `synthSize` element specifies a function, the value is the number of logic elements estimated for one instance of the function logic. This is generally taken to be one logic unit per operation per bit of output. For example, logical OR of two six-bit values is assumed to synthesize into six logic elements. Some operations, such as bit concatenation and bit field extraction, do not allocate any logic. Multiplication and variable-amount shift operations are assumed to require $n_1 \times n_2$ logic units, where n_1 and n_2 are the numbers of bits in the operands.

Expressions involving HDL constants and expressions where the value is never used are likely to be simplified or eliminated in synthesis. That can not be forseen by the `synthSize` function, and tedns to cause over-estimation of logic utilization. Expressions where multiple operations may be combined in one unit of hardware logic may also be estimated as claiming more resources than in fact are allocated. For example, the bit expression *(A AND B) XOR (C OR D)* consists of three bit operations so would be estimated at three logic elements. In fact, this can be implemented in one of a Xilinx FPGA's four-input LUTs. If synthesis allocates a block multiplier in place of a discrete implementation of multiplication, or if a function is implemented as a lookup table in block RAM, then this over-estimates logic allocation and does not account for the resources actually allocated.

In some cases, `synthSize` may underestimate the amount of logic allocated to a function's implementation, or to the hardware allocated to a data element. This can happen, for example, when fanout or signal distribution issues lead the synthesis tools to duplicate logic elements, typically registers, in a way not specified by the logic design. It may also happen when synthesis implements a single logical memory using multiple block RAMs, and implicitly adds selection logic that lets the collection of RAMs act like a single memory.

The `synthSize` logic assumes that logic utilization is strictly additive. For example, if function *P(x)* is *Q(x)* + *R(x)*, `synthSize` estimates this by summing the estimates for *Q*, *R*, and the addition. Many tools would flatten P during synthesis, exposing all of the logic of Q and R, somewhat the way an optimizing compiler may inline HLL functions. This creates additional opportunities for global optimization, for example by eliminating sub-expressions common to the bodies of Q and R. As a result, the additivity assumption may overestimate the logic needed for P.

Experiments to date have not found cases where estimation errors affected the validity of array sizing. Still, this is an open area for further exploration. Although the fact of synthesis estimation is central to the LAMP tools, the techniques for performing it are readily modified. Estimation improvements should be easy to implement without fundamental changes to the LAMP tools.

5.7.3 Designer's estimation of HDL code

Because LAMP does not analyze the HDL code for an application accelerator, the developer must provide synthesis estimation. There is no fixed interface to which synthesis estimation functions must conform. The designer may use any strategy desired for performing synthesis estimation.

In order to make synthesis estimation reusable, and to limit difficulty in maintaining the estimation functions, developers should put the estimation function for each HDL component in the LAMPML source file for that component. When the component is reused in another application, its synthesis estimates can be reused with it. When one component contains instances of another, estimation of the outer component's resource usage should be made in terms of calls to the inner component's estimation functions.

Synthesis estimation should not assume any particular set of import parameter values for a component. In particular, when a component is parameterized to generate selectable numbers of inner design units, the estimation function should accept that number as a parameter input. That allows the estimation function to be used for "what-if" explorations of different configurations.

5.7.4 System sizing

The reason for performing synthesis estimation is to support system sizing, i.e. making decisions about the actual sizes to use for the repetitive structures in the application accelerator. Ideally, the amount of logic resources required, x, and the structural parameter n would be related by some invertible function $R(n) = x$. Then for a given amount x' of logic resources, the structural parameter n' would simply be $R^{-1}(x') = n'$.

The examples and discussion of section B.1, however, show that realistic systems do not necessarily have simple or invertible expressions for resource usage. Most credible applications will have only a few structural parameters, however, and a modest range of valid values for the parameters. Simply trying different parameter settings, using the assumptions stated in Appendix B's Equation 4, is assumed to be an affordable way to determine the solution to that optimization problem. Appendix C gives a detailed example in LAMP notation.

6 RESULTS, CONTRIBUTIONS, AND FUTURE DIRECTIONS

The first result of this research is that BCB applications of many kinds are infact amenable to FPGA acceleration. They demonstrate speedups of 100-1000×, relative to a PC, for applications having widely differing patterns of communication and synchronization. Since these problems have scientific and commercial interest in themselves, that kind of speedup shows that FPGAs can have significant value as application accelerators. Sections 6.1.1 and 6.1.2 present quantitiave performance measurements, and show the importance of scaling each computing array to the largest size possible. Section 6.1.3 notes the relative ease with which customizations can be made using the prototype LAMP tools.

Section 6.2 summarizes the contrinutions of the current research in several areas. It describes the BCB applications themselves, and observes that applications in other areas show many of the same features that made FPGA acceleration work so well in these cases. It describes novel features of the LAMP tools that distinguish it from traditional logic design tools and that enable development of application accelerators. Finally, this section describes the component reusability concepts presented in the case studies. Some, like reuse of control and communication, are distinctive features of LAMP; others demonstrate kinds of reusability that could have been implemented with other tools, but have not been noted elsewhere.

Finally, section 6.3 notes areas for future exploration. Additional case studies, of course, would serve many purposes, including usability testing and suggestions for future enhancements. Many additional enhancements to the tools are forseen, including improvements to the user interface and in integration with the logic design tools on which synthesis depends.

6.1 Results

The initial performance results of this research come from the hand-coded case studies, described in Chapter 2. They are summarized briefly, and their significance discussed. The LAMP tools are also evaluated, showing how they adapt computing arrays to available resources, according to the resource demands of individual applications and resource pools of specific FPGAs.

These case studies demonstrate the FPGA's potential for hundred-fold application acceleration, or more. Beyond that simple fact, these figures demonstrate the value of application-aware customization. Table 2 shows that customizing the accelerator to application specifics can mean roughly a 2:1 increase in the number of PEs for a given FGPA platform. That directly means a 2:1 increase in parallelism, and in the sizes of problems that can be addressed directly. Because PEs for different applications differ in circuit complexity, they also differ in maximum allowable clock rate, also by a factor around 2:1. Combining the two effects, this demonstrates a 4:1 performance improvement due to application awareness.

Table 3 in section 6.1.2 shows the same benefit of application-specific customization, and shows it more strongly. These application-specific accelerators show a 7:1 performance range. This implies a 7:1 performance improvement for the simplest cases, relative to handling the same logic with more general hardware.

Table 2 . Approximate String Matching Performance

Match Cell	Character Rule	String Type	Logic (slices)	Clock (ns)	Cells in 2VP30	Speed GCUP/s	Speedup
NW	3GHz Xeon PC implementation					0.046	
NW	Exact match	DNA	109	12.9	125	9.68	210
NW	IUPAC wildcard	DNA	108	13.7	126	9.19	200
NW	Fixed table	DNA	111	14.6	123	8.42	183
NW	RAM table	DNA	108	16.8	126	7.50	163
SW	3GHz Xeon PC implementation					0.029	
SW	Exact match	DNA	190	13.3	72	5.41	186
SW	Fixed table	DNA	193	15.9	70	4.40	152
SW	Exact match	protein	205	13.0	66	5.07	175
SW	Fixed table	protein	239	25.5	57	2.23	77

6.1.1 Approximate String Matching

Table 2 [Van06] summarizes the performance results for the string matching family of applications, based on implementation for a Xilinx XC2VP30 FGPA. The leftmost column states whether the test used a Needleman-Wunsch (NW) global alignment or the more complicated Smith-Waterman recurrence relation. Speed is measured in 10^9 character updates per second (GCUP/s), where each character update is one computation in one of the grid cells of Figure 7. The 77-210× acceleration, relative to a 3GHz processor, is worthwhile in itself, especially given the relatively small (VP30) member of the Xilinx Virtex II Pro family used. The real value of this application, however, is in showing the different resource utilization and clock rate for each member of the application family. The number of cells (degree of parallelism) varies by more than 2:1 over the range of application family members, and the clock rate by almost as much. The result is that the performance of the demonstrated family members covers a 4.3:1 range. Although the "fixed table/protein" implementation could have been used to perform the "exact match/DNA" operation, given with proper constants, precise tuning to the exact/DNA case makes much better use of the FPGA resources. This clearly demonstrates the value of tuning an accelerator the application specifics.

This also demonstrated the value of swappable components in an accelerator – the NW/SW choice of recurrence relations is orthogonal to the "character rule" and "string type" of Table 2. The actual implementation included end-gap options and idiosyncratic parameters for several different substitution matrices (after Nei00 p.35) that can easily be expressed in LAMP notation, but poorly and indirectly in VHDL.

6.1.2 Rigid Molecule Interaction

Table 3 [Van06a] summarizes performance results for the convolution core for rigid molecule interactions. Table 3 uses score-accumulates per second (SAC/s) as the performance measure. This corresponds directly to measurements of multiply-accumulate operations per second (MAC/s) measure that describe performance of standard correlation, but replaces multiplication with the general scoring function of Equation 2.

Molecule voxel values range in size from 2-7 bits per value, and scoring functions also differ in complexity. The current study constrains the correlation array to cubical aspect ratio, although non-cubical computation arrays may be of interest in some applications. The ACP force law uses a logic-based implementation of a lookup table. It requires a relatively large amount of logic per cell, since concurrent execution requires each cell to have its own instance of the table. The SNORM law uses a simplified surface normal vector, and uses the dot product to determine surface normal opposition.

Table 3. Correlation array performance by force law

Force law	Bits per voxel	Correlation array size		Clock MHz	Performance 10^9 SAC/s
KK[1]	2	14^3 =	2744	98.9	271.4
PSC[2]	7	12^3 =	1728	88.3	152.6
GSC[2]	2	10^3 =	1000	59.8	59.8
ACP[3]	5	8^3 =	512	72.6	37.2
ES[4]	6	10^3 =	1000	72.8	72.8
SNORM[5]	7	11^3 =	1331	90.4	120.3

Notes: 1. Similar to Katzir-Katchalski [Kat93]. 2. Described in [Che02]. 3. Simplified atomic contact potentials. 4. Electrostatic force and collision detection. 5. Surface normals and collision detection [Hal02]

Performance varies even more widely for this application than for approximate string matching, over a 7.3:1 range of values. Table 4 (after [Van04b]) compares performance of a different configuration, corresponding to the KK force law on a VP100 platform.

This assumes transform-based correlation on the PC, as it would usually be performed; timing is for one transform-inverse pair only. This test used the 3D transform and inverse from a standard reference [Pre92], which required padding both molecule grids to 128^3 or 256^3 (the sizes needed to handle worst-case rotations). The two tests use molecule grids of size 66^3 and 100^3 for the larger molecule, and 14^3 for the smaller in both cases. The "average" FPGA result comes from averaging FGPA performance over all 3-axis rotations, which differ in the amount of padding required to hold the rotated images of their bounding boxes.

Table 4. FPGA Performance vs PC implementation

Result size	FPGA (ms)	3GHz Xeon (ms)	FPGA speedup
66*14 (avg)	12.2	4250	209×
100*14 (avg)	73.5	36840	501×

In combining this result with Table 3, it must be remembered that PSC, GSC, and ES would each have required two transforms and an inverse, SNORM would have required four transforms and an inverse, and ACP sums over a nonlinear function that can not be handled at all using transform techniques.

6.1.3 Programmability

The proper measure of programmability would involve testing with volunteers, both in creating new LAMP models and in customizing models to new applications. The crude state of the initial LAMP tools would tend to give results that would not correctly represent a releasable form of the tools, however, and such testing lay outside the range of the current research.

Instead, the size of each application concretion, measured in lines of CLAMP code, is taken to indicate the amount of effort in customizing the accelerator to a specific application. For the rigid molecule interaction application application-specific voxel data type declarations and scoring functions totalled 24 (for KK) to 117 (for ACP) raw source lines, not stripped of blanks or comments. These small number suggest that even relatively complex customizations can be written in a modest amount of high level code.

6.2 Summary of contributions

This research makes contributions in several areas of FPGA-based acceleration of BCB applications. The first is that BCB applications of many different kinds are all amenable to FPGA acceleration. It seems likely that other scientifically and commercially important application areas would also benefit, but were outside the scope of the current work. The second area of contributions lies in the design and implementation of design tools for FPGA accelerators. It now seems clear that accelerator design is an activity distinct from traditional logic design, and that many features of logic design tools need to be reconsidered for the emerginf field of FPGA-based computation. The third area of contribution includes not only novel, reusable components, but novel reuse of existing components and novel categories of components for reuse.

6.2.1 BCB application acceleration

When the current research began, FPGA accelerators had been reported for specific, hand-crafted forms of the string alignment problems, but very few other BCB applications. Based on existing literature, it seemed that few other areas were amenable to FPGA acceleration. It also appeared that each application would be a unique artifact, and that reuse even between closely related applications would be more of a happy accident than a goal of the design or the design tools.

The case studies of chapter 2 decisively change that impression. In the first place, many computations of widely different structures have been shown to benefit from FPGA acceleration. Brief studies suggest that many more applications will also benefit, including point-point, point-grid, 3D interpolation, and new kinds of string analysis, with additional opportunities in areaws outside the BCB applications of this study. In the second place, it is clear that application families are attractive targets for acceleration, not just point solutions to special cases of algorithms.

6.2.2 LAMP tool implementation

The prototype LAMP tools demonstrate the concept that design of FPGA-based application accelerators is very different from traditional logic design. A number of distinctive features combine to give LAMP tools unique expressive power, including:

· **Support for families of applications.** The initial case studies lay groundwork that show two major features of FPGA-based application acceleration. First, application specialists use many variations on the basic algorithms being accelerated. Point solutions have profoundly limited value. Second is that hardware design skills are generally needed for exploiting the FPGA's full performance potential, but such skills are relatively rare. Customizable families of applications make it possible to amortize logic design costs over many uses, but still support the unique features needed by different application specialists.

· **Separation of logic design from application-specific customization**. Logic designers and application specialists must both participate in accelerator design, in many applications. LAMP tools allow each participant a reasonably familiar design notation, while making the application specialist independent of the logic designer for routine customization tasks.

· **Automated sizing of computation arrays.** Target applications use repetitive arrays of processing elements, and achieve high performance through parallelism of the PEs. LAMP tools enable automated exploitation of the highest possible parallelism for each unique application accelerator, on a given FPGA. This takes advantage of the FGPA's fine-grained resource allocation, and automatically exploits the additional resources of larger FPGAs. No other logic design tools are known to support automated selection of the application's degree of parallelism in this application-specific and FPGA-specific way. Formalisms are presented for describing the different factors that go into deciding the degree of parallelism to use in any one instance, based on the general character of the application family, the specifics of the application family member, and the amounts of the various logic resources available in a particular FPGA.

· **Synthesis of object oriented accelerator descriptions.** Although SystemC and System Verilog are both OO languages, they do not include OO features in their synthesizable subsets. Although other experimental OO synthesis systems have been demonstrated, LAMP uses a unique combination of OO semantics to support close integration of high- and low-level system component specifications.

· **Parameterization of data types and functions.** OO system descriptions specify, in abstract form, the specifics that distinguish the different members of an application family. Interface inheritance provides a natural and familiar way for application specialists to provide the concrete details that specify their particular applications. Strong typing, enforced by interface inheritance rules, helps ensure proper implementation of application family members. This level of parameterization is impossible for standard logic design tools.

· **Inverted flow of control.** Parallelism, communication, synchronization, and memory access patterns commonly dominate the difficulty of FPGA design, and also dominate the performance of FPGA accelerators. Traditional design based on IP blocks assumes that system-building is the task of creating communication networks between leaf blocks – i.e., supplying the difficult, performance-critical part of the design. LAMP tools assume that communication and parallelism are the reusable design components, and contain customizable leaf blocks. This corresponds closely to the "inverted flow of control" in GUI systems or component systems like Java's Enterprise Beans, where the complexity of event processing and data access lies outside the application, and the control system invokes application-specific logic when data becomes available.

· **Generality of applications.** Many FPGA design tools implement a particular class of application accelerator. Examples include systems for 2D image processing [Rin01] or streaming data [Ags95, Men06]. LAMP tools address any application with reusable communication structure, and especially with variable-sized arrays of processing elements.

Subsets of these features have appeared in other logic design tools. The combination of all of these features is unique to the LAMP tools, however. They distinguish LAMP from other logic design tools as a tool set for creating application accelerators.

6.2.3 Reusability and reusable components

Future development of the LAMP tools will rely, in part, on the availability of reusable structures for high-performance computing. Case studies to date have contributed in three areas: novel computing structures for reuse, novel uses of known computing structures, and novel kinds of reusability for hardware components.

The case studies developed several new computing structures, including the combinatorial data distribution network described in section 2.1.1 and the analysis network of section 2.1.5. Other structures from these case studies may also be new, at least to some readers. Yet others, such as application-specific memory interleaving, are also under development and may be expected to emerge from future case studies.

The second kind of reuse extends widely accepted current practice. It appears, for example, in the rotated addressing and correlation arrays of section 2.1.3. Rotated indexing is well known in computer graphics applications. Two-dimensional convolution and correlation arrays have been well studied and widely reported in the signal processing literature, and those arrays can readily be extended to three dimensions. The novelty lies not just in their combination, but in their generalization to complex data types and in their applicability to chemistry applications far from their original uses.

Finally, these examples employ reusability in terms of function and data type parameters, which are not widespread in the hardware world. That kind of reusability creates the possibility of reusing control and communication components in ways that current HDLs support poorly if at all. For example, the filter memory of section 2.1.3 may not be wholly novel, although it was developed independently for the docking case study. Reuse of that component depends, in part, on the the novelty of reuse by redefinition of the summarizing function. Likewise, the 3D correlation array used by the same application has been reported elsewhere. Its novelty lies not just in generalizing the correlation function, but in parameterizing that correlation structure in terms of inner components. The filter, in order to run at streaming rates, is a non-trivial control structure. The correlation array is largely a communication and synchronization structure. Both violate the general rule that function is reusable but control and communication are not, and violate the rule that reusable components are leaf blocks without accessible internal structure.

These case studies offer not just reusable components, but new kinds of reuse and new possibilities for reuse of existing intellectual property. Reusable components multiply the value of their development effort by the number of ways they are used, but new kinds of reusability multiply the value of many different components.

6.3 Future directions

The LAMP tools uniquely address FPGA-based accelerator design as an activity distinct from traditional logic design. They explore new ways of combining the different skills of the logic designer and application specialist, without requiring the logic designer's expertise for routine customizations to the accelerator logic. They represent only an initial investigation, however, and can be extended in many directions, including new case studies, user interface enhancements, and semantic extensions.

Case studies

New case studies will exercise LAMP's basic features more exhaustively. More importantly, they will also suggest new features to increase the usability or semantic capability of the tools. Additional data will also be helpful for increasing the accuracy of synthesis estimation.

Case studies based on OO techniques will have additional value as examples of OO hardware design. Many OO HDLs have been reported in the past, typically with code samples a few lines long. These barely suffice to show the features of the language; few, if any reported examples have been large enough to demonstrate the real value of OO design. Additional case studies will have value for the applications they implement, and also serve as examples for demonstrating OO techniques to other logic designers.

Tool extensions

LAMP's current UI uses text input only. Although CLAMP can be used in some parts of the LAMP model, the annotated HDL must still be hand-edited, so annotation is an important target for new tool development. A GUI would also be helpful for hiding some of the organizational details of data stored for each LAMP model, and for guiding application specialists in tailoring LAMP models to specific applications. Error reporting can be also be enhanced, to state the cause of compilation errors more user-oriented terms.

Semantic extensions to LAMPML will be suggested by additional experience with the tools, and by enhancements to the tool suite. LAMPML features, to date, have centered on the synthesizable accelerator code. LAMP UI tools may require new kinds of data in the model, including informational messages about model-specific features and selection menus displaying alternative components.

Other major areas of extension would allow the application specialist more of the design freedom now reserved to the logic designer. "Drag and drop" design, for the system's general structure, would support a reasonably familiar style of interface while hiding implementation details from the user. Experience suggests that pervasive use of visual metaphors becomes unwieldy at finer levels of detail. Textual representations of data definitions and arithmetic expressions are far more concise, and easily understood by the large base of experience software developers.

Appendix A. LAMPML Reference Manual

Many points in the LAMPML definition require some type to be a subclass or superclass of another. Loosely speaking, this means sub- or superclass at any number of steps of separation. More formally, both sub- and super-classing define transitive relationships. If type B is an *immediate subclass* of A, it means that type B's definition refers specifically to A as its superclass. Type A is considered to be a subclass of itself, and any immediate subclass of a subclass of a subclass is also a subclass, recursively. The definition of the superclass relationship has the same general form. For reasons of notational convenience, rules for well-formed subclasses are presented in section A.3.

Compile time vs. run time

LAMP processing evaluates as many expressions as possible during the compilation step, before handing the application over for synthesis. The following expressions are considered to be compile time constant expressions (CTCEs): any decimal or hexadecimal constant, a boolean constant `true` or `false`, a typedef instance built from CTCEs, an operator expression involving only CTCEs, a function call in which all parameters are CTCEs, or any variable, field, or const symbol defined in terms of CTCEs. Other expressions, such as a multiplication by zero, may also be CTCEs, but those cases are not defined as part of LAMP's guaranteed behavior.

Scoping and qualified names

A LAMPML symbol is any contiguous string of characters, starting with an English letter or underscore character '_', followed by zero or more English letters, digits, and underscore characters. Symbols are case sensitive, i.e. switching a letter form upper to lower case creates reference to a difference symbol. Symbol resolution always starts in the current scope: function, class, application, or entitydef. If the symbol is not found in the current scope, then superclass scopes are searched, and finally the scope containing the current scope (e.g. application surrounding a function definition).

In many cases, those rules are not adequate for accessing a desired symbol. In those cases, scoped names may be used. A scoped name is a sequence of one or more symbols, separated by colon ':' characters. The scoped name is interpreted recursively, by taking the symbols one at a time, in left to right order. Resolution starts in the current scope, and determines the scope in which the leading symbol is resolved. That scope becomes the scope in which the remained of the scoped name is resolved.

Scoped names are used for extracting fields from a symbol to which a typedef instance has been assigned. Scope resolution proceeds as usual, until a function parameter, const, other symbol is found that holds a value. Then, the typedef (including its supertypes) becomes the scope in which name resolution is performed. This also operates recursively, in the case of a field holding a typedef value.

A.1 LAMPML markup

LAMPML is the LAMP Markup Language, an XML-based notation. It has three logical subsets, for model definition, HDL annotation, and top-level integration. These subsets overlap in many areas, especially handling of arithmetic expressions.

In XML terminology, an *element* is the basic unit of composition. Since an element may enclose other elements or text, it consists of starting and ending markers, `<elemName>` and `</elemName>` respectively. Different element types replace the logical `elemName` with some case-sensitive symbol unique to that element type. The start marker always appears before the end marker. If one element is nested inside another, its start and end markers must both be inside the start and end markers of the containing element. If two elements are both contained in some other, at the same nesting level, then the start and end markers of the first must precede the start and end markers of the second. One element type may be repeated within another. In general, the order in which markers appear is important, but specific elements may not assign significance to the order. If a marker does not contain any others and does not contain text, the start and end markers may be combined into one: `<elemName />`.

Elements may have *attributes*, each with its own name. Attributes appear inside of the start marker or combined start/end marker, after the element name and separated by white space. Each attribute is written as a symbol name, equals sign and attribute value enclosed in double-quotes. Each attribute is specific to one or more element types. An element may have zero or more attributes separated by white space, but no attribute may appear more than once in any instance of an element type. New lines within the marker have no significance. Order of the attributes within the element marker is not significant. An empty element with two attributes `attr1` and `attr2` may be written as `<elemName attr1="some Value" attr2="other value" />`, but the order of the two attributes has no meaning.

Because HDL code is embedded in XML, a few systematic changes must be made. XML reserves several characters, including '<' less-than, '>' greater-than, '&' ampersand, and apostrophe. Of course, these characters are also used throughout the HDL code. Since the characters themselves can not be used, XML defines replacement strings that take the places of those characters. Those characters must be replaced, respectively, by '`&lt;`', '`&gt;`', '`&amp;`', and '`&apos;`'.

The model definition subset of LAMPML creates definitions of classes, data types, functions, and constants. The interface between the annotated hardware description and the application specialist is expressed in this notation. The application specialist uses this notation to create application-specific definitions for an instance of an accelerator. XML elements in that subset are summarized in the following table.

Table 5: LAMPML elements for model definition

`class`			Abstract or concrete interface definition
	`constant`		Abstract or concrete constant
	`typedef`		Abstract or concrete data type
		`field`	Defines one of the values within a tuple
	`function`		Abstract or concrete function definition
		`param`	Name and type of formal parameter to the function
		`setVar`	Computation variable used within the function
		`returnValue`	Value to be returned from the function

The HDL annotation subset interacts with the HDL definitions for the accelerator family. It inserts LAMPML application logic into HDL code, and extends some of the reusability features of the base language. The following table summarizes the HDL annotation subset of LAMPML.

Table 6: LAMPML elements for HDL markup

`entityDef`			Container element for an annotated compilation unit of HDL.

	`class`
	`function`
	`typedef`
		See Table 4	Symbol definitions for local use

	`entityUse`		Create an instance of another entity
		`bind`	Create a value binding for a `symImport` element in a structurally nested entity.
		`bindType`	Create a type binding to a `typeImport` element in a structurally nested entity.
	`insertHdr`		Inserts a reference to an automatically-generated header file at the current point in the HDL.
	`insertLocal`		Inserts the automatically-generated local symbol definitions at the current point in the HDL.
	`message`	`text` expression[s]	Generates messages at standard output

Table 6: LAMPML elements for HDL markup

`symDef`	Create a computation symbol for use locally
`symExport`	Create a computation symbol with a value that's readable from a containing entity
`symImport`	Create a computation symbol with a value that can be set by a containing entity
`typeImport`	Create a type symbol, such that any containing entity can change the type to which the symbol refers
`writeHdr`	Export text into the shared definition file
`HDL text`	Literal HDL text representing the logic content of the entity

The integration subset indexes the set of files used for an accelerator family, and creates bindings that define a family member. An `application` element may define the application family as a whole, by collecting the HDL components and interface classes together. It can also define a specific instant of an application accelerator, by starting with the family information and adding concrete class definition that implement application-specific data types and functions.

Table 7: LAMPML elements for application integration

`application`			Lists the set of classes and HDL entities in an accelerator or accelerator family.
	`class` `constant` `function` `typedef`	See Table 4	Symbol definitions for local use
	`message`	`text` `expression[s]`	Generates messages at standard output
	`root`		Defines the entity used at the top level of the structural hierarchy
		`bind`	Assigns a value to a `symImport` symbol in the root entity
		`bindType`	Assigns a type to a `typeImport` symbol in the root entity

Table 7: LAMPML elements for application integration

`use`	Specifies a file containing an entity definition
`useClass`	Specifies a file containing a class definition

A.1.1 Model definition

The model definition subset of LAMPML creates the abstract definitions that connect application-specific logic to the HDL code, and allows concrete definitions of the application-specific data types and functions.

Element: `class`

This element defines a class, an interface definition that defines functions, constants, and data types.

Appears in:	Top level `application`		Outermost element in source file that defines a class. Also used in an `application` element to create definitions specific to that `application`.
Attributes:	`name`	required	Symbol name for this class.

	`extends`	optional	Qualified symbol name of the superclass, if present. If no superclass is specified, the superclass is implicitly the special class `baseType`.
Contains:	`constant`	≥ 0	Define a concrete or abstract constant definition as part of this class interface.
	`function`	≥ 0	Define a concrete or abstract function definition as part of this class interface.
	`typedef`	≥ 0	Define a concrete or abstract data type definition as part of this class interface.

Element: `constant`

Defines a symbolic constant. If the constant value is not provided, then the constant definition is considered abstract.

If the constant has the same name as a constant defined in a superclass, then the current definition over-rides that superclass definition. In that case, the declared type of the constant must be a subclass of the type that the superclass used for the constant. The type of the value assigned to the constant (if any) must be a subclass of the constant's declared type.

Appears in:	`class`		
	`application`		
Attributes:	`name`	required	Symbol name, must not already be in use by another field or constant in the current scope, unless this definition over-rides another in the superclass.
	`type`	required	Qualified symbol name for the data type of the named constant. If this definition over-rides a superclass `constant`, then it must be a subclass of the superclass `constant`'s type.
Contains:	expression	0 or 1	If present, this is the constant value. If missing, this defines an abstract constant.

Element: `describe`

Adds descriptive commentary. This element is a comment, and has no effect on the semantics of the containing element. It is allowed anywhere that another element could be used, except at the top level. This is used in all LAMPML subsets. Because it is allowed so pervasively, no special mention is made of it, even when a description of some LAMPML element specifies a list of other elements that it may contain.

Appears in:	Many places	Descriptive text, has no semantic effect.

Contains:	Freeform text	Used for descriptive comments

Element: `field`

Defines one of the fields in a `typedef` tuple definition.

Appears in:	`typedef`		
Attributes:	`name`	required	Symbol name, must not already be in use by another field in this `typedef`, unless this definition over-rides another in the superclass.
	`type`	required	Qualified symbol name for the data type of the field value. If this definition over-rides a superclass `typedef`, then it must be a subclass of the over-ridden field's type.
Contains:	expression	optional	If present, this is the field's default value. Fields with default values do not need to be assigned when the type is used in an `instance` element.

Element: `function`

Defines a function interface. If a function body is provided, then this function may be called from any scope in which it is visible. Otherwise, this creates an abstract function definition that must be over-ridden before being used.

If there is already a function by this name in some superclass, then this function definition is an over-ride of the superclass definition. It may be abstract or concrete, irrespective of whether the suprclass definition is abstract, or concrete. It must have the same number of parameters as the function definition in the superclass, and each parameter type must be a subclass of the parameter type for the corresponding parameter in the superclass.

If one or more elements are present after the list of `param` elements, then this is a concrete function definition. It may be used in a `functionCall` element in any context where a compile time expression is required. Elements after the last `param` must be zero or more `setVar` elements, and the last must be a `returnValue`. The `setVar` elements must each assign values to variables with different names, and none of the variables may have the same name as a parameter. Each `setVar` variable may be referenced in any expression within the function body, after the `setVar` in which it is defined. The `returnValue` element contains the expression that the function returns as its value.

Functions may be defined recursively, but recursive evaluation is limited to compile time.

Appears in:	`application`
	`class`
	`entityDef`
		Contexts in which a function may be defined. Abstract functions may be defined only in class definitions.

Attributes:	`name`	required	Symbol name, must not already be in use by another function in this class, unless this definition over-rides another in the superclass
	`returns`	required	Data type of the function's return value. If this definition over-rides a superclass function definition, then it must be a subclass of the superclass function's return type.
Contains:	`param`	≥ 0	Definitions of the function's formal parameters. All formal parameters must have distinct names. If this function definition over-rides a superclass function definition, then the number of parameters must be the same and the new parameter types must be subclasses of the corresponding types in the superclass definition.
	`setVar`	≥ 0	Computation variables. Each `setVar` symbol must be distinct from other `setVar` and `param` symbols in this function. If present, *`returnValue`* must also be present

`returnValue`	0 or 1	If present, must be last. Defines the expression to be used as the function's return value. If absent, the function is abstract.

Element: `param`

Defines a formal parameter to a function. This always appears as an element within a `function` element. The parameter name string must not be not be the name of another parameter to the same function.

Appears in:	`function`		
Attributes:	`name`	required	Must be a valid symbol name, not already used as a parameter name in this function
	`type`	required	Qualified symbol name of the parameter type

Element: `returnValue`

This appears exactly once in each function definition. It is the last (and possibly only) element, and defines the expression to be used as the function's return value.

Appears in:	`function`	Occurs once, as the last element. Not present if the function definition is abstract.

Contains:	expression	required	Expression to be passed as the function's return value.

Element: `setVar`

Defines a symbol for local use. This symbol is not visible externally. It must be assigned a value at the time of definition. The name string must not already be in use in the local scope as a formal parameter or in another setvar element.

Appears in:	`functionDef`		
Attributes:	`name`	required	Must be a valid symbol name, not already used as a parameter name or `setVar` variable in this function
	`type`	required	Qualified symbol name of the variable type
Contains:	expression	required	

Element: `typeDef`

A `typedef` element defines an application data type, either a tuple value (like a C `struct` or Ada `record`), or a scalar. If the type name repeats a type name already used in a superclass of the containing class, then this is an over-ride of that supertype definition. If the type explicitly extends another, then the extended class is the supertype.

Any fields not mentioned in this `typedef` are inherited from the supertype. Any field in this supertype and also in the superclass `typedef` must be redeclared with a subtype of the definition in the supertype. Any field in this supertype that is not present in the supertype is an extension to the supertype.

Appears in:	`application`		
	`class`		
	`entityDef`		
Attributes:	`name`	required	Symbol name for this data type.
	`extends`	optional	Qualified symbol name of the supertype, if present.
Contains:	`field`	≥ 0	Defines one of the data elements in this type. If the field name is the same as a field name in the supertype, then this is an over-ride definition. In that case, the field type must be a subclass of the type of the field being over-ridden.

A.1.2 HDL annotation

The HDL annotation subset of LAMPML integrates LAMP logic with the HDL components that comprise the accelerator logic. The HDL code is enclosed in an `entityDef` element, and has LAMPML annotation mixed in freely.

Element: `bind`

Create a binding of a value import, defined in the entity specified by the enclosing element, to a value defined in the current context. The imported value must be a subclass of the type specified by the `import` declaration in the entity. This must be provided for all `importSym` declarations that do not have default values assigned.

This is used both in the `entityUse` element in the HDL annotation subset of LAMPML, and also in the `root` element of the integration subset.

Appears in:	`entityUse`		
	`root`		
Attributes:	`importSym`	required	Symbol name defined in an `importSym` element of the entity being instantiated.
Contains:	expression	required	Expression or text string to bind to the named symbol.

Element: `bindType`

Create a binding of a type import, defined in the entity specified by the enclosing element, to a type defined in the current context. The imported type must be a subclass of the type specified by the `typeImport` declaration in the entity. It does not necessarily need to be a concrete class definition, however, unless the entity uses it in a way that requires concrete definition. This must be provided for all `importType` declarations that do not have default types assigned.

This is used both in the `entityUse` element in the HDL annotation subset of LAMPML, and also in the `root` element of the integration subset.

Appears in:	`entityUse`		
	`root`		
Attributes:	`importSym`	required	Symbol name defined in an `importType` element of the entity being instantiated.
	`bindTo`	required	Qualified name of a defined data type or class type.

Element: `entityDef`

Each annotated HDL file consists of exactly one `entitydef` element. This is the top-level element that contains all information about the annotated HDL.

Appears in:	Top level	This is never contained in another element.

Attributes:	`name`		Must be a valid symbol name.
Contains:	Text	≥ 0	HDL language statements
	expression	≥ 0	Evaluate to values to be inserted into HDL code
	`entityUse`	≥ 0	Instances of entities internal to the current one
	`function`	≥ 0	Defines a function for use within the entity
	`insertHdr`	1	Defines position at which automatically generated header file is to be inserted.
	`insertLocal`	1	Defines position at which automatically generated local definitions are to be inserted.
	`message`	≥ 0	Generates text at standard output
	`symDef`	≥ 1	Defines a symbol for local use and assigns it a value
	`symRef`	≥ 1	HDL reference to a LAMPML symbol
	`symExport`	≥ 1	Defines a symbol readable from outside the current scope, and optionally assigns it a default value
	`symImport`	≥ 1	Defines a symbol assignable in `bind` elements outside the current scope, and optionally assigns it a default value

`typedef`	≥ 0	Defines a type definition for use within the entity
`typeImport`	≥ 1	Defines a type symbol assignable in `bindType` elements outside the current scope, and optionally assigns it a class constraint
`writeHdr`	≥ 1	Exports text from the current entity to the common header file shared across all entities and generated by the LAMP tools.

Element: `entityUse`

Creates an instance of a new logical entity nested within the current entity. This creates the logical structure of the entity only, and does not actually instantiate logic. Instead, this works with the HDL's instantiation mechanisms, and adds capabilities not present in the underlying language. Use and behavior of this construct are described in detail in section 5.4, and usage examples are given in Appendix C.

Appears in:	`entityDef`		
Attributes:	`name`	required	Symbol name of the entity to be used as the component instance.

Contains:	`bind`	≥ 0	Binds a value to one of the value import symbols defined by a `symImport` statement in the instantiated entity.
	`bindType`	≥ 0	Binds a class type to one of the type import symbols defined by a `typeImport` statement in the instantiated entity.

Element: `insertHdr`

Defines the point in the HDL file at which reference to an automatically generated header file is to be inserted.

Appears in:	`entityDef`	Must appear exactly once per `entityDef` element

Element: `insertLocal`

Specifies the point in the HDL file at which automatically-generated local definitions are to be inserted. This appears once in the declaration area of a VHDL architecture. If multiple architectures appear in one `entityDef` source file, the `insertLocal` element should be repeated once in the declaration area of each one.

Appears in:	`entityDef`	Usually appears exactly once per `entityDef` element.

Element: `message`

Generates text at standard output, usually the console output. This is useful for examining values assigned to application symbols and for other user-defined status reporting.

Appears in:	`application`	Displays user message at stdout
	`entityDef`	
Contains:	expression	Value to be output.
	unformatted text	Literal text is copied verbatim to standard output. Newlines (carriage returns) in the text appear in the output.
	`symRef`	Displays symbol resolution in the message context

Element: `symDef`

Creates a local symbol definition, accessible only within the current entity. Must be unique at a given lexical scope, but replaces any symbol with the same name defined at an outer scope and may be replaced by another definition and a more deeply nested scope.

Appears in:	`entityDef`	Defines a values used locally in calculations.

Attributes:	`name`	required	Symbol name visible both inside and outside the containing entity. The symbol must not already be in use in the current scope.
	`type`	required	Qualified symbol name for the value type
Contains:	expression	required	Value to be assigned to the symbol. Must be a subtype of the declared type.

Element: `symExport`

Creates a local symbol definition, accessible outside of the current entity. The symbol's value *must* be assigned when the symbol is defined, and can not be changed.

Appears in:	`entityDef`		
Attributes:	`name`	required	Symbol name visible both inside and outside the containing entity. The symbol must not already be in use in the current scope.
	`type`	required	Qualified symbol name for the value type
Contains:	expression	required	Value to be assigned to the symbol. Must be a subtype of the declared type.

Element: `symImport`

Creates a local symbol definition within the current entity, such that the symbol value may be over-ridden wherever the current entity is instantiated. The symbol may be assigned a default value, to be used if the entity instantiation does not bind a new value to the symbol. If a default value is present, the instantiation need not bind a value. If the value is missing, however, the symbol binding at entity instantiation is required.

Appears in:	`entityDef`		
Attributes:	`name`	required	Symbol name visible both inside and outside the containing entity. The symbol must not already be in use in the current scope.
	`type`	required	Qualified symbol name for the value type
Contains:	expression	optional	If present, defines the default value for the symbol

Element: `symRef`

Symbol reference, inserts symbol into HDL code.

This is distinct from the `varRef` element used in expressions. A `varRef` element is replaced by the data content of the variable named. A `symRef` element is replaced by the symbol itself. This is used with symbols defined in a `typeImport` element, which are bound to some named type by default bindings or by `bindType` elements where the component is instantiated. When a `symRef` element names one of these `typeImport` symbols, the element is replaced by the name of the type to which the symbol is bound. This may imply additional HDL text elsewhere, to define symbols or function bodies in synthesizable form.

Appears in:	`entityDef`		
Attributes:	`name`	required	Qualified name of symbol being referenced.

Element: `typeImport`

Defines a type input to the current entity. The `name` string is the name by which the type is referenced locally. The `type` symbol defines a superclass type for the `type` symbol. When the current entity is instantiated using an `entityRef` or `root` element, a `bindType` must assign an actual type to this import symbol. That actual type must be a subclass of `type`. The binding may be omitted if `type` is concrete, in which case `name` refers to `type` itself.

Appears in:	`entityDef`		
Attributes:	`name`	required	Symbol name visible both inside and outside the containing entity. The symbol must not already be in use in the current scope.

`type`	required	Name of the type bound by default to the named symbol. If a `root` or `entityUse` element refers to this class, it may over-ride this type definition with a `bindType` element. In that case, the new binding must be to a subclass of this type.

Element: `writeHdr`

Inserts text into the system-generated header file. The LAMP tools create a header file containing definitions that are to be shared by the whole set of HDL files generated during compilation. If some HDL component has definitions to export to other components, it uses a `writeHdr` element to add text to that file.

Appears in:	`entityDef`	
Contains:	expression	Value to be assigned to the symbol. Must be a subtype of the declared type.
	unformatted text	Literal text is copied verbatim to standard output
	`symRef`	Inserts a HDL-compatible form of the LAMP symbol, including any necessary auxiliary definitions.

A.1.3 Integration subset

The LAMPML integration subset handles the top-level indexing and aggregation functions. It lists the entire set of files required for any one application family.

Element: `application`

There is exactly one `application` element in any model or model instance. This is the top level aggregate, the one contains all other definitions. The application as a whole is identified by the symbol specified in the `name` attribute.

The `application` element is used in two ways. First, it aggregates the set of entities and classes that define an application family. This application normally lacks the concrete application and FPGA-specific definitions needed for creating an instance of an accelerator. In the second form, the abstract `application` definition is copied and additional data provided for a specific application. That second, fully specified application can be used to instantiate an accelerator for some application

Appears in: Top level			
Attributes:	`name`	required	Symbol name of the entity to be used for the application.

Contains:	`class`
	`const`
	`function`
	`typedef`
		≥ 0	Define symbols specific to this application

	`message`	≥ 0	Generates text at standard output
	`root`	1	Binds a class type to one of the type import symbols defined by a `typeImport` statement in the root entity. Must be the last element in the `application`.
	`useClass`	≥ 1	Load one or more class declarations from the named files.
	`use`	≥ 1	Load an annotated HDL entity from the name file.

Element: `root`

Defines an instance of an entity, defined in an `entityDef` declaration in one of the preceding `use` statements. Bindings contained in the root definition refer to imported symbols defined in the named entity, for import symbols representing both types and values. A binding must be provided if the symbol is not assigned a default value where it is declared in the entity. If a default value is provided, it replaces the symbol value in this instance of the root entity.

Appears in:	`application`		
Attributes:	`name`	required	Symbol name of the entity to be used as the root instance.
Contains:	`bind`	≥ 0	Binds a value to one of the value import symbols defined by a `symImport` statement in the root entity.
	`bindType`	≥ 0	Binds a class type to one of the type import symbols defined by a `typeImport` statement in the root entity.

Element: `use`

Specifies an annotated HDL file to be included in the model or instance. The file name must resolve to a valid entry in the local file system, and must contain valid LAMPML syntax. Outside of that, the character string used for a file name has no significance. In particular, it has no relationship to the definitions it contains, and need not use any particular suffix.

The `use` elements in one application must all refer to `entityDef` elements with different `name` symbols.

Appears in:	`application`		
Attributes:	`fileName`	required	Character string representing a file containing an annotated HDL entity. The entity name symbol must not duplicate any other symbol in the current scope.

Element: `useClass`

Imports one or more class definitions to be used in the current model. The file name must resolve to a valid entry in the local file system, and must contain valid LAMPML syntax. Outside of that, the character string used for a file name has no significance. In particular, it has no relationship to the definitions it contains, and need not use any particular suffix.

The `useClass` elements in one application must all refer to `class` elements with different `name` symbols.

Appears in: `application`			
Attributes:	`fileName`	required	Character string representing a file containing a class definition. The symbol used as the class name must not be used elsewhere in the current scope.

A.1.4 Expressions

Computed expressions appear in all subsets of LAMPML. The syntax of expressions is recursive, allowing expressions of arbitrary complexity. Expression values may be of any type, including integer, boolean, bit-string, and user defined types. The following elements create expressions.

Element: `functionCall`

Invokes a LAMPML function. The `functionCall` element is evaluated and logically replaced by the value to which it evaluates. The named function must have a concrete definition at the point where it is invoked.

If all of the parameters are compile time expressions, then the `functionCall` reference is also a compile time expression. If HDL symbols are passed to the function, the `functionCall` element is treated as a run-time expression. HDL symbols must be passed as the contents of a `passParam` element. When `passParam` is used in place of a

LAMPML expression, type checking can not be performed. Any errors in type matching will be detected during synthesis.

Compile time expressions may use functions recursively, but run-time expressions may not use recursion.

Attributes:	`name`	required	Qualified name of the function to be invoked.
Contains:	expression `passParam`	≥ 0	Actual parameters to the function. The number of values must match the `functionCall` declaration. Parameter value types must each be of a subtype of the corresponding formal parameter's type.
Return type:			Highly dependent on specific usage.

Element: `instance`

Creates an instance of some data type defined by a `typeDef` element. The `type` attribute specifies the name of the type for which an instance is being created. This must contain one `setvar` for each field in the type, where the variable name in the `setvar` refers to the field name. The `setvar` for some field may be omitted only if the field has a default value assigned in the definition of `type`, in which case that default value is used for the field value. It may be used anywhere an expression is expected

Attributes:	`type`	required	Qualified symbol name specifying the type of which an instance is required.
Contains:	`setVar`	≥ 0	Each one assigns a value to one field of the type.
Return type:			Specified by the `type` attribute.

Element: `literal`

Create a value of some primitive type

Attributes:	`type`	required	Qualified name of the data type to be instantiated.
Contains:	Raw text	required	Unformatted text, a string interpreted as the primitive value.
Return type:			Specified by the `type` attribute.

Element: `paramType`

This is not itself an expression element, but an element within `synthSize`. See the description of the `synthSize` element for information on using this.

Appears in:	`synthSize`		
Attributes:	`name`	required	Qualified symbol representing a data type.

Element: `passParam`

This element appears only inside of a `functionCall` element, not as an expression itself. It is used to pass HDL values to LAMPML functions, so may be used only in function parameters for function calls made within an `entityDef` element. See the `functionCall` description for information on using this element.

Appears in:	`functionCall`	
Contains:	text	HDL expression to be passed to the LAMP function.

Element: `synthSize`

Estimate the number of units of logic resources necessary for creating one instance of a data type or function. See section 4.4.6 for a discussion of accuracy issues in synthesis estimation. This is useful only when the function or data element is instantiated for runtime evaluation. Compile time functions and values do not necessary have any direct effect on the synthesized output, so synthesis estimation of compile time expressions is not meaningful.

If the `name` attribute refers to a data type, this returns the number of bits needed for allocating one value of this data type. If the `name` attribute specifies a function, the returned value estimates the number of logic elements needed for one instance of the function.

Contained `paramType` elements are meaningful only when the name attribute refers to a function. If absent, synthesis estimation uses the parameter types specified as formal parameter types in the function definition. It is always valid, however, to pass an actual parameter that is a subtype of the formal parameter. Depending on differences between the subtype and declared type of the formal parameter, that could have significant effect on resource estimation.

Use the `paramType` elements when the actual parameters to a function instance differ from the formal parameter types. When used, there must be one `paramType` element per formal parameter. Each `paramType` must name a data type that is a subtype of the corresponding formal parameter.

Attributes:	`name`	Qualified symbol of a data type or function.
Contains:	`paramType` ≥ 0	If the name attribute refers to a function, then type specifications for each parameter may be supplied. Each `paramType` element corresponds, by position, to one of the function's formal parameters and must be a subtype of the corresponding parameter.
Returns:	`integer`	Estimated number of logic elements required

Element: `uniqueID`

Creates a symbol that is guaranteed to be globally unique. In particular, one instance of the uniqueID element in one entity will generate different, unique values for each instance of the entity. This expression generates a value of type `symbol`.

Attributes:	`prefix`	optiona l	String to use as leading substring of the generated symbol.
Returns:	`symbol`		Result is of primitive type `symbol`.

Element: `varRef`

This element is replaced by the data content of the variable. It is used to substitute the compile time constant value of the variable into an expression or fragment of HDL text.

Appears in:	expressions		
Attributes:	`name`	required	Qualified name of the symbol whose value is to be used in the current position. Must be visible in the current scope.

A.2 Predefined symbols

The following symbols are predefined by the LAMP tools, and are available to the LAMPML developer.

A.2.1 Predefined types

The predefined types are the primitive data types from which more complex types are built.

`baseType`

This data type is the base class for inheritance. Every class except this is a subclass of some of other. If no explicit superclass is named using the `extends` attribute, then a class is implicitly a subclass of this type.

`boolean`

Boolean data has the conventional logical meaning. Two primitive values exist in this type: `true` and `false`. These are returned by relational tests, are required as test values in `if` function calls, and are inputs and outputs of logical functions.

`integer`

This type covers the full range of signed 64 bit integer values. Subtypes of `integer` restrict the range of the value, and affect the number of bits allocated to each value.

natural

This covers the full range of unsigned 63-bit values. This is a subtype of `integer`.

std_logic

Use this data type for LAMPML logic that represents VHDL values with type of the same name. The LAMPML type supports only bit values 0 and 1, not the full set of generalized bit values allowed by the VHDL standard.

std_logic_vector

This maps closely to the VHDL data type of the same name, with the same restrictions on bit values as `std_logic`. Subtypes of this type specify a number of bits *n*, and the range of bit values is implicitly 0 to *n-1*.

symbol

Symbol values are used for on-the-fly generation of names for different instances of a single entity.

A.2.2 Predefined functions

The following functions are provided as primitives available in LAMPML.

Function: add

Arithmetic sum of two or more values.

Parameters:	≥ 2	`integer`

Result:	`integer`	sum of parameter values.

Function: `and`

Logical AND of boolean parameters.

Parameters: ≥ 2	`boolean`	
Result:	`boolean`	Logical AND of all parameters.

Function: `bitcnt`

Determine the number of bits required for a value, $\lfloor log_2\, x \rfloor$. If the parameter is a negative value, the number of bits for its positive form is reported, plus one. If the parameter is zero, the value is one.

Parameters: 1	`integer`	
Result:	`integer`	Number of bits required to represent the value, as described. For compile-time use only, not for synthesis.

Function: `divide`

Integer division of first value by the second. If the first is not an exact multiple, the result is truncated. This should be used only for compile-time expressions.

Parameters: 2	`integer`	
Result:	`integer`	Result of dividing first parameter by second.

Function: `eq`

Equality test between two values. The initial implementation restricts this to comparison between integer values.

Parameters: 2	`integer`	
Result:	`boolean`	True if values have same bit pattern.

Function: `extract`

Extracts a bit field from a value. The first parameter is the value from which a bit field is to be extracted. The second parameter is the position at which the bit field starts. The least significant bit is numbered zero, and more significant bits have higher positions – negative values are not allowed. The third parameter specifies the number of consecutive bits to extract, starting at the given position and working towards higher significance. The number must be positive, zero- and negative-length fields are not allowed. The start position plus the field width must not exceed the width of the first parameter.

Parameters: 1	`integer`	Value from which to extract bits
	`std_logic_vector`	

	2	`integer`	Start position and field width
Result:		`Same as type of first parameter`	Bit field extracted from the original value.

Function: *`tName`*`:fromStdVec`

Converts a bitstring value to a LAMPML typed value. LAMP defines a different form of this function for each data type named *`tName`*, and the function name is scoped under that type as shown. This function is normally used in annotated HDL code, to convert typed data to bitstrings in context where only bitstrings are allowed.

Parameters:	1	`std_logic_vector`	Bitstring to convert. The length of the bitstring is given by `synthSize(tName)`
Result:		`tName`	Typed value, according to the type in which the fromStdVec function is scoped..

Function: `ge`

Inequality test between two values.

Parameters:	2	`integer`	
Result:		`boolean`	True if first parameter value is greater than or equal to the second.

Function: `gt`

Inequality test between two values.

Parameters: 2	`integer`	
Result:	`boolean`	True if first parameter value is greater than the second.

Function: `iand`

Bitwise AND of arithmetic values.

Parameters: ≥ 2	`integer`	
Result:	`integer`	Bitwise AND parameter values.

Function: `if`

This function returns a value selected according to the boolean condition[s] provided. If corresponds to the following logic:

```
if (b1)       then v1
elseif (b2) then v2
…
else        vN ;
```

There must be an odd number of parameters. Starting from parameter number 1, the first, odd-numbered parameters are boolean test conditions, except the last parameter. Even-numbered parameters and the last parameter are values from which selection is made.

All of the value (as opposed to test) parameters contribute to determining the type of the expression. The return type one of the parameter types, and is the one of which all others are a subtype. If the parameter types of the values do not include one that is the supertype of all other, then the expression is not correctly formed.

Parameters:	Odd N = 2I+1	mixed	Alternation of boolean and other data types.
	# 2i-1, 1≤ i≤ I	`boolean`	Test conditions. The values are tested in order, until one is found true. In that case, the value is parameter number 2i. If none are true, the return value is parameter number 2I+1.
	# 2i, 1≤ i≤ I # 2I+1	any	Possible expression values, selected as described above.
Result:		any	Selected according to logical conditions.

Function: `inot`

Bitwise inverse of arithmetic value.

Parameters: 2	`integer`	
Result:	`integer`	All parameter bits inverted.

Function: `ior`

Bitwise OR of arithmetic values.

Parameters: 2	`integer`	
Result:	`integer`	Bitwise OR parameter values.

Function: `le`

Inequality test between two values.

Parameters: 2	`integer`	
Result:	`boolean`	True if first parameter value is less than or equal to the second.

Function: `lshft`

First argument arithmetically shifted left by the amount of the second argument. The second argument, shift amount, must have a non-negative value for compile-time evaluation.

Parameters: 2	`integer`	
Result:	`integer`	First argument, left-shifted by second argument's amount.

Function: `lt`

Inequality test between two values.

Parameters: 2	`integer`	
Result:	`boolean`	True if first parameter value is less than the second.

Function: `mod`

Integer remainder when first parameter is divided by the second. This should be used only for compile-time expressions.

Parameters: 2	`integer`	
Result:	`integer`	Remainder after dividing first parameter by second.

Function: `multiply`

Arithmetic product of two or more values.

Parameters: 2	`integer`

Result:	`integer` Product of parameter values.

Function: `ne`

Inequality test between two values. The initial implementation restricts this to comparison between integer values.

Parameters: 2	`integer`
Result:	`boolean` True if the parameters differ.

Function: `negate`

Additive inverse of the value. The result is the original arithmetic value with the sign changed.

Parameters: 2	`integer`
Result:	`integer` Arithmetic negation (unary '-').

Function: `not`

Logical negation.

Parameters: 2	`boolean`
Result:	`boolean` Logical inverse of parameter.

Function: `or`

Logical OR of boolean parameters. This is not a short-circuit operator, i.e. it evaluates all parameters even if the result is already known to be true.

Parameters: 2	`boolean`	
Result:	`boolean`	Logical OR (parity) of all parameters

Function: `rshift`

First argument arithmetically shifted right by the amount of the second argument. The result is sign-filled. The second argument, shift amount, must have a non-negative value for compile-time evaluation.

Parameters: 2	`integer`	
Result:	`integer`	First argument, right-shifted by second argument's amount.

Function: `subtract`

Arithmetic difference of two values.

Parameters: 2	`integer`	
Result:	`integer`	First parameter minus second.

Function: *`tName`*`:toStdVec`

Converts a LAMPML typed value into a bitstring. LAMP defines a different form of this function for each data type named *`tName`*, and the function name is scoped under that type as shown. This function is normally used in annotated HDL code, to convert bitstrings back into typed data from contexts where only bitstrings are allowed.

Parameters:	1	*`tName`*	Typed value to convert
Result:		`std_logic_vector`	Bitstring to resulting from conversion. The length of the bitstring is given by `synthSize(tName)`

Function: `xor`

Exclusive OR of boolean parameters

Parameters:	≥ 2	`boolean`	
Result:		`boolean`	Exclusive OR (parity) of all parameters

A.3 High level syntax

XML notation, though easily machine-readable, makes a poor representation for human use. Although the XML-based LAMPML defines the fundamental semantics of the tools, an alternate representation, Convenience LAMP (or CLAMP) may be used as well. This section defines CLAMP syntax, with references to the LAMPML elements that implement them. The HDL annotation subset is can not generate marked up HDL directly, because of the tight textual coupling between the markup and the HDL text itself. Instead, this subset is a convenience for the HDL developer. It allows the developer to express design concepts in a relatively readable format (unlike XML), and is parsed into XML. The developer then uses the generated XML to annotate HDL files, or uses it as a skeleton in which to add HDL text.

CLAMP differs from existing languages in too many ways to call it a "version" or "derivative" of some more familiar language. That said, CLAMP syntax owes much to C++ and especially to Java. Where CLAMP expresses a familiar concept, it often uses familiar notation; novel appearance for its own sake is not a goal.

Literal CLAMP symbols are shown in **`bold`**. A symbol or punctuation mark that it *`italicized`* is part of the metalanguage, used for defining the syntax but not to be used in actual CLAMP code. Parentheses *`( )`* in the metalanguage indicate grouping in the usual way, and are distinct from parentheses **`( )`** that are actually used in CLAMP code.

Square brackets *[ ]* indicate that the contents may be inserted in CLAMP code or omitted, and are distinct from square brackets **[ ]** used in CLAMP code. An asterisk *** indicates that the one item to the left may be repeated zero or more times. A group of items separated by vertical bars */* represent alternatives, of which exactly one should be chosen. Metalanguage *symbols*$_1$ may be subscripted. The subscript has no meaning, but helps in clarifying a description where a single metasymbol is used more than once in a definition.

The *symbol* metasymbol refers to any LAMP symbol: an underscore character '_' or English letter, followed by any number of English letters, digits, and underscore characters. The *scopedSymbol* metasymbol refers to a sequence of one or more symbols, separated by period '.' characters. The meaning of a scoped symbol is described at the beginning of section Appendix A. LAMPML and CLAMP separator characters are different (colon vs. period), but scoped symbols have the same meanings in all other ways.

Comments may use either of the C++ forms. A comment may start with a // digraph outside of quotation marks, and continue to the first following new-line or end of file. The second form starts with a /* pair, not quoted, and continues to the next */ pair, irrespective of new-line characters. The /* and */ do not nest like parentheses, so /* /* /* /* */ is one valid, complete comment. Presence, absence, and content of comments are irrelevant to the meaning of CLAMP code in which they appear.

A.3.1 Model definition subset

class *symbol [***extends** *symbol]*

    { *(typedef* **;** *| const* **;** *| function* **;** *)** **}**

This is a top-level definition, represented by a `class` element in LAMPML. If the `extends` clause is present, it defines the superclass of which this is subclass. All symbols defined in the class are accessible from outside the class. Every `typedef`, `const`, or `function` must have name unique within the class, even among definitions of other kinds. For example, it is an error for a `typedef` and a const declaration to create the same name.

A subclass exports all the symbols that its superclass does, and possibly more as well. It is also free to over-ride any symbol defined in the superclass, as long as the over-ride is of the same kind (`typedef`, `const`, or `function`) as the over-ridden symbol, and follows additional over-riding rules defined below. If the subclass contains a symbol not defined by the superclass, that is an extension to the superclass interface.

const *scopedSymbol symbol* [**:=** *expression*] **;**

The `const` declaration defines a data value exported from the containing class. The *scopedSymbol* names the data type of the value, and the *symbol* defines the name by which the constant value is accessible. The type may also be specified by one of the primitive symbols, `boolean`, `integer`, or `natural`.

When there is an expression assigned to the symbol, it is a concrete constant definition. Omission of the initializer creates an abstract constant definition. Like its concrete counterpart, an abstract constant is part of the interface defined by the class, and can be used in expressions where its type is appropriate. An abstract constant can not be evaluated, however. Instead, the const declaration must be over-ridden by a subclass, and assigned a value in the subclass. The subclass definition, accessed through its abstract interface, can be evaluated.

If *symbol* is also defined in the superclass, it must also be a `const` symbol. The type of the subclass constant may be redefined, as long as the subclass constant's type is a subtype of the type used by the superclass `const` declaration. The subclass definition may be concrete or abstract irrespective of concreteness in the superclass.

typedef *symbol*$_1$ *[* **extends** *scopedSymbol*$_1$ *]*

[**{** *[scopedSymbol*$_2$ *symbol*$_2$ **;** *] ** **}** *]*

or

typedef *symbol*$_1$

(**boolean** */* **integer [** *expression* **]** */* **natural [** *expression* **]** *)*

Defines *symbol*$_1$ to be a data type. A type may be a tuple of named fields, or may be based on one of the primitive types `boolean`, `integer`, or `natural`.

The first syntactic form defines a tuple. It consists of a set of fields, each named by a *symbol*$_2$. Every *symbol*$_2$ must be unique within the typedef. The *scopedSymbol*$_2$ defines the data type of the field, and must be a valid type name or primitive. No guarantees are made about the HDL representation of the type.

The typedef may be a subtype of another typedef, named in the `extends` clause. In that case, the new typedef contains all the fields in the supertype, and may contain additional fields. Fields in the supertype may be over-ridden, by reusing a superclass field name in the subclass definition. In that case, the type of the over-riding field must be the same as that in the supertype, or a subtype of it.

If the class containing the typedef is extends another class containing a typedef of the same name, then the new typedef is implicitly an extension of the one in the superclass. This also implies that, if *symbol*$_1$ is used in the superclass, that it is also a typedef definition.

If the bracketed list of field definitions is missing, then the type definition is abstract. It may be referenced wherever a type name is required, but must be assigned a meaning by a subclass definition. An abstract type with no extends clause may be over-ridden by a subclass definition of either syntactic form.

The second syntactic form treats the symbol as an alias of one of the predefined types. The expression value in the integer or natural declaration states the number of bits to be assigned to the value. An `integer` value includes a twos-complement sign bit, `natural` values are unsigned. If the field is over-rides a superclass definition, the superclass definition must be of the same type, and the new type must contain at least as many bits as the superclass type. The value of the *expression*, if present, must be at least 1 and no more than 32.

```
function scopedSymbol₁ symbol₁
        ( [ scopedSymbol₂ symbol₂ [ , scopedSymbol₂ symbol₂ ]* ] )
        [ { [ scopedSymbol₃ symbol₃ := expression ; ]*
                return expression } ]
```

This defines a function named *symbol*$_1$, with return type *scopedSymbol*$_1$. The list of formal parameters is enclosed in parentheses, and may be empty. If nonempty, it consists of a set of formal parameters name *symbol*$_2$, each with its type specified by *scopedSymbol*$_2$. Every formal parameter's name must be unique within the parameter list.

If the function body is omitted, then the function is abstract. It may be used in expressions, but must have a body provided by a subclass over-ride before being used. If the function body is present, the function body is concrete. The body consists of zero or more variable definitions, and ends with a return statement. Each variable must have a *symbol*$_3$ name distinct from all others in that function, and distinct from all parameter names. The variable's type is specified by its corresponding *scopedSymbol*$_3$.

Any function name may be used only once in a class. If the function name is used in the superclass, then it over-rides that function definition. The over-ride may be concrete or abstract irrespective of the concreteness of the superclass definition. An over-ride must have the same number of parameters as the over-ridden function definition, must have a return type that is a subtype of the over-ridden function's, and must have parameter types that are subtypes of the over-ridden function's. Parameter names need not be the same as in the over-ridden function.

The function may be defined recursively, but recursive functions can only be evaluated at compile time. It is an error to synthesize a recursively defined function. This applies whether or not the recursion is indirect, i.e. through some sequence of other functions that eventually call the current function.

A.3.2 Integration subset

`application` *`symbol`* **`{`**

`(useClass | use | const | typedef | function | class)* root` **`}`**

This aggregates all of the files used for an application: class definitions, annotated HDL files, and a designated root component. As a convenience, is also allows auxiliary definitions of any kind. All `const`, `typedef`, `function`, and `class` definitions must have distinct names, and must be distinct from the class names defined by `useClass` statements

`useClass` `"filename string"` **`;`**

Accesses a file containing a class definition. The class and all of the symbols it defines are available after this statement. Every class in one application must have a unique name, which is not related to the name of the file in which it is contained.

`use “filename string” ;`

Accesses a file containing annotated HDL, i.e. containing an `entity` definition. All of the symbols exported from entity are available after this statement. Every entity in one application must have a unique name, which is not related to the name of the file in which it is contained.

`root` *`symbol [`* **`{`** *`binding [ , binding ] *`* **`}`** *`]`*

This defines one entity as the root component, i.e. the top-level component that is instantiated once and in turn instantiates all other components. Bindings have one of the two following forms:

`class` *symbol*$_1$ **`:=`** *symbol*$_2$ or *symbol*$_1$ **`:=`** *expression* .

In the first case, the *symbol*$_1$ must be one of the symbols named in an `import class` statement in the root entity. The *symbol*$_2$ must name a class definition compatible with any constraints set in the `import class` statement in the entity definition. In the second case, the *symbol*$_1$ must be one of the symbols named in a value `import` statement in the root entity. Then, the expression must have a type compatible with the *symbol* type declared in the `import` statement. Every symbol imported by the root entity must appear in a binding, unless it is declared with a default binding. No *symbol*$_1$ may appear twice in the list of bindings.

A.3.3 HDL annotation subset

entity *symbol* **{** *(typedef | function | import | instance |*

scopedSymbol symbol **:=** *expression* **;** *)** **}**

In addition to local definitions for defining the HDL model, this describes the hierarchy of entity inclusion references. Assignment statements (with a `:=` operator) declare a type, a variable name, and a value for the variable. The assigned value must have type compatible with the declared type, and the variable name is exported from the `entity` definition. The variable name must not appear elsewhere within the entity declaration, either in an assignment, `import`, or other symbol definition statement. The `typedef` and `function` definitions have the forms described in section A.3.1. Only the instance and import statements are new to the HDL annotation subset of CLAMP.

instance *$symbol_1$ $symbol_2$ [* **{** *binding [* **,** *binding] ** **}** *]* **;**

This defines instantiations of other entities. Note that this does not create an HDL instance (or `component` in VHDL). It does, however make component definitions of entity type *$symbol_1$* available, with the bindings given, using the name *$symbol_2$*. This translates into an `entityUse` element in LAMPML, and creates a logical instance. See that element's description for a more complete description of this statement's semantics. Bindings have the same syntax and meanings as in the `root` statement of section A.3.2. Instance names (*$symbol_2$*) can be used in scoped names, to allow access to constants and functions defined in those `entity` declarations.

import *(**class** symbol$_1$ [**extends** symbol$_2$]*

    | *scopedSymbol symbol$_3$ [* **:=** *expression])* **;**

These two forms of the import statement correspond to LAMPML's typeImport and symImport elements, respectively. Their meanings are described in section A.1.2.

A.3.4 Expressions

Expressions may evaluate to any type, whether primitive or defined by a `typedef` statement. A *primary* expression is one of the following:

(*expression* **)**	Used for grouping
scopedSymbol	Refers to a field, constant, or variable
integer literal	Use C syntax for hex or decimal constants, to 64 bits
true | **false**	Literals for boolean values
function call	Invokes a user-defined function
typedef instance	Creates an instance of a tuple value defined by a `typedef`
"string"	Quoted character string.

Character strings are also used to `std_logic` and `std_logic_vector` literals. Those string may use only the characters 1 (one) and 0 (zero) between the quotes.

A function call has conventional form, a *scopedSymbol* function name followed by parentheses enclosing zero or more expressions for actual parameters. The number of actual parameters must be the same as the number of formal parameters in the function definition, and the expression types must match the types of the corresponding definitions of formal parameters. A typedef instance has the following form:

```
new scopedSymbol ( [symbol := expression
          ( , symbol := expression ) * ] )
```

The *scopedSymbol* names a type to instantiate. Each *symbol* names a field in the type, whether defined directly or inherited from a supertype. Every field in the type must appear in one assignment, unless it has a default value assigned. No field may be assigned twice in one instance.

Operator expressions

Operator expressions have one of the following formats:

1. *a binaryOp b*

2. *unaryOp a*

3. *a*[*b* : *c*]

4. **if** b_1 **then** a_1 *[* **elsif** b_2 **then** a_2 *]** **else** a_3

Expressions of formats 1 and 2 resemble constructs in most familiar programming languages. Operators (*binaryOp* and *unaryOp*) and precedence are defined in Table 8. Expression format 3 extracts a bit field from the primary expression *a*.

Expression format 4 expresses conditional execution, a choice between the several values a according to boolean tests *b*. Exactly one of the expressions *a* will be evaluated. First, expression b_1 is evaluated. If true, expression a_1 is evaluated and used as the expression value. If b_1 is false, then expressions b_2 (if any) are evaluated, in order until one is found to be true. In that case, the corresponding expression a_2 is evaluated and used as the expression value. If none of the *b* values are true, then a_3 is evaluated and used as the expression value. This compile-time behavior makes it safe to define recursive functions, since deeper recursion or recursion termination is controlled by the decision about which *a* and *b* expressions to evaluate.

Post-synthesis behavior of the `if` expression is different. In that case, all *a* and *b* expressions are evaluated. This means that synthesized logic can not use recursive definitions. Since all branches of the `if` expression are evaluated, recursion can not be halted.

Table 8 lists the available operators and their precedence. Operators of higher precedence bind their operands more tightly than operators of lower precedence. For example, the '`*`' operator has higher precedence than '`+`', so the expression `a*b+c` has the meaning `(a*b)+c` .

Table 8: CLAMP operators and precedence

Operator	Prece-dence	Result type	Operand type[s]	Meaning
`if then elsif else`	0	(*)	(*)	*Conditional execution. Operand types and result types are defined in section A.2.2
`a || b`	1	`boolean`	`boolean`	Logical OR
`a ^^ b`	2	`boolean`	`boolean`	Logical exclusive OR (XOR)
`a && b`	3	`boolean`	`boolean`	Logical AND
`a = b`	4	`boolean`	`integer`	Equality test
`a > b`	4	`boolean`	`integer`	Inequality test
`a < b`	4	`boolean`	`integer`	Inequality test
`a >= b`	4	`boolean`	`integer`	Inequality test
`a <= b`	4	`boolean`	`integer`	Inequality test
`a != b`	4	`boolean`	`integer`	Inequality test
`a | b`	5	`integer`	`integer`	Bitwise OR of integer values

Table 8: CLAMP operators and precedence

Operator	Prece-dence	Result type	Operand type[s]	Meaning
`a ^ b`	6	`integer`	`integer`	Bitwise exclusive OR (XOR) of integers
`a & b`	7	`integer`	`integer`	Bitwise AND of integer values
`a << b`	8	`integer`	`integer`	Arithmetic left shift. Zeros fill the MSB of the result value.
`a >> b`	8	`integer`	`integer`	Arithmetic right shift. Sign-filled for signed *a* values, zero-filled for unsigned *a* values.
`a + b`	9	`integer`	`integer`	Twos-complement addition
`a - b`	9	`integer`	`integer`	Twos-complement subtraction
`a * b`	10	`integer`	`integer`	Twos-complement multiplication
`a / b`	10	`integer`	`integer`	Twos-complement division. Should be used only in compile-time expressions.

Table 8: CLAMP operators and precedence

Operator	Precedence	Result type	Operand type[s]	Meaning
`a % b`	10	`integer`	`integer`	Arithmetic modulus operation. Should be used only in compile-time expressions.
`-a`	11	`integer`	`integer`	Twos-complement negation
`~a`	11	`integer`	`integer`	Bitwise negation
`!a`	11	`boolean`	`boolean`	Logical negation
`#a`	11	`natural`	`integer`	Number of bits needed to represent an integer value, as described in the `bitcnt` function. Should be used only in compile-time expressions.

Table 8: CLAMP operators and precedence

Operator	Prece-dence	Result type	Operand type[s]	Meaning
`a[b:c]`	12	`integer`	`integer`	Bitfield extraction. Expression *b* specifies the LSB to return, starting at 0 and counting up. Expression *c* has specifies the positive number of bits to extract, and should be a constant for post-synthesis evaluation. If the type of expression *a* is `std_logic_vector`, then the result of the expression also has type `std_logic_vector`.

Appendix B. SIZING OF COMPUTATION ARRAYS.

Whatever the computation array and FPGA capacity, the accelerator designer generally wants one thing that no current design tools are able to state explicitly: *as many PEs as possible*, in order to maximize parallelism in between the PEs. This depends on the resource utilization per PE, permissible sizes for computation and memory arrays, and FPGA capacity.

There are three sources of information that affect an application accelerator's implementation: the choice of FPGA, which specifies the available amounts of each computing resource, the pattern of usage (or *growth law*) common across a given family of related accelerator designs, and the resource utilization specific to a particular member of the application family. This section discusses how those factors combine to define an application-specific accelerator.

FPGA resources

The FPGA resources are simply the programmable logic, hardware multipliers, block RAMs, and other features accessible to the logic designer. (Connectivity resources are usually allocated by development tools, and not directly available to the logic designer.) A larger FPGA in a given product family contains more of some or all resources, potentially allowing a larger computation array for a given application's accelerator. The resources of interest are expected to differ between applications or application families;

one family member may require hardware multipliers where another does not, for example.

Care must be taken in creating the resource abstraction since some resources are available only in specific quanta, such as block sizes for RAM bits. Extra care must be taken when the abstraction must cross FPGA product lines, since resources from different vendors are not always directly comparable. Block RAMS typify resource differences between vendors: the Xilinx Virtex-II Pro products contain 18Kb block RAMs, but comparable Altera Stratix-II chips offer a combination of 512b, 4Kb, and 512Kb RAMs.

Computation arrays and resource-limited growth

Different members of a given family of applications generally require different amounts of each FPGA resource for implementing a single PE in a computation array. Data paths can differ in bit width, and basic computations differ in the complexity of the calculations performed. Either way, the number of gates to implement a single PE will generally differ. As a result, the number of PEs that can be implemented in a given FPGA's resource budget will differ according to the specific computation.

Large FGPAs contain multiple different kinds of resources including hardware multipliers block RAMs as well as programmable logic elements. Different members of an application family differ in not just the quantity of resource needed per PE, but also the kind of resource. For example, simple docking applications use only logic resources for scoring voxels that overlap. Other family members may require RAM-based lookup tables for scoring, or hardware multipliers. Because different members of an application family consume different FGPA resources, different resouyrces limit the number of PEs per FPGA in each case. For example, a RAM-intensive applicationwould use only a small part of the FPGA's logic resources if RAMs required for more PEs are not available.

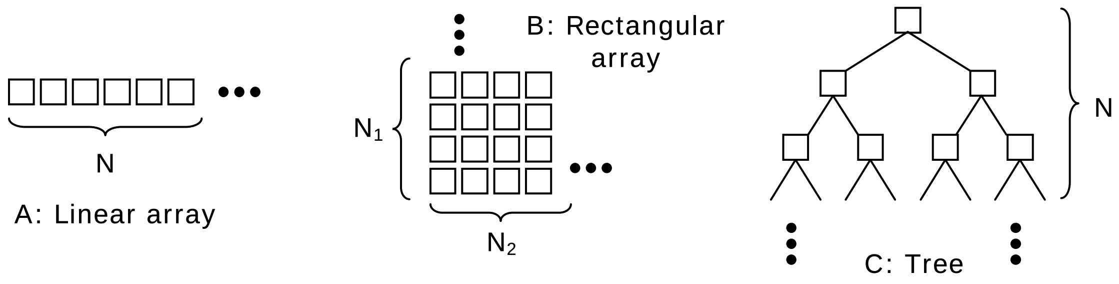
Many accelerator designs consist of more than one scalable design component. For example, the docking application consists of a logic-intensive computation array that interacts with RAM-based FIFOs. The sizes of the computation array and the FIFO array are determined by different features of the accelerator design. The two arrays also consume different kinds of resources: the computation array size is limited by the FPGA's logic resources, but the FIFO array is limited by available RAM resources. It is worth noting that the two array sizes are not wholly independent of each other.

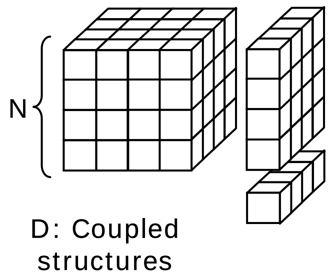
B.1 Array growth laws

In addition to application-specific resource usage per PE, and FPGA-specific resource availability, a third factor affects the size of the PE array that fits into a given FPGA: the array's *growth law*. This is the set of arithmetic rules that define the set of allowable array sizes. A crucial feature of the growth laws, explained in the next section, is that they invert the sense of the structural parameters commonly used for specifying numbers of component instances in some logic design. In this analysis, the set of parameter values is not supplied by the designer, but chosen by the tools to create the most useful accelerator possible given the resources available.

For generality and for ease of discussion, the same formalisms address arrays of memory elements, computing elements, and combinations. The goal of the discussion is to characterize repeatable arrays of computing resources, whatever the resource may be.

Figure 24 suggests growth laws for several kinds of computation array. Figure 24A, a linear array, is the simplest. It allows an array of size *N* for any positive integer value of the structural parameter *N*. If, as in Figure 24B, an array has rectangular shape, then there are two different structural parameters, N_1 and N_2, giving the two dimensions of the array. Figure 24C demonstrates an exponential rather than polynomial growth law, as just one





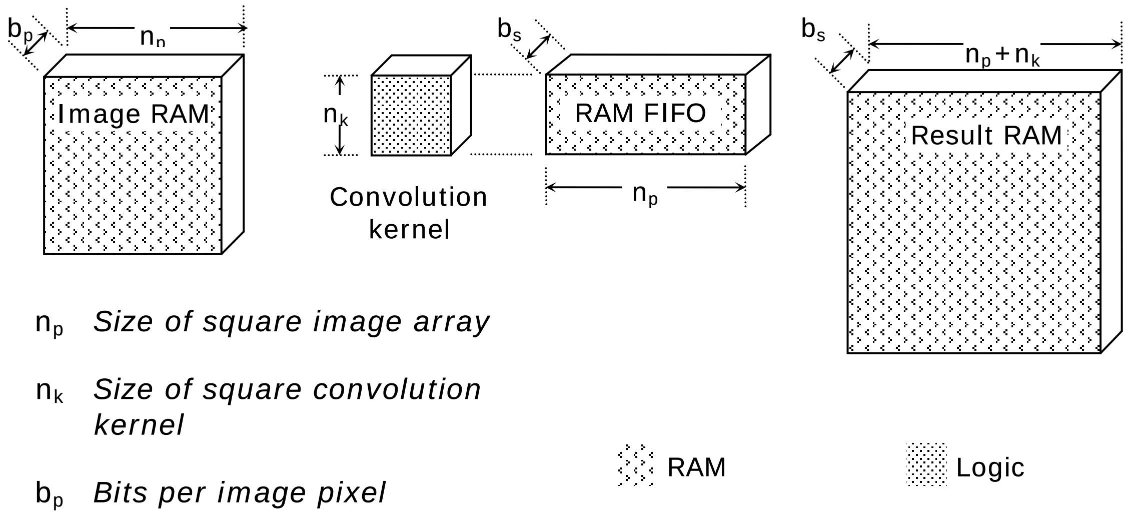
Array structure	*Growth law*
A: Linear	$k\ N$
B: Rectangular	$k\ N_1\ N_2$
C: Binary tree	$k\ (2^N - 1)$
D: Coupled structures	$k_1\ N + k_2\ N^2 + k_3\ N^3$

Figure 24. Growth of computation arrays

example of growth laws of arbitrary complexity. The case study of section 2.1.1, shown in Figure 6, uses a bus structure based on combinatorial Steiner systems. In that example, a term in the growth law involves an expression $n \in \{C_3^i \mid i=4,5,6,\ldots\}$, where $C_j^i = {}^{i!}/_{j!(i-j)!}$, a polynomial with j terms.

Figure 24D illustrates multiple coupled structures, possibly representing a cubical computation array, a square array of row reductions, and a linear array of column reductions, where the sizes of the structures are locked to each other. Each structure has a different constant value (k_1, k_2, k_3) representing resource utilization per computation cell. In this case, the growth law is represented by a polynomial with separate terms for each of the inter-related structures. Of course, all complicating features can occur in the growth law for any one system: multiple structural parameters describing nonlinear relationships between coupled structures.

Except for a linear structure, the growth law constrains the set of sizes available to a computation array. Suppose some system has a square computation array, with a growth law of N^2. The system could have enough logic resources to assemble (for example) 20 PEs. The biggest array that can be built from that set of resources has 16 PEs, the largest square integer less than or equal to 20. Resources for the other four possible PEs are not wasted, they are simply unusable given the problem constraint.



b_p
n_n
Image RAM
n_k
Convolution kernel
b_s
RAM FIFO
n_p
b_s
$n_p + n_k$
Result RAM
n_p *Size of square image array*
n_k *Size of square convolution kernel*
b_p *Bits per image pixel*
RAM
Logic

Figure 25. Sample application: Two-dimensional convolution, with resource

Figure 25 shows a hypothetical application, 2D convolution of an image with some kernel. For simplicity, it is assumed that both image and kernel are square. Both have different and arbitrary edge dimensions set by structural parameters n_p for the image plane and n_k for the kernel. As in Figure 24B, the two structural parameters are decoupled. FIFOs are needed in this application to hold partial sums of incomplete terms in the convolution, and RAMs are needed for the original image and convolution result. Assume that the kernel array uses only logic resources, and that the memories use negligible logic resources. Then the logic and RAM resource utilizations are approximated by the expressions labeled Equation 3, below.

$$Logic = g_k n_k^2 + g_c$$

$$RAM(bits) = b_p n_p^2 + b_s n_p n_k + b_s(n_p + n_k)^2$$

Equation 3. Resource usage for a sample system

where g_k, g_c, b_p, and b_s are application-dependent coefficients representing (respectively) logic elements per kernel PE, a fixed overhead of IO and control logic, number of bits per image pixel, and number of bits per convolution sum. Additional analysis is likely to show that g_k depends on the values of b_p and b_s. This does not change the basic form of the relationships, but adds more terms to the expression.

Sizing the accelerator

The initial assumption was that accelerator arrays should be *as big as possible*, subject to the a) the resources defined by a given FPGA, b) the exact resource usage implied by a given member of a computation family, and c) the growth law specific to that family of applications. Given these, it becomes possible to state the array size in terms of one or more structural parameters n_i. The ideal implementation is defined by the relation shown in Equation 4.

$$\underset{N \mid V(N) \wedge \forall j : r^F_j \geq C_j(N, B)}{\arg\max} U(N)$$

Equation 4. Determining maximum sizes of computation

Equation 4 uses a familiar hardware design concept in an unfamiliar way. The familiar concept is, as shown in Figure 22, an array of hardware elements where array sizes are set by some set of structural parameters *N*. In traditional design, *N* is an input to the system. Here, however, the values of *N* emerge from other aspects of the system design. Vector *N* is not a set of constants to be plugged in, but a set of variables for which solution is sought.

Table 9 . Terms in resource constraint computation.

Term	*Meaning*	*Origin*
$N=(n_1, n_2, \ldots n_I)$	*Non-negative, integer-valued structural parameters n_i that describe the accelerator configuration*	*Family*
$V(N)$	*Predicate that tests whether$(n_1, n_2, \ldots n_I)$ meet architectural validity criteria, irrespective of the resource usage implied*	*Family*
$C_j(N, B)$	*Functions stating consumption of resource j for design parameters N and application-specific coefficients B*	*Family*
$U(N)$	*Utility of an accelerator described by design parameters N*	*Family*
$R^F=(r_1, r_2, \ldots r_J)^F$	*Integer amounts of resource j available in FPGA F*	*FPGA*
$B=(b_1, b_2, \ldots b_K)$	*Coefficients b_k giving application-specific resource utilization per repeatable element of the accelerator.*	*Member*

Table 9 gives the meaning of the symbols used in Figure 2. The *origin* of each term is the source that provides the required information: the choice of specific FPGA, the *family* of related applications, or the specific *member* of the application family for which an accelerator is to be realized.

Vector B identifies the amounts of logic resources used by each application-specific data type or function in a specific member of the application family. This vector should not be an explicit input to the design tools. Instead, the application-specific resource demands should be self evident, based on the actual functions and data type declarations used in a member of the application family.

Utility is a scoring function such that higher values indicate preferable accelerator configurations. If $U(N_1) = U(N_2) \wedge N_1 \neq N_2$, then configurations 1 and 2 are different but equally desirable; either could be chosen. Absolute values of the utility function have no significance. The ranks of utility values simply establish the ordering of more and less desirable solutions.

The validity predicate $V(N)$ captures the idea that some systems allow only certain sets of values for the structural parameters N. Validity constraints on structural parameters apply not only individual parameters, but to arbitrary relationships between sets of parameters. In Figure 23, for example, it is reasonable to apply the constraint $n_p \geq n_k$, i.e. that the image is at least as big as the convolution kernel. Predicate V does not verify coefficients B, the resource usage specific to any one member of the application family, since it is assumed that they are constructed correctly by the design tools and do not need to need further checking.

Equation 4 suggests that the most desirable accelerator is the largest one that fits the FPGA-specific resources, once application-specific usage demands and constraints are known. Maximization of U for configuration parameters N is, in general, a difficult problem, the exact solution of which is beyond scope of this discussion. Because realistic parameter sets N have modest cardinality and modest integer ranges, exhaustive search of the configuration space defined by V is assumed to be acceptable.

Without loss of generality, the U, C_j and V are assumed to be monotonic, although in slightly different senses. Let N and N' be sets of structural parameters, such that N' is identical to N in all positions except one, where $n_i < n_i'$. It is assumed that predicate V is monotonic in the sense that, if $V(N)$ is false, then $V(N')$ is false as well. This means that, after some value of n_i, all larger values (holding all other parameters constant) are also invalid. It is also assumed that $U(N') \geq U(N)$, i.e. that the value of an accelerator increases with at least weak monotonicity in all components (and subsets of components) of N. Functions C_j are assumed monotonic in the same sense as U, for any given application family member characterized by some fixed B.

These constraints are not strictly necessary for Equation 4 to be valid. There is intuitive appeal in the idea that larger n_i represent larger computation structures so have utility U at least as high, and in the idea that larger structures consume at least as much of each FPGA resource C_j. The real reason for monotonicity, however, is pragmatic.

Monotonic objective functions *U* are far easier to maximize than non-monotonic functions.

Table 10 . Resource constraint terms for sample application (Figure 25)

Term		***Meaning***
N		Tuple of architectural parameters
	n_p	Edge dimension of the square image array, in units of pixels
	n_k	Edge dimension of the square convolution array, number of coefficients in the convolution kernel.
$U(N)$	$n_k \times n_p$	Utility function. Desirability of the solution improves as either architectural parameter value increases
B		Vector of application-specific implementation parameters.
	b_p	Number of bits per image pixel
	b_s	Number of bits per convolution sum
	g_k	Number of gate resources needed to implement one cell of the square convolution array
	g_c	Constant representing fixed overhead for control logic, irrespective of the specifics of the accelerator.
$V(N)$	$n_p \geq n_k$	The image must be at least the size of the convolution kernel.
$C_j(N, B)$		Given by Equation 3
R^F		FPGA's resource vector: {logic units, RAM bits}

Monotonicity of C_j helps in limiting the number of configurations examined, since larger accelerators would consume at least as much of any resource, and would continue to violate resource constraints. Non-negativity of n_i and monotonicity of predicate V take the place of some alternative mechanism for setting lower and upper bounds on parameter values for searches across parameter space. As an example, consider Equation 4 when applied to the two-dimensional convolution of Figure 25. Table 10 summarizes the correspondence. Not knowing any particular image or convolution kernel in advance, the best accelerator is taken to be the one that handles the largest possible image and the largest possible convolution kernel, both assumed to be square.

B.2 Confounding factors in sizing and synthesis estimation

The premise that bigger computation arrays are necessarily better does not always hold. Some specific applications use strings or arrays of fixed size, so allocations above that size do not increase performance. In other specific cases, growth law assumptions do not match the physical facts of the problem. For example, the rigid molecule interaction case study assumes cubical arrays for holding the two molecules. Very few molecules actually have bounding boxes of 1:1:1 aspect ratio. Non-cubical arrays fit such cases more tightly, so fewer PEs in the array are allocated to empty padding.

Synthesis, placement, and routing also create performance problems when the application uses a large fraction of the FPGA's resources, especially when utilization exceeds 90%. Placement becomes difficult, because early, arbitrary placement decisions create unforeseeable constraints on later assignments of logic resources. Routing becomes increasingly difficult, as well. Localized congestion causes the router to create circuitous connections between logic elements. Excessively long paths in turn claim disproportionate amounts of routing resources, making later routing decisions yet harder. As a result, near-optimal assignments of logic and connectivity become increasingly harder to find as utilization increases. Wiring delays increase dramatically, often exceeding 2× the delay (i.e. 50% the speed) of similar circuits with lower utilization. Although problematic, this research has not encountered cases where increased parallelism (and utilization) has actually decreased performance due to clock speed degradation.

The choice of array size depends on the estimate of the arrays use of FPGA resources, commonly called *synthesis estimation*. For many reasons, synthesis estimation is a difficult and imprecise. It is clearly subject to many confounding effects, including:

· **Estimation uncertainty**. Resource estimation is an open problem, so any estimation functions C_j and coefficients B can only be approximate. For safety, they must be made conservative and thereby run the risk of under-utilizing resources.

· **Dependence on specific tools and FPGAs**. It is well known that different synthesis tools generate results that differ in the amount of logic resources required for implementing a given function. Even synthesis tool options and pragmas affect resource utilization.

· **Hidden scaling**. It is difficult for estimation functions C_j to properly capture size-dependent effects introduced during synthesis. For example, a large broadcast networks use long-distance routing resources (when available), but smaller configuration need use only short-range resources. Synthesis tools are known to add repeaters to large data distribution networks, claiming resources not explicitly allocated by the logic design. In principle, these can be quantified and added to functions C_j, but that may be infeasible in practice.

· **Inadequate hardware modeling**. Equation 4 is phrased in terms of the different resources $(r_1, r_2, \ldots r_J,)^F$, available on FPGA F. The problem is that different FPGA fabrics have different resource sets that can not always be treated as interchangeable. Programmable LUTs of different sizes are not directly comparable, for example, and different FPGAs are known to provide RAM blocks in different amounts and with different (length × width) configuration options.

· **Alternative implementations**. Different resources can often be used for implementing a given function. Lookup RAMs, random logic, or fixed-function blocks can all implement multiplication, depending application details. Suppose an array of multiplier-based PEs consumed all block multipliers in some FPGA. Depending on resource availability, more PEs could still be built using multipliers built from random logic or RAM tables. This creates estimation difficulties that are difficult to address.

- **Unacknowledged dependencies**. This discussion has assumed, for example, that an application family's number of summands is independent of the number of application-specific sum bits. The data path width specified by the number of sum bits certainly affects the number of summands possible for a given amount of logic. It is also true, however, that numbers of sum bits must be different to represent worst-case totals for different numbers of summands. The resource model in this example makes the simplifying assumption that the configuration parameters N have no affect on the application-specific coefficients B. Additional cross terms between N and B can be added to resource estimates C_j, however, to represent this kind of dependency.

Despite these problems, crude synthesis estimates are fairly easy to develop, adequate for typical sizing computations. Also, the details of synthesis estimation do not appear explicitly in Equation 4. Improved estimation techniques can be used as they become available, with no fundamental change to the sizing process. In the extreme, actual synthesis could be performed for each design alternative and actual utilization figures used instead of heuristic estimates, giving perfect precision of estimation. This is of theoretic interest, however, and unlikely to be part of any feasible solution.

Appendix C. SAMPLE APPLICATION: DOCKING

The computation core of the accelerator for solid molecule interaction is a generalized 3D correlation. This section presents the LAMP tool input for the computation core. The internal representation, LAMPML, is the XML format described in appendix A.1. Although convenient for machine processing, it presents a poor human interface. Convenience LAMP (or CLAMP), described in appendix A.3, is more readable, but requires translation into LAMPML. Because of the interactions with the underlying HDL, CLAMP can not represent all of the logic design. As a result, this section uses CLAMP where possible, but also uses LAMPML directly.

The core of the docking application consists of the following files:

· DockingApp.lamp: Top-level index information. This file collects all of the other files used for the docking application, including the bindings used for one instance of the application.

· FPGAresource.lamp: Abstract definitions of the FPGA resources. Concretions of this abstraction help with synthesis estimation, and define the quantities of logic resources available on any specific FPGA

· XC2VP.lamp: Partially specializes `FPGAresource` to represent the memory allocation logic common across all members of the Xilinx Virtex II Pro family.

· XC2VP70.lamp: Concrete resource definitions for the Xilinx Virtex II Pro VP70.

· XC2VP100.lamp: Concrete resource definitions for the Xilinx Virtex II Pro VP100. Any one application would use either this file, the `XC2VP70` file, or some other file specific to another FPGA, but no more than one of them in any given application.

· DockingBase.lamp: Abstract definitions used by the docking application.

· ScoreBase.lamp: First-level specialization of `DockingBase`. This defines concretions of the scoring data types and operations.

· KatchalskiKatzir.lamp: Second-level specialization of the `DockingBase` interface. This implements the logic specific to the Katchalski-Katzir scoring function [Kat93]. This, `ChenWengGSC`, and `ChenWengPSC` are all concretions of the `DockingBase` abstraction. Any one of these may be used, but only one in any given accelerator.

· ChenWengGSC.lamp: An alternative to KatchalskiKatzir.lamp, this implements the GSC scoring function defined by [Che02].

· ChenWengPSC.lamp: Another alternative to KatchalskiKatzir.lamp, this implements the PSC scoring function also defined in [Che02].

· ConfigWrapper.lamp: Top-level component. This does not add directly to the logic design, but performs the sizing operation that determines the largest computation arrays that the FPGA can support.

· Conv3D.lamp: Highest level component in the computation array

· Conv2D.lamp: Inner component of Conv3D

· Conv1D.lamp: Inner component of Conv2D

· MolCell.lamp: Unit cell of the correlation array

· FixedLenFifo.lamp: Definitions for the synchronous FIFO used within the correlation array.

· DPRAM.lamp: Defines a basic dual-ported RAM.

Throughout this discussion, the correlation operation used in the docking algorithm is referred to as a convolution. Correlation and convolution are, strictly speaking, distinct operations. In practice, however, they perform operations so nearly identical that the terms may be used interchangeably.

C.1 The docking model

The files listed above comprise the LAMP model for the correlation core of the docking application. Figure 26, below, relates those files to the model structure shown in Figure 22. In order to clarify the relationship to that figure, the name of each block is shown as a stereotype name, and the list of files that fill each role of the stereotype is shown as the block content.

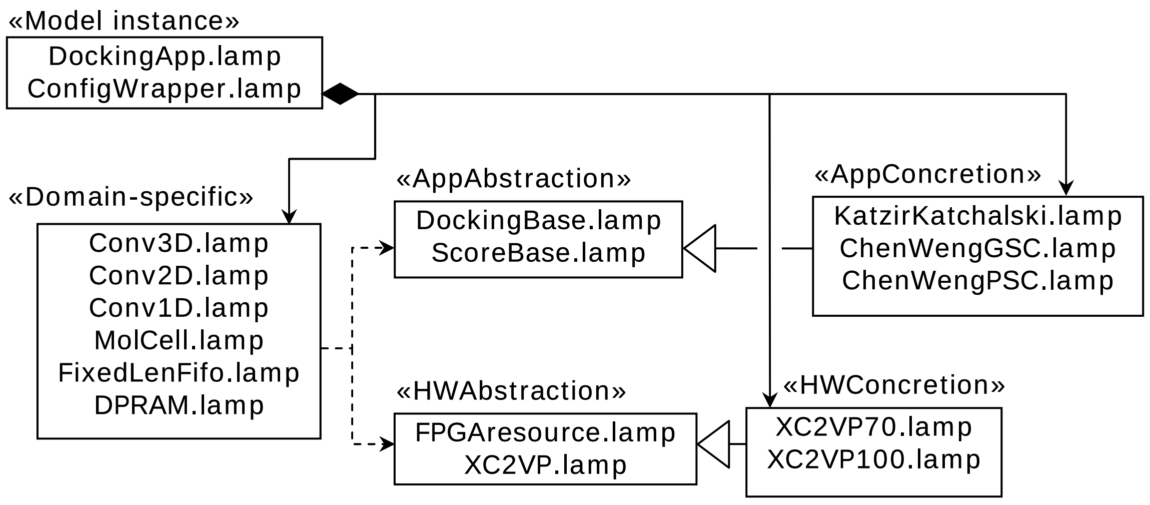


Figure 26. LAMP model for docking correlation core

There is always some amount of interpretation in assigning features of an application to roles defined by stereotypes, so other assignments could have been made, especially for classes that create partial specializations of abstractions (`ScoreBase` and `XC2VP`). Figure 26 includes the following elements, identified by stereotype name:

- **Model instance**: These are the top-level containers for all other parts of the application. The `DockingApp` file aggregates the other files into a whole. The `ConfigWrapper` file performs the sizing operations that decide on specific dimensions of the computation arrays.

- **Domain-specific**: The domain specific files implement logic that is shared across all members of the computation family. These files implement the logic that spans the domain of docking cores, but does not contain logic specific to any one member of that computation family.

· **HWAbstraction**: The FPGAresource class defines the logic resources available on any Xilinx FPGA. The `XC2VP` class partially specializes the abstract description to implement the memory allocation logic common to all Xilinx Virtex II Pro family members.

· **HWConcretion**: Each concretion describes the resources available within some particular FPGA. Only one of these is used in any actual accelerator, because each accelerator must be built for a specific FPGA platform.

· **AppAbstraction**: These files connect the domain-specific logic to the application-specific logic. They state the application abstractions that are used by the domain-specific modules, and that must be implemented by the modules specific to any one member of the family of applications. The `DockingBase` definitions are fully abstract. Definitions in `ScoreBase` create utilities for scoring arithmetic that are shared across many application concretions. It represents a partial specialization of `DockingBase`, and not the only scoring arithmetic that one could imagine.

· **AppConcretion**: Three examples have been provided, named according to the authors that proposed the specific scoring functions. All of them are subclasses of

ScoreBase and, by inheritance, DockingBase. Any of these may be used as the concrete scoring logic for an accelerator, but only one at a time may be used.

C.2 Docking files in detail

This section examines the CLAMP code that implements the correlation core of the docking application. LAMPML has been omitted, since CLAMP notation is far more readable. HDL code has also been omitted, because its implementation is straightforward and because the volume of it would clutter discussion of the novel LAMP constructs.

DockingApp.lamp

This is the only file in the application built form the integration subset of CLAMP. It maintains the list of files needed for a particular instance of the application.

For brevity and readability, comments and blank lines have been stripped from this version of the file. The actual file would generally include C++ comments, both // and /* */ forms, for the benefit of human maintainers. Comments and blank lines have been stripped from all the other files, as well, so no further mention will be made.

```
application Docking {
   useclass "DockingBase.xml";
   useclass "ScoreBase.xml";
   useclass "ChenWengPSC.xml";
   useclass "FPGAresource.xml";
   useclass "XC2VP.xml";
   useclass "XC2VP70.xml";
   useclass "XC2VP100.xml";
   use "MolCell.xml";
   use "DPRAMmodel.xml";
   use "FixedLenFifo.xml";
   use "Conv1D.xml";
   use "Conf2D.xml";
   use "Conv3D.xml";
   use "ConfigWrapper.xml";
   root ConfigWrapper rootComponent {
      class FpgaContent := XC2VP70,
      class AppClass := ChenWengPSC }
}
```

In a fully developed tool environment, this file would not normally be visible. It contains indexing information that collects the other design units in a single accelerator, and would be manipulated by the tool environment. One `application` file would represent the accelerator in its generic, unspecialized form. Abstract class definitions in that file would imply the definitions that the user must supply, along with the set of symbols imported by the `root` component.

Lines 2-6 name the class definitions that define the parts of the accelerator that are customized by each application. Lines 9-15 define the list of annotated HDL files used in this project. Lines 16-18 state which of the components is to be instantiated as the unique, top-level element, and associates concrete definitions with that model's abstractions. Although both XC2VP70 and XC2VP100 hardware definitions are available, only one of them at a time may be bound the FPGAcontent symbol of the root component.

FPGAresource.lamp

This file defines the FPGA's computing resources, in abstract form. It does not state the actual quantities specific to any one FPGA, but states the kinds of resources expected by synthesis estimation. In order to connect the descriptive text more closely to the features being described, the file has been interleaved with the commentary. Line numbers on each code fragment show its original position in the file.

```
1. class FPGAresource {
2.    const natural logicCells;
```

This specifies the number of logic elements available on an FPGA. The meaning of this amount may vary, but it is interpreted as slices in FPGAs in the Xilinx product family. This is an abstract constant. This interface item may be used in calculations, particularly sizing operations, but must be bound to a concrete class for the expression to be evaluated.

```
3.    const natural ramBlocks;
4.    const natural hardMultiply;
```

Likewise, these two values represent the number of block RAMs and dedicated multipliers in the Xilinx product family. This description of available resources would need to change, for example, when porting the application to a Stratix FPGA. Where the Xilinx product family has only one size of block RAM resources, the Stratix has three: 512b, 4Kb, and 512Kb, in various numbers.

```
5.    function natural nRams(   natural wordSize, natural nWords);
6. }
```

This is an abstract function definition. It specifies everything a programmer needs to know in order to use this function in calculations, but does not specify the way in which the function is evaluated. That must be provided in a concretion of this class before the function can actually be used. This function has no system-defined meaning. Within this application, however, it is used in synthesis estimation. An application may instantiate RAMs of many different sizes, larger or smaller than the FPGA's block RAM. If the application's logical RAM is larger than the FPGA's physical RAM, then several physical RAMs must be ganged to implement the one logical structure. This function examines the size of logical RAM requested, and returns the number of physical RAMs required for its implementation.

Line 6 closes the bracket opened on line 1, and ends the class definition.

XC2VP.lamp

This class represents the features specific to the Xilinx Viretx II Pro product family, but shared across all members of that family.

```
class XC2VP extends FPGAresource {
   const natural ramBits := 16*1024;
   const natural minWords := 512;
   function natural nRams(natural wordSize, natural nWords) {
      natural effWords := if nWords > minWords
            then nWords else minWords;
      return (wordSize * effWords + ramBits-1) / ramBits
   };
}
```

The `ramBits` declaration does not match any symbol in the superclass. As a result, it creates a new interface item in `XC2VP` that was not present in `FPGAresource`. This interface item is not accessible when `XC2VP` is accessed through `FPGAresource`, because it did not exist at that level of inheritance. It is, however, available internally to the class and to any client that accesses the `XC2VP` interface explicitly.

This concrete definition of function `nRams` over-rides the abstract definition in `FPGAresource`, so it can be used anywhere that expression evaluation is required. The logic of this function funds the number of bits requested by the given numbers of words and bits per word. This is not completely accurate, however. The Virtex II Pro block RAMs may be configured as 16K×1 (words × bits), 8K×2, 4K×4, 2K×9, 1K×18, or 512×36. Logical RAM structures may be built by ganging physical RAMs, but each of the ganged RAMs must have one of those configurations. The exact allocation function, however, is non-trivial, and specific configurations may be handled best by several RAMs in a combination of configurations. This approximation is adequate for current needs, however, and can easily be replaced by any other expression the developer chooses.

The `XC2VP` class provides a concrete definition only for the `nRams` symbol. It inherits abstract definitions for all other symbols from `FPGAresource`, so those symbols have abstract definitions in this class as well. Because not all of its symbols, including inherited ones, have concrete definitions, this can not be used by itself in sizing calculations.

XC2VP70.lamp

This is a subclass of `XC2VP` and, by inheritance of `FPGAresource`. It specifies exact values for the constants left undefined in those classes file. Because this fully defines all of the interface items in `FPGAresource`, or inherits concrete definitions, it can be used for expression evaluation.

```
class XC2VP70 extends XC2VP {
    const natural logicCells := 66176;
3.    const natural ramBlocks := 328;
4.    const natural hardMultiply := 328;
5. }
```

Line 1 simply opens the class definition, and states that XC2VP70 is a subclass of XC2VP, the definitions that are partially specialized for the Xilinx Virtex II Pro product family. This does not need to define a concrete nRams function because it inherits the function that's generic across this FPGA product line

Constant values were transcribed from the manufacturer's data sheet for this product. It is not explicit in these const declarations, but all of them correspond to abstract definitions in FPGAresource. As a result, these concrete declarations over-ride the superclass' abstract definitions.

XC2VP100.lamp

This is also a subclass of XC2VP and, by inheritance for FPGAresource. It just fills in values from the FPGA vendor's data sheet that match the meaning of the abstract constants defined in FPGAresource. Because this concretely defines or inherits concrete definitions of all interface elements in FPGAresource, it can be used in system sizing calculations. This is an alternative to the XC2VP70 class – only one of these definitions would be used in any actual accelerator configuration.

```
6.    class XC2VP100 extends XC2VP {
7.       const natural logicCells := 99216;
```

```
        const natural ramBlocks := 444;
        const natural hardMultiply := 444;
    }
```

DockingBase.lamp

This class defines the abstractions for the docking application. Once these abstract definitions are given concrete meanings, the docking application can be synthesized.

```
1. class DockingBase {
2.    typedef scoreType;
3.    function scoreType makeScore(integer a);
4.    function scoreType addScore(scoreType a, scoreType b);
5.    function scoreType
6.            summarizeScores(scoreType a, scoreType b);
7.    const scoreType zeroScore;
8.    const scoreType summaryInit;
```

These declarations have to do with score computation. Line 2 gives an abstract definition of the score type. At this level of abstraction, it has only identity, and no other visible features. Line 3 provides an abstraction for a data conversion function, allowing positive or negative integer values to be cast as score values. Line 4 provides an addition operator. Line 5 is used in the filtering operation that reduces the total volume of scoring data. The remaining two constants are used as initialization values, for score summation and for summarization respectively.

```
9.      typedef cellType;
10.     function scoreType
11.          cellScore(cellType largeMol, cellType smallMol);
12.     const cellType emptyLg;
13.  };
```

These declarations have to do with the voxel values used to represent the two molecules. The `cellType` declaration just says that some data type definition will hold a cell of one molecule or the other. The `cellScore` function represents the arbitrary function *F* of Equation 2. The generalized correlation is not a sum-of-products, but a sum-of-F, summed over this function. The constant declaration of line 12 defines a value representing empty space, used for padding in 3-axis rotations.

ScoreBase.lamp

This class is a partial concretion of the `DockingBase` class. It implements common logic for handling score values, in a way that can be reused across many different models of chemical phenomena.

```
class ScoreBase extends DockingBase {
   const natural scoreBits := 10;
   typedef scoreVal integer[scoreBits];
   typedef scoreType {
      scoreVal value,
      boolean overflow };
```

The `scoreType` definition over-rides the abstract definition in the superclass, `DockingBase`. Enough significance bits must be allocated so that credibly large positive or negative values can be handled without overflow, but over-allocation would waste resources that could have been put to use in creating additional processing elements. If the range of signed values were not adequate, two's complement overflow would turn excessively positive values into negative ones, and vice versa. Overflow would invert the sense of the result, clearly not a desirable outcome. Saturating arithmetic could have been used to clamp results to the most positive or negative value. This has the disadvantages of losing knowledge of the overflow condition, which can be important to the analysis, and of requiring more hardware than the scheme chosen. In the event of overflow, the current scheme records that fact in a "sticky" flag. From that point forward, the fact that overflow occurred is preserved, no matter what operations occur later. The significance bits have no meaning in that case, except for the sign bit.

```
function scoreType makeScore(scoreVal a) {
   return new scoreType(
      overflow := false,
         value := a )
   };
```

This function converts an integer value into a score. Note that this function does not have exactly the same signature as the `DockingBase` function that it over-rides. The integer parameter here is has a restricted range, where it was unrestricted in the superclass. It does, however, have compatible type, so the over-ride is valid.

```
   typedef additionTemp integer[scoreBits+1];
   function scoreType addScore(scoreType a, scoreType b ) {
      additionTemp tmpSum := a.value + b.value;
      boolean oflo :=
         tmpSum[scoreBits:1] != tmpSum[scoreBits-1:1];
      boolean negOut := if a.overflow then a.value < 0
         elsif b.overflow then             b.value < 0
         else                              tmpSum < 0;
      natural signOut := if negOut then 1 << (scoreBits-1)
         else 0;
```

```
22.         return new scoreType(
23.            overflow := a.overflow || b.overflow || oflo,
24.            value := signOut | tmpSum[0:scoreBits-2] )
25.     };
```

This function adds two score values, taking possible overflow into account. Line 12 defines a computing value, with additional bits to allow detection of overflow. Line 14 determines the sum, using the temporary register with extended precision. Lines 15-16 use basic facts of two's complement arithmetic to examine two one-bit fields of the sum and determine whether magnitude of the sum is too large to fit the significand of the score. Lines 17-21 determine the sign of the output, either by propagating an input overflow or by reporting the sum's actual sign. In the case where operands `a` and `b` both report overflow and differ in sign, the result arbitrarily takes the sign of operand `a`. Summation of opposed overflows can not have a meaningful value; this approach at least records the fact of overflow. Lines 22-24 create the result. Line 23 reports overflow, whether propagated or newly detected, using a logical OR of all overflow conditions. Line 24 reports the sum with its proper sign, using a bitwise OR operation. Overflow conditions make the significand irrelevant, but preserve the meaningful values for the sign of the result.

```
   function scoreType summarizeScores(scoreType a,
                                      scoreType b)
   {
      boolean pickA := if a.overflow  then a.value >= 0
         elsIf b.overflow             then b.value < 0
         else                         a.value > b.value;
      return if pickA then a else b
   };
```

Many alternative ways exist for combining multiple score values into one summary value. This implements the familiar “max” function, with cases that handle overflow conditions. Positive overflow beats (or ties) any other value, negative overflow loses to (or ties) any other value, and all other cases depend on straightforward inequality testing.

```
   const scoreType zeroValue := makeScore(0);
   const scoreType summaryInit := new scoreType(
      overflow := true, value := -1);
};
```

These are straightforward definitions of values defined by the `DockingBase` superclass. The value for initializing a score summary is chosen to match the summary function implemented: the most negative possible value, compared with any other in a “max” operation, reports that other value. If the summary had been based on sums of scores, it would probably have been zero, or other value according to the application logic.

KatchalskiKatzir.lamp

The first widely recognized use of correlation for docking [Kat93] implemented a simple scoring function: a reward for overlap of surface regions, a penalty for collision of interior regions, and no score for any other interaction, including a complete miss. Because of the implementation technologies available, it was most efficient for those researchers to force their logic into a product form, so that efficient transform-based correlation could be used. They encoded molecule volume elements as complex numbers. Interior voxels were positive imaginary values, surface voxels were positive real values, and the real part of a product would yield a negative score for collisions or a positive score for surface contact. This implementation uses logical operations instead of complex multiplication to determine the kind of interaction, allowing a compact voxel representation and a simple scoring function.

```
class KatchalskiKatzir extends ScoreBase {
    typedef cellType {
        boolean isInterior,
        boolean isSurface };
```

```
const cellType emptyCell := new cellType(
   isInterior := false, isSurface := false );
```

Voxel values may be of any type. Here, the voxel (cell) representation is the most direct possible statement of the significant voxel state. An empty cell, exterior to a molecule, is neither interior to it nor in its surface region.

```
const scoreType collisionScore := makeScore(-10);
const scoreType touchScore := makeScore(3);
```

These values represent the collision and surface contact scores directly. The values are arbitrary. Even if they do not match the original Katchalski-Katzir values exactly, they encode the same logic and can readily be modified to any desired values.

```
function scoreType cellScore(
   cellType largeMol, cellType smallMol)
{
   return if largeMol.isInterior && smallMol.isInterior
            then     collisionScore
         elsif largeMol.isSurface && smallMol.isSurface
            then     touchScore
         else        zeroScore
   };
};
```

This function implements the logic for scoring any one cell-cell overlap. Again, it does not use the original phrasing of the correlation – that would have been an arithmetic product of complex values. The net result is the same, though, and readily implemented on FPGA logic.

ChenWengGSC.lamp

Other correlation-based scoring algorithms have been developed, including the Grid-based Shape Complementarity (GSC) and the superior Pairwise Shape Complementarity (PSC) scoring functions compared by Chen and Weng [Che02]. The following example implements the logic of the GSC scoring function. As with the Katchalski-Katzir, this example replaces arithmetic operations with logical tests, allowing efficient FPGA-based implementation. The original transform-based algorithm required the voxel scoring functions to be sums of arithmetic products. It is possible that, by making the logic of the molecule interaction tests more visible, that variants may be easier to implement, including variants that would be difficult to phrase as complex products.

```
class ChenWengGSC extends ScoreBase {
    typedef cellType integer[2];
    const cellType SolvAccessible := 0;
    const cellType SolvExcluding := 1;
    const cellType Core := 2;
    const cellType emptySpace := 3;
```

The basic logic of the GSC scoring function divides the 3D molecule structure into categorical values describing the molecule content at each grid point. Like the Katchalski-Katzir algorithm, this distinguishes empty, interior, and surface regions of each molecule. Surface regions, however, are distinguished by their solvent interaction properties, giving a slightly richer representation of the underlying chemistry.

```
        const scoreType SE_SA := makeScore(1);
        const scoreType SE_CSE := makeScore(-9);
        const scoreType Core_SA := makeScore(-9);
        const scoreType Core_CSE := makeScore(-81);
```

Scores for each voxel pair depending on whether the interacting voxels are solvent-accessible (SA), solvent-excluding (SE), or core regions. In some cases, no distinction is made between SE and core, abbreviated as CSE.

```
function scoreType cellScore(cellType Lg,cellType Sm){
   return
      if Lg = Core then
         if Sm = SolvAccessible         then Core_SA
         elsif Sm = emptySpace          then scoreZero
         else                           Core_CSE
      elsif Lg = SolvExcluding then
         if Sm = SolvAccessible         then SE_CSE
         elsif Sm = emptySpace          then scoreZero
         else                           SE_CSE
      else                              scoreZero
   };
};
```

This implementation of `cellScore` replaces the linear complex functions of the original problem statement with logical tests that achieve the same net result. The encoding based on complex values was clever, but the different kinds of interactions seem easier to identify in this phrasing.

ChenWengPSC.lamp

The same paper that proposed the GSC scoring function compared it to PSC. This file implements the logic of the PSC algorithm

```
class ChenWengPSC extends ScoreBase {
   typedef LgCell extends cellType{
      boolean solvExcl,
      boolean isCore,
      boolean hasNeighbor };
```

This algorithm has the distinctive feature that the voxel representations in the two molecule representations differ. This demonstrates the flexibility of the inheritance constructs in the LAMP design language. It also demonstrates additional opportunities for compact representations, possibly allowing higher degrees of parallelism. This distinguishes core and solvent-excluding surface regions, as in the GSC algorithm, and replaces the "solvent-accessible" term with one that looks for neighboring atoms.

```
   typedef nbrCount integer[5];
```

```
typedef SmCell extends cellType{
   boolean solvExcl,
   boolean isCore,
   nbrCount nbrs };
```

The smaller of the two molecules distinguishes core and solvent-excluding surface regions. It also counts the number of neighboring voxels, representing the belief that higher numbers of neighbors to any locale increase the strength of the interaction.

```
   const natural SE_SE := -9;
   const natural SE_Core := -27;
   const natural Core_Core := -81;
```

These scoring values have much the same meaning as in the GSC scoring system.

```
   function scoreType cellScore(LgCell largeMol,
                                SmCell smallMol) {
      natural otherScore :=
         if smallMol.isCore then
            if largeMol.isCore       then Core_Core
            elsif largeMol.solvExcl  then SE_Core
            else                          0
```

```
21.            elsif smallMol.solvExcl then
22.               if largeMol.isCore       then SE_Core
23.               elsif largeMol.solvExcl  then SE_SE
24.               else                     0
25.            else                        0;
26.         natural nbrScore := if largeMol.hasNeighbor
27.            then smallMol.nbrs
28.            else 0;
29.         return makeScore(nbrScore + otherScore);
30.      };
31.   };
```

Lines 16-25 work much the same way as in the GSC algorithm. The difference is in lines 26-28. This replaces scoring of the "solvent-accessible" voxel pairs with a reward proportional to the number of neighboring voxels in the large molecule, if the current voxel of the small molecule is sensitive to its neighbors at all. This is the term that makes use of the asymmetric voxel types, the boolean in the small voxel and the value count in the larger one.

The choice of which value goes where is based on the different computing resources consumed by the arrays that hold the large and small molecules. Logic resources hold the small molecule values within the computation array, but on-chip RAM holds the larger molecule. The incremental logic required for the few extra voxel bits makes little difference to the total logic size in the scoring cell, but would have increased the number of RAM bits per voxel from three to seven. This assignment of small and large molecules generally allows the largest sizes for both molecules.

ConfigWrapper.lamp

The configuration wrapper is the first of the CLAMP entity definitions in this example. It is important to note that these files correspond to annotated HDL text, but do not themselves contain the HDL text. These files are used to generate LAMPML output in XML format. That XML markup is to be integrated by hand into the HDL text. The CLAMP text is more readable, however, so it is used for descriptive purposes.

Although this file is nominally for HDL annotation, it does not contribute any synthesizable logic to the application. Instead, it implements the function for sizing the computation arrays to the amounts of logic resources in the current FPGA. This doe not implement the full form of the optimization stated in Equation 4. Instead, it determines the largest computation array, i.e. the largest size of the small molecule grid that can be contained in the available logic. Then, it determines the largest size of larger molecule that can be held within the RAM resources at hand.

```
entity ConfigWrapper {
    symbol entName := uniqueID(configure);
```

```
import class AppClass extends DockingBase;
import class FpgaContent extends FPGAresource;
```

These are the only two items imported into the sizing logic: the application-specific functions and data types, and the statement of FPGA resources. Note that there is no input that explicitly specifies the size of either array.

```
typedef configChoice {
   natural convBlockSize,
   natural fifoSize };
```

This record states the choices made for a system configuration: the edge dimension of the convolution computation block, and the size of FIFO used for holding temporary results. Although the computation array can accept different sizes in its X, Y, and Z dimensions, this makes the simplifying assumption that array sizes along all axes are the same, i.e. that the array is a cube.

```
instance Conv3D sizing {
   class AppClass := AppClass,
      class FpgaContent := FpgaContent,
      xCells := 1,  yCells := 1,     zCells := 1,
      xFifoLen := 1,                 yFifoRows := 1 };
```

The `sizing` instance creates a dummy entity description. The dummy has all the logical internal structure of the `Conv3D` array (described below), but the constants describing feature sizes are chosen arbitrarily. Only the class bindings matter, because they define the resource utilization for any chosen array size. Although this creates a logical instance of Conv3D bound to the parameters shown, no synthesizable instance of `sizing` is ever created. Instead, functions defined in this logical instance are used for computing resource demands.

Lines 9 and 10 have the same symbol name on the right and left sides of the assignment. Within each line, the two symbols are distinct. In line 9, the right-hand occurrence of `ApplClass` is resolved in the scope of the current `entityDef`. It refers to the `import` statement at line 3, and represents whatever class is bound to that `import` symbol when a logical instance of this entity is created. The left-hand occurrence of `AppClass` refers to the `import` symbol in the entity for which the logical instance is being created, an instance named `sizing` of entity `Conv3D`. The proper interpretation for line 9, then, is that an `import` symbol defined in `Conv3D` and named `AppClass` binds to the same class reference as a symbol coincidentally named `AppClass` and defined in `ConfigWrapper.`

```
13.     function configChoice confBlocks(configChoice confTry) {
14.        natural bumpConv := confTry.convBlockSize + 1;
15.        natural logicUsage := sizing.synthLogic(
16.           bumpConv, bumpConv, bumpConv,
17.           confTry.fifoSize, confTry.fifoSize);
```

```
   natural RAMusage := sizing.synthRAM(
      bumpConv, bumpConv, bumpConv,
      confTry.fifoSize, confTry.fifoSize);
   boolean logicFits :=
      logicUsage <= FpgaContent.logicCells;
   boolean RAMover := RAMusage > FpgaContent.ramBlocks;
   return if RAMover then confTry
      elsif logicFits then configBlocks(new configChoice(
         convBlockSize := bumpConv,
         ifoSize := confTry.fifoSize))
      else confRam(confTry)
};
```

Function `confBlocks` performs the first of two sizing steps, sizing of the computation array. Parameter `confTry` is assumed to be some configuration that has already been found to be acceptable. The goal of this function is to try larger arrays and report the largest allowed. If no larger array is allowed, then `confTry` is used as the system configuration. Recursion generates successively larger configurations until one is rejected, then the last one not rejected is used.

LAMPML and CLAMP do not contain looping or conditional execution statements. Instead, they use recursion and conditional expression evaluation to achieve the same results. Since the recursive functions complete execution before synthesis begins, and long before execution of the accelerator logic, they do not cause any problems for processing of the underlying HDL.

Line 14 starts evaluation of the larger configuration. It increments the size of the proposed computation array. Lines 15-17 evaluate the amount of logic that would be needed by `sizing`, a `Conv3D` component, for implementing an array of that size and of `confTry`'s given buffer size. Logic and memory are the two resources considered in this estimation, and both may, at least in principle, change for any change of parameter values. As a result, the amount of RAM claimed by an array of proposed size must also be checked. Lines 18-20 determine the RAM utilization for the proposed array size.

Lines 21-22 perform simple tests to determine whether resource demands of the proposed array sizes fit within the resources of the FPGA. These tests are phrased in terms of class `FpgaContent`, which is imported (i.e. an input) to this entity. That class has no definition in this entity, except that it is defined to export the interface specified in abstract class `FPGAresource`. Depending on the bindings assigned by the entity that instantiates this class, any FPGA's resources could have been used. Different resource limits could create different results for these inequalities, allowing different array sizes. This entity is instantiated by the root statement in `DockingApp.lamp`, which binds a concrete class definition to `FpgaContent`. In this example, resource constraints for the Xilinx Virtex II Pro VP70 are used.

Porting this application to a larger array size requires a different binding of `FpgaContent`, creating a different set of resource limits. This logic works the same way no matter what the actual amount of those limits, so in independent of FPGA type. Choice of actual FPGA is made in `DockingApp.lamp`, the file that integrates the full set of knowledge required for any one accelerator.

Lines 23-26 evaluate the outcome of the proposed configuration, given that input `confTry` is assumed to be acceptable. There are three possible outcomes to this test. First (line 23), the newly proposed configuration may be found to exceed available amounts of RAM. The input is assumed to have used acceptable amounts of RAM, so that configuration is used, and no further claims for RAM resources are proposed. The second possible outcome is that RAM and logic utilization are both within limits, so addition claims for logic resources (line 24-25) will be tried. The third possible outcome (line 26) is that logic resources can not support a larger array, so the RAM-based FIFO arrays will be sized by function `confRAM`.

```
30.     function configChoice confRAM(configChoice confTry) {
31.        natural bumpFifo := confTry.fifoSize + 1;
32.        natural logicUsage := sizing.synthLogic(
33.           confTry.convBlockSize, confTry.convBlockSize,
34.           confTry.convBlockSize, bumpFifo, bumpFifo);
35.        natural RAMusage := sizing.synthRAM(
36.           confTry.convBlockSize, confTry.convBlockSize,
37.           confTry.convBlockSize, bumpFifo, bumpFifo);
38.        boolean blewLimit :=
39.              (logicUsage >= FpgaContent.logicCells)
40.           || (RAMusage >= FpgaContent.ramBlocks);
41.        return if blewLimit then confTry
42.           else confRAM(new configChoice(
43.              convBlockSize := confTry.convBlockSize,
44.              fifoSize := bumpFifo))
45.     };
```

Function `confRAM` implements the second step in this application's sizing logic. Once a size is chosen for a computation array, it determines the largest amount of buffer space possible within the RAM limitations. This function's logic resembles that in `confBlocks`. It assumes that the input describes a configuration with acceptable logic and RAM utilization. It then increments the configuration parameter known to claim RAM resources (line 31, and determines the amounts of RAM and logic claimed by the newly proposed configuration (lines 32-37).

Since the computation array is already assumed to be the largest possible, it is only necessary to choose between using the known-good RAM configuration (line 41) and trying a larger one lines 42-44).

Referring to Equation 4, it should be clear how these functions capture all terms in the maximization problem. The result of utility function U is implicit. It favors larger computation array sizes. When two computation arrays have the same size, a tie-breaking rule favors the larger RAM array. Tuple `configChoice` implements array N, the configuration parameters. Class `FpgaContent` implements the resource limits R, and includes auxiliary logic related to computing the resource limits. Cost functions C are implemented explicitly in `Conv3D` as `synthLogic` and `synthRAM`. Class `AppClass` contains all needed information about the resource demands implied by a particular choice of data types and application-specific functions. It implicitly contains the information of vector B, the sizing parameters specific to any one member of the accelerator family. Maximization is explicitly performed these two functions. The predicate V that tests architectural validity does not appear explicitly, but is silently taken to be true for all positive values of `convBlockSize` and `fifoSize`.

```
    configChoice sysConfig := confBlocks(new configChoice(
        convBlockSize := 1, fifoSize := 1));
```

The `sysConfig` value contains the system configuration parameters that result from maximizing resource utilization as described above. It starts with a dummy configuration of the smallest possible size, as an initial value for the recursive search of configuration space.

```
   instance Conv3D convArray {
      class AppClass := AppClass,
      class FpgaContent := FpgaContent,
      xCells := sysConfig.convBlockSize,
      yCells := sysConfig.convBlockSize,
      zCells := sysConfig.convBlockSize,
      xFifoLen := sysConfig.fifoSize,
      yFifoRows := sysConfig.fifoSize };
}
```

Once the system configuration parameters have been determined, it is possible to create a system with that configuration. Again referring to Equation 4, it is easy to see how this combines information from the all of the sources identified in section B.1: the application family (`Conv3D`), the specific application family member (`AppClass`), the FPGA resource specifics (`FpgaContent`), and the sizing parameters that derive from those three.

As a simplification, the *X*, *Y*, and *Z* axes of the convolution array are assigned the same sizes, as are the *X* and *Y* axes of the buffers that hold partial correlation sums. There is no fundamental reason for this other than convenience – another implementation might have reason to used different sizes in either case.

Conv3D.lamp

As with all CLAMP entity definitions, this expresses only the LAMP logic. In use, it is translated into XML-based LAMPML, then integrated by hand with the HDL application logic. This file shows only the LAMP logic, without the HDL content.

The rest of the CLAMP files in this example follow the same general structure as this one, so will be described in generally less detail. This set of declarations serves two major purposes: it defines the synthesizable entities that form the top level of the computation array, and it performs one level of synthesis estimation. Each of these is described in detail at the corresponding part of the file.

```
entity Conv3D {
    symbol entName := uniqueID(conv3D_);
```

The entity's name symbol (`entName`) is exported from CLAMP and LAMPML into the HDL used for defining the structure of the computation array.

```
    import class AppClass extends DockingBase;
    import class FpgaContent extends FPGAresource;
```

These declarations import the application-specific details of the accelerator and the definitions relating to resource availability in the FPGA.

```
import natural xCells;
import natural yCells;
import natural zCells;
import natural xFifoLen;
import natural yFifoRows;
```

These values import the dimensions of the computation array and of the buffer arrays. The values have meanings illustrated in Figure 27. The FIFOs hold intermediate scoring results, and depend on the size of the large molecule. The computation array holds the small molecule voxel values, one per computation cell, and performs the generalized correlation sum.

```
natural xyFIFOwords := (xFifoLen + xSize) * yFifoRows;
```

This value computes the number of words needed for one plane FIFO.

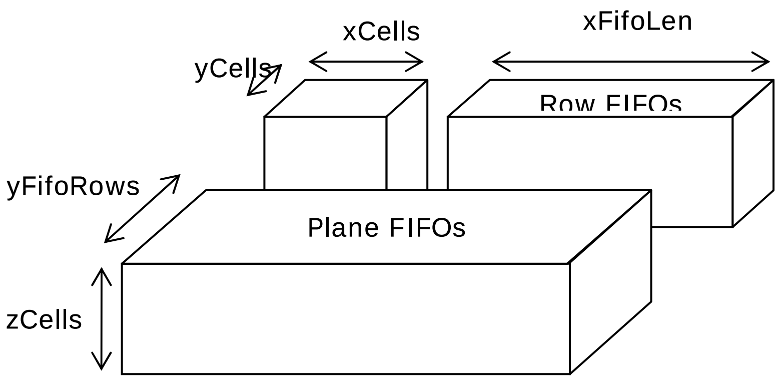


Figure 27. Import values for Conv3D correlation array

```
instance FixedLenFIFO planeBuffer {
   class dataType := AppClass.scoreType,
   class FpgaContent := FpgaContent,
   maxLag := xyFIFOwords,
   blank := AppClass.zeroScore };
```

This creates the FIFO entity definition to be used for the plane buffer. It does not actually instantiate the logic of the FIFO. Instead, it creates a compatible definition that can be instantiated by the HDL code. The `dataType` binding defines the type of the data word to hold in the FIFO. Although this is a `typeDef` rather than a class, the same syntax is used for both. The `FpgaContent` assignment appears to have the same symbol on both sides of the assignment operator. The two occurrences of `FpgaContent` are in fact different. The left hand symbol is an import in `FixedLenFIFO`, and is resolved in that symbol context. The right hand side is resolved in the `Conv3D` context. The intent of this binding is to pass the `FpgaContent` definition recursively down the hierarchy of LAMP definitions.

```
instance Conv2D convPlane {
   class AppClass := AppClass,
   class FpgaContent := FpgaContent,
   xCells := xCells,
20.         yCells := yCells,
21.         xFifoLen := xFifoLen };
```

The convPlane declaration creates a logical LAMP entity that can be instantiated by the HDL code. It follows the same logic as the instance statement at line 11, so its class bindings recursively propagate the symbol definitions of the current context into the new Conv2D context.

```
22.      function natural synthLogic(
23.         natural xSize, natural ySize, natural zSize,
24.         natural fifoXlen, natural fifoYrows)
25.      {
26.         natural planeBufSize :=
27.            (fifoXlen + xSize) * fifoYrows;
28.         natural planeLogic :=
29.            convPlane.synthLogic(xSize, ySize, xFifoLen) +
30.            planeBuffer.synthLogic(planeBufSize);
31.         return zSize * planeLogic
32.      };
```

```
33.     function natural synthRAM(
34.        natural xSize, natural ySize, natural zSize,
35.        natural fifoXlen, natural fifoYrows)
36.     {
37.        natural planeBufSize :=
38.           (fifoXlen + xSize) * fifoYrows;
39.        natural planeRAM :=
40.           convPlane.synthRAM(xSize, ySize, xFifoLen) +
41.           planeBuffer.synthRAM(planeBufSize);
42.        return zSize * planeRAM
43.     };
44.  }
```

These two functions, `synthLogic` and `synthRAM`, perform synthesis estimation for this component, based on the designer's knowledge of its internal structure. The two work much the same way. Each one represents the amount of space needed for the FIFO (lines 26-27 and 37-38). Based on that, each one computes the amount of resource, logic or RAM, needed by the 2D plane of convolution logic (lines 29 and 40) plus a plane buffer (lines 30 and 41). The result is taken to be that amount of resource, times the number of planes.

Note that the imported symbols are accessible to these function bodies, but the import values (`xCells`, etc) are not used in these estimates. These functions use only parameter values, so they can compute resource estimates for any size of computation array, not just the sizes bound to import values for this particular instance. That lets the logic in `ConfigWrapper` examine resource utilization for many hypothetical configurations, not just the ones that have been instantiated.

Conv2D.lamp

This entity definition follows the same general pattern seen in Conv3D.lamp .

```
entity Conv2D {
    symbol entName := uniqueID(conv2D_);
    import class AppClass extends DockingBase;
    import class FpgaContent extends FPGAresource;
    import natural xCells;
    import natural yCells;
    import natural xFifoLen;
```

Class imports have the same meaning as in `Conv3D`. Value imports do too, except that `Conv2D` has a proper subset of `Conv3D`'s value imports.

```
    instance FixedLenFIFO rowBuffer {
        class dataType := AppClass.scoreType,
            class FpgaContent := FpgaContent,
            maxLag := xyFIFOwords,
            blank := AppClass.zeroScore };
```

```
instance Conv1D convRow{
   class AppClass := AppClass,
   class FpgaContent := FpgaContent,
   xCells := xCells };
```

Definitions of inner entities follow the same pattern as in Conv3D, except that the plane is built from one-dimensional computation and FIFO arrays.

```
function natural synthLogic(
   natural xSize, natural ySize, natural fifoXlen)
{
   natural rowLogic := convRow.synthLogic(xSize) +
      rowBuffer.synthLogic(fifoXlen);
   return 5 + ySize * rowLogic +
      synthSize(AppClass.cellType)
};
```

Synthesis estimation for logic resources is slightly more complex than for Conv3D. The ySize * rowLogic term propagates estimates from inner components to the current level. The constant 5 represents allocation for a control register in the HDL code that is not visible to the LAMP code.

The synthSize expression represents the amount of logic needed for a staging buffer that helps reduce fanout delays. The buffer itself is not represented in the LAMP code, so can not be added automatically to the synthesis estimate. The data type of the value held in that buffer is, however, derived from LAMP code. The register holds a `AppClass.cellType` value, and this expression determines the number of logic elements needed for enough one-bit registers to hold that value. Referring to Equation 4, this represents inclusion of application-specific information from vector B into the estimated demand for logic resources.

```
   function natural synthRAM(
      natural xSize, natural ySize, natural fifoXlen)
   {
      natural rowRAM := convRow.synthRAM(xSize)
         + rowBuffer.synthRAM(fifoXlen);
      return ySize * rowRAM
   };
}
```

RAM allocation for this entity follows the pattern set earlier. This does not have additional terms, as `synthLogic` does, because there are no additional RAM allocations beyond the row buffers.

Conv1D.lamp

This defines the one-dimensional convolution array used as component within `Conv2D`. It follows the same general pattern seen in `Conv3D` and `Conv2D`, but does not need direct access to the FPGA-specific definitions. The one-dimensional row consists of `MolCell` components, so an instance definition is created for those inner blocks.

```
entity Conv1D {
   symbol entName := uniqueID(conv1D_);
   import class AppClass extends DockingBase;
   import natural nCells;
   instance MolCell convCell {
      class AppClass:= AppClass };
   function natural synthLogic(natural numCells)
      { return numCells * convCell.synthLogic };
   function natural synthRAM(natural numCells)
         { return numCells * convCell.synthRAM };
   }
```

Another small difference also sets this apart from the previous entity declarations. The synthesis estimation functions depend on estimates from an inner component (`convCell`) as before. In this case, however, the inner component exports synthesis estimates as symbol values rather than functions. The “variable” part of a `MolCell` element is only in the data types and function logic that it uses, defined by `AppClass`, not in any imported value settings.

MolCell.lamp

The MolCell component is the leaf component in the convolution array.

```
entity MolCell {
   symbol entName := uniqueID(molCell_);
   import class AppClass extends DockingBase;
   natural synthLogic :=
      synthSize(AppClass.cellType) +
      synthSize(AppClass.scoreType) +
      synthSize(AppClass.cellScore,
         AppClass.cellType, AppClass.cellType) +
      synthSize(AppClass.addScore,
            AppClass.scoreType, AppClass.scoreType);
      natural synthRAM := 0;
   }
```

This entity is the leaf structure in the convolution array. It has no inner components under LAMP management, so does not contain any `instance` declarations.

The resource estimation function does not depend on any outside information other than that in the binding to the `AppClass` import. As a result, the `synthLogic` and `synthRAM` resource estimates are phrased as constant values rather than functions. The `synthRAM` estimate should be self-explanatory. The `synthLogic` expression, however, requires knowledge of the HDL implementation of this entity. The four terms, respectively, represent a register holding a voxel value, a register holding a scoring sum, a function for scoring voxel values, and a function for summing score values.

FixedLenFifo.lamp

The fixed length FIFO holds each input for some number of number of cycles, then releases it, a form of digital delay line. The amount of delay can be adjusted during a setup operation, but is held fixed for the duration of any computation. Because of the expected lengths of delays, the FIFO is implemented in terms of block RAM resources.

```
entity FixedLenFIFO {
    symbol entName := uniqueID(flFIFO_);
    import class FpgaContent extends FPGAresource;
    import class dataType extends baseType;
    import dataType blank;
    import natural maxLag;
```

The FIFO implementation depends on RAM allocation (described in terms given by `FpgaContent`), as well as a length limit, `maxLag`. There is no explicit word size, because that value is implicit in the type of data processed by the FIFO, `dataType`.

```
  instance DPRAM fifoBuf {
    class dataType := dataType,
    class FpgaContent := FpgaContent,
      nWords := maxLag};
```

This instance declaration describes the dual-ported RAM used to implement the FIFO.

```
    function natural synthLogic(natural nWords)
      { return fifoBuf.synthLogic(nWords) + 3 * #nWords +  5 };
```

This component's logic synthesis estimate depends on the estimates for the inner RAM, on registers for reading, writing, and resetting RAM addresses, and a few units of control logic represented by the constant 5. Each address-related register is given `#nWords` bits, an expression that counts the number of bits needed to represent the `nWords` value.

```
    function natural synthRAM(natural nWords)
      { return fifoBuf.synthRAM(nWords)  };
  }
```

RAM usage depends only on the usage of the inner component, `fifoBuf`.

DPRAM.lamp

The dual-ported RAM is the only other leaf component in the computation core of the correlation array. It uses the same set of language constructs seen in the other CLAMP code samples above.

```
entity DPRAM {
    symbol entName := uniqueID(dpram_);
    import class dataType extends baseType;
    import class FpgaContent extends FPGAresource;
    import natural nWords;
    function natural synthLogic(natural nWords)
       { return #(nWords-1) };
```

Logic estimation for the dual-ported RAM need only account for one address register, with enough bits to hold the largest possible address (`nWords-1`).

```
    function natural synthRAM(natural nWords)  {
       return FpgaContent.nRams(synthSize(dataType),nWords)
    };
}
```

This function uses the FPGA's own allocation algorithm to determine the number of RAM elements that will be needed to hold `nWords` number of values, each of type `dataType`. Note that this does not directly encode any information about the RAM resources, except for the implicit assumption that there is only one kind of block RAM available. This phrasing leaves the DPRAM component independent of the FPGA used to implement it, and independent of the technique used to determine how many units of the FPGA's RAM resources are needed. When the `FpgaContent` symbol is bound to a different concretion of the `FPGAresource`, this component automatically inherits the proper FPGA-specific behavior.

REFERENCES

[3Dl04] 3Dlabs, Inc. *Wildcat Realizm User's Guide*. 2004.

[3Dl04a] 3Dlabs, Inc. *3Dlabs Wildcat Realizm Technology Questions and Answers*. 2004.

[Aba96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York. 1996

[Acc03] AccelChip. *DSP design using top-down design methodology*. Technical report, AccelChip, Inc., 2003.

[Acc04] Accellera Organization, Inc. *SystemVerilog 3.1a Language Reference Manual: Accellera's Extensions to Verilog®,* Accellera Organization, Inc. Napa CA 2004

[Arc05] AcroDesign Technologies, Inc. http://www.gmvhdl.com/acrodesign/research.html verified on 38 Nov 2005.

[Act01] Actel Corporation. *IP Solutions Improve Time to Market and Reduce Risk.* Sunnyvale, CA. 2001

[Act03] Actel Corporation. *ACTgen® Macros Reference Guide*. Sunnyvale, CA. 2003

[Ada95] ISO/IEC. *Ada Reference Manual, ISO/IEC 8652:1995(E) with COR.1:2000*. The Mitre Corporation. 2000

[Adi02] Adiga, N.R., et al. *An overview of the BlueGene/L supercomputer*. In Proceedings of Supercomputing '02. 2002.

[Ags95] K. Agsteiner, D. Monjau, and S. Schulze, S. *Object-oriented high-level modeling of system components for the generation of VHDL code*. Proceedings of the European Design Automation Conference. 1995.

[Alt90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. Journal of Molecular Biology. 215(3):403-10. 1990.

[Alt03] Altera Corporation. *Altera DSP Builder Reference Manual*. San Jose, CA. 2003

[Alt05] Altera Corporation. Stratix II Device Handbook, Volume 1. San Jose CA. 2005.

[Amd67] Gene M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. In Proceedings AFIPS 1967 Spring Joint Computer Conference, volume 30, pages 483–485, April 1967

[Ann05] Annapolis Micro Systems, Inc. *WILDSTAR II PRO for PCI*. Annapolis, MD, 2005.

[Arc94] Denis Archambaud, Pascal Faudemay, Alain Greiner (1994). *RAPID-2, An Object-Oriented Associative Memory Applicable to Genome Data Processing*. Proceedings of the 27th Annual Hawaii International Conference on System Sciences (HICSS-27). 1994.

[ARC05] ARC International. Product brochure from http://www.arc.com (27 Sep 2005)

[ATI02] ATI Technologies, Inc. *Radeon™ Software Developer's Kit (SDK) 2.0*. Markham, Ontario. 2002.

[ATI05] ATI Technologies, Inc. *ATI CrossFire™ Technology White Paper*. 2005.

[Azi04] Navid Azizi, Ian Kuon, Aaron Eiger, Ahmad Darabhia, and Paul Chow. *Reconfigurable Molecular Dynamics Simulator*. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2004

[Baj94] R. S. Bajwa, R. M. Owens, and M. J. Irwin. *The MGAP's Programming Environment and the *C++ Language*. Proceedings of the Conference on Application Specific Array Processors. 1994

[Bal01] Pierre Baldi and Søren Brunak. *Bioinformatics: the Machine Learning Approach*. The MIT Press, Cambridge MA USA. 2001.

[Bar85] Mario R. Barbacci, Steve Grout, Gary Lindstrom, Michael P. Maloney, Elliot I. Organick, and Don Rudisill. *Ada as a Hardware Description Language: An Initial Report*. Proceedings of the Conference on Computer Hardware Description Languages and their Applications. 1985.

[Bar04] G. Bardouleau and J. Kulp. *FPGAs and software components*. High Performance Embedded Computing Conference. 2004.

[Baz00] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. *Fast Template Placement for Reconfgurable Computing Systems*. IEEE Design and Test of Computers, Jan-Mar 2000 pp. 68-83.

[Bec00] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 2000

[Bel03] S. Belkacemi, K. Benkrid, and D. Crookes. *A logic based hardware development environment*. Proceedings of the Conference on Field-programmable Custom Computing Machines. 2003.

[Ben99] Gary Benson. *Tandem Repeats Finder: a Program to Analyze DNA Sequences*. Nucleic Acids Research 27(2)573-580. 1999.

[Ben02] K. Benkrid, D. Crookes, A. Benkrid, and S. Belkacemi. *A Prolog-based hardware development environment*. Proceedings of the International Conference on Field Programmable Logic. 2002.

[Bia04] William Bialek and David Botstein. *Introductory Science and Mathematics Education for 21st-Century Biologists*. Science 303: 788-790. 2004.

[Bio05] Biocelleration Ltd. *BioXL/H - Technical Information*. http://www.biocceleration.com/BioXLH-technical.html, verified 27 July 2005

[Bje99] P. Bjesse, K. Claessen, and M. Sheeran. *Lava: Hardware design in haskell*. Sigplan Notices 34(1)174–184. 1999

[Bjö02] Dag Björklund and Johan Lilius. Fr*om UML Behavioral Descriptions to Efficient Synthesizable VHDL*. Proceedings of the IEEE Norchip Conference. 2002

[Bla90] T. Blank. *The MasPar MP-1 architecture*. In Proceedings of the 35th IEEE Computing Conference, pp. 20-24. 1990

[Blu00] H.-M. Blüthgen, and T. G. Noll. *A Programmable Processor for Approximate String Matching with High Throughput Rate*. Proceedings of the Application Specific Systems, Architectures, and Processors. 2000.

[Boh01] A.P.W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, and C. Ross. *One-step Compilation of Image Processing Applications to FPGAs*. IEEE Symposium on Field-Programmable Custom Computing. 2001

[Boh04] Wim Böhm and Jeffrey Hammes. *A Transformational Approach to High Performance Embedded Computing*. Proceedings of the High Performance Embedded Computing. 2004.

[Bor94] Manjit Borah, Raminder S. Bajwa, Sridhar Hannenhalli, and Mary Jane Irwin. *A SIMD solution to the sequence comparison problem on the MGAP*. In Proceedings of the International Conference on Application Specific Array Processors, pp. 336-345. 1994

[Bro05] D. Brunina. *FPGA accelerations of BLAST*. Master's thesis, Department of Electrical and Computer Engineering, Boston University, 2005.

[Bru02] Kim B. Bruce. 2002. *Foundations of Object-Oriented Languages*. MIT Press, Cambridge MA

[Buc04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*. ACM Transactions on Graphics 23(3)777-786, Special Issue: Proceedings of the 2004 SIGGRAPH Conference

[Bur05] Dave Bursky. *Shrinking Features Broaden ASIC Options*. Electronic Design, 13 Jan 2005.

[But03] M. Butts. *Molecular electronics: All chips will be reconfigurable.* Tutorial, 13th Int. Conf. on Field Programmable Logic and Applications, September 2003.

[Cel00] Celera, Inc. *Celera Genomics Completes the First Assembly of the Human Genome*. Press Release, Rockville, MD, 2000

[Cel03] Celoxica, Ltd. *Handel-C Language Reference Manual for DK2.0*. Abington, U.K., 2003.

[Cha03] C. Chang, K. Kuusilinna, B. Richards, and R. Broderson, R. *Implementation of BEE: A real-time large-scale hardware emulation engine*. Proc. International Conference on Field Programmable Gate Arrays. 2003.

[Cha05] Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. *Initial Experiences Porting a Bioinformatics Application to a Graphics Processor*. Proceedings of the 10th Pan-Hellenic Conference in Informatics. 2005.

[Cha05a] Roger D. Chamberlain and Berkley Shands. *Streaming Data from Disk Store to Application*. Proceedings of the of 3rd Internationall Workshop on Storage Network Architecture and Parallel I/Os, pp. 17-23. September 2005.

[Che90] G.-D. Chen, and D. Gajski. *An intelligent component database for behavioral synthesis*. ACM/IEEE Design Automation Conference. 1990.

[Che02] Chen, R. and Z. Weng. *Docking Unbound Proteins using Shape Complementarity, Desolvation, and Electrostatics*. Proteins: Structure, Function and Genetics 47:281-294, 2002

[Che03] Lok-Lam Cheng, D. W. Cheung, and Siu-Ming Yiu. *Approximate string matching in DNA sequences*. Proc. Database Systems for Advanced Applications. 2003

[Che03a] Rong Chen, Li Li, and Zhiping Weng. ZDOCK: *An Initial-Stage Protein-Docking Algorithm*. Proteins: Structure, Function and Genetics 52:80-87. 2003.

[Che03b] Rong Chen and Zhiping Weng. *A Novel Shape Complementarity Scoring Function for Protein-Protein Docking*. Proteins: Structure, Function, and Genetics 51:397-408. 2003.

[Che05] Z. Cheng, et al. *A genome-wide comparison of chimpanzee and human segmental duplications*. Nature 437:88-93. 2005

[Cho91] E. Chow, T. Hunkapiller and J. Peterson. *Biological Information Signal Processor*. Proc. Application Specific Array Processors, 1991.

[Chu98] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawryzynek. *Object oriented circuit generators in Java*. Conference on Field-programmable Custom Computing Machines. 1998.

[Cla05] Peter Clarke. *Scottish Companies for FPGA Computing Alliance*. EE Times, 25 May 2005.

[Cle05] ClearSpeed Technology plc. http://www.clearspeed.com

[Cle05a] ClearSpeed Technology plc.*CSX Processor Architecture White Paper*. 2005

[Com02] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck. *Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing*. IEEE Transactions on VLSI 10(30)209-220. 2002

[Com02a] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys, 34(2) 171–210, 2002.

[Con04] A. Conti, T. VanCourt, and M. Herbordt. *Flexible FPGA acceleration of dynamic programming string processing*. In Proc. Field Programmable Logic and Applications 2004.

[Con04a] Conflex Corporation. *MDGRAPE-2: Molecular Dynamics Supercomputing for the PC*. 2004

[Cra00] Cray, Inc. AMBER and Cray, Inc.: *Providing Supercomputing to Life Science*. Seattle, WA, 2000

[Cra05] Cray, Inc. *Cray XD1 Supercomputer*. www.cray.com/products/xd1, 2005.

[Cul99] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan-Kaugmann, San Francisco, CA, 1999.

[Cur05] Derek R. Curd, Punit S.Kalra, Richard J. LeBlanc, Stephen W. Trynosky, Jeffrey V. Lindholm, and Trevor J. Bauer. *Partial Reconfiguration of a Programmable Logic Device using an On-Chip Processor*. U. S. Patent 6,907,595.

[Dah99] David Dahle, Leslie Grate, Eric Rice, and Richard Hughey. *The UCSC Kestrel General Purpose Parallel Processor*. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. 1999.

[Dal03] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, François Labonté, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, Ian Buck. *Merrimac: Supercomputing with Streams*. SC2003. November 2003, Phoenix, Arizona

[Dam04] Robertas Damaševičius and Vytautus Štuikys. 2004. *Application of UML for Hardware Design Based on Design Process Model*. Proceedings of the 2004 Conference on Asia and South Pacific Design Automation. IEEE, p.244-249

[Dim02] Matthew W. Dimmic, Joshua S. Rest, David P. Mindell, and Richard A. Goldstein. *rtREV: An Amino Acid Substitution Matrix for Inference of Retrovirus and Reverse Transcriptase Phylogeny*. Journal of Molecular Evolution 55:65-73. 2002.

[Dom98] R. Domer, and D. Gajski. *Comparison of the scenic design environment and the SpecC system*. Tech. rep., University of California at Irvine. 1998.

[Dom02] R. Domer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual Version 2.0*. SpecC Technology Open Consortium. 2002.

[Dou03] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta. *Balboa*: *A component-based design environment for system models*. IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems 22(12)1597–1612. 2003.

[Dra00] Bruce Draper, Walid Najjar, Wim Böhm, Jeff Hammes, Charlie Ross, Monica Chawathe, and José Bins. *Compiling and Optimizing Image Processing Algorithms for FPGAs*. IEEE International Workshop on Computer Architecture for Machine Perception. 2000

[Du94] M.-W. Du and S. C. Chang. *An approach to designing very fast approximate string matching algorithms*. IEEE Transactions on Knowledge and Data Engineering 6(4)620-633. 1994

[Dur98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press. 1998.

[Eck96] Wolfgang Ecker. *An Object-Oriented View of Structural VHDL Description*. Proceedings of the VHDL International User's Forum. 1996

[ECMA99] ECMA. *Standard ECMA-262, 3rd edition. ECMAScript Language Specification*. European Computer Manufacturer's Association. 1999

[ECMA05] ECMA. *Standard ECMA-334, 3rd edition. C# Language Specification*. European Computer Manufacturer's Association. 2005

[ECMA05a] ECMA. *Standard ECMA-367. Eiffel Analysis, Design, and Programming Language*. European Computer Manufacturer's Association. 2005

[Edw04] S. Edwards. *The challenges of hardware synthesis from C-like languages*. Int. Workshop on Logic Synthesis. 2004.

[Eld96] James J. Eldredge and Brad L. Hutchings. *Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs*. Journal of VLSI Signal Processing, 12:67-86. 1996.

[Ell73] Robert Ellis. *Experiences with an Evolving System*. Computer 6(10). October 1973.

[Eng04] Wolfgang Engel, ed. *Shader X2: Introductions and tutorials with DirectX® 9*. Wordware Publishing, Inc. Plano TX. 2004.

[Ere04] Mattan Erez, Jung Ho Ahn, Ankit Garg, William J. Dally, and Eric Darve. *Analysis and Performance Results of a Molecular Modeling Application on Merrimac*. Proceedings of the Supercomputing '04 Conference, November 2004, Pittsburgh, Pennsylvania

[Est63] G. Estrin, B. Bussell, R. Turn, and J. Bibb. *Parallel Processing in a Restructurable Computer System*. IEEE Transactions of Electronic Computers. 12:747-755. 1963.

[Eva03] Evans and Sutherland Computer Corporation. *simFUSION® 6000: The Open, PC-Based Simulation Solution*. 2003.

[Eva03a] Evans and Sutherland Computer Corporation. *RenderBeast™ Scalable Visualization Solutions*. 2003.

[Fai93] Rickard E. Faith and Doug L. Hoffman. Technical Report *TR93-051: A Computer Architecture for Fast Approximate Pattern Matching*. http://www.cs.unc.edu/~techlib/FILE.html on 25 Jul 2005

[Fau95] Pascal Faudemay and Laurent Winckel. *A High Level Language for the RAPID-2 Massively Parallel Accelerator Board*. International Conference on Algorithms and Architectures for Parallel Processing. 1995.

[Fel04] Joseph Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Sunderland MA. 2004.

[Fon04] John W. Fondon III and Harold R. Garner. *Molecular origins of rapid and continuous morphological evolution*. Proceedings of the National Academy Science 101:19058-18063. 2004

[Fos80] M. J. Foster and H. T. Kung. *The Design of Special-Purpose VLSI Chips*. Computer, 13(1)24-40. 1980.

[Fri95] Lisa Friendly. *The Design of Distributed Hyperlinked Programming Documentation*. International Workshop on Hypermedia Design 1995

[Fuj05] Fujitsu Computer Corp. B*ioServer: Cost-effective, energy-efficient, grid computing*. 2005.

[Gal95] D. Galloway. *The Transmogrifier C hardware description language and compiler for FPGAs*. Conference on Field-programmable Custom Computing Machines. 1995.

[Gam94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design Patterns:Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1995.

[Gar05] A. Gara et al. *Overview of the Blue Gene/L System Architecture*. IBM Journal of Research and Development 49(2/3)195-252. 2005.

[Ger03] Vassilios Gerousis (general chair). SystemVerilog Chairs and Champions Document Version 3. http://www.eda.org/sv-ec/SV_3.1_Web/SVChairsChampionsResponse.pdf (verified on 8 Sep 2005)

[GGF02] The Global Grid Forum (2002). *Advanced Reservation API and A Grid Monitoring Architecture*. http://www.ggf.org/Documents/GFD/GFD-E.5.pdf and http://www.ggf.org/Documents/GFD/GFD-I.7.pdf, respectively (verified on 15 Jan 2003)

[Gid05] GiDEL Ltd. http://www.gidel.com/products.htm (verified on 27 July 2005)

[Gil01] W. J. Gilmore. A Programmer's Introduction to PHP 4.0. Apress, Berkeley CA. 2001

[Gim03] Eike Gimpe and Frank Oppenheimer. *Extending the SystemC Synthesis Subset by Object-Oriented Features*. Proc International Conference on Hardware-Software Codesign and System Synthesis. 2003

[Gir93] Emil Girczyc and Steve Carlson. *Increasing design quality and engineering productivity through design reuse*. Proceedings of the ACM IEEE Design Automation Conference (DAC). 1993

[Giv00] Givargis, T., and Vahid, F. *Parameterized system design*. In International Workshop on Hardware/Software Codesign. 2000.

[Glo02] The Globus Project (2002). *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. http://www.globus.org/research/papers/ogsa.pdf on 15 Jan 2003

[Gok91] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweely, and Daniel Lopresti. *Building and Using a Highly Parallel Programmable Logic Array*. IEEE Computer, Jan 1991, pp. 82-89

[Gok97] M. Gokhale, and E. Gomersall, E. *High level compilation for fine grained FPGAs*. Conference on Field-programmable Custom Computing Machine. 1997.

[Gok00] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, M. *Stream-oriented FPGA computing in the Streams-C high-level language*. Conference on Field-programmable Custom Computing Machines. 2000.

[Gol99] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. *PipeRench: A coprocessor for streaming multimedia acceleration*. In International Symposium on Computer Architecture (1999).

[Gol00] Brian B. Goldman and W. Todd Wipke. *QSD quadratic shape descriptors. 2. Molecular docking using quadratic shape descriptors (QSDock).* Proteins. 38(1):79-94. 2000

[Gos05] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Third Edition*. Addison-Wesley, Boston. 2005

[Gov04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin and Dinesh Manocha. *Fast Database Operations using Graphics Processors*. Proc. of ACM SIGMOD, 2004

[Gov05] Naga K. Govindaraju, Nikunj Raghuvanshi and Dinesh Manocha. *Fast and Approximate Stream Mining of Qualities and Frequencies Using Graphics Processors*. Proc. ACM SIGMOD, 2005

[Gra89] J. P. Gray and T. A. Kean. *Configurable Hardware: A New Paradigm for Computation*. Proceedings of the Decennial CalTech Conference on VLSI. 1989.

[Gra01] L. Grate, M. Diekhans, D. Dahle, and R. Hughey. *Sequence analysis with the Kestrel SIMD parallel processor*. In Pacific Symposium on Biocomputing, pp. 323-334. 2001

[Gra03] Kris Gray. *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press. 2003

[Gre85] Steve Gregory, Bob Neely, and Graem Ringwood. *Parlog for Specification, Verification, and Simulation*. Proceedings of the Conference on Computer Hardware Description Languages and their Applications. 1985.

[Gu05] Y. Gu, T. VanCourt, and M. C. Herbordt. *Accelerating molecular dynamics simulations with configurable circuits*. Proc. of Field Programmable Logic and Applications 2005.

[Gu06] Y. Gu, T. VanCourt, and M. C. Herbordt. *Accelerating Molecular Dynamics Simulations with Configurable Circuits*. IEE Proceedings on Computers & Digital Techniques, accepted for publication in 2006.

[Guc99] Steven A. Guccione and Delon Levi. *Run-Time Parameterizable Cores*. In Patrick Lysaght, James Irvine and Reiner W. Hartenstein, editors, Field-Programmable Logic and Applications, pages 215-222. Springer-Verlag, Berlin, August / September 1999. Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications, FPL 1999. Lecture Notes in Computer Science 1673.

[Guc99a] Steve Guccione, Delon Levi, and Prasanna Sundarajan. *JBits: Java based interface for reconfigurable computing*. Proc. Military and Aerospace Applications of Programmable Devices and Technologies. 1999.

[Guc02] Steven A. Guccione and Eric Keller. *Gene Matching Using JBits*. Proc. Field Programmable Logic 2002

[Guo95] S. Guo, and W. Luk. *Compiling Ruby into FPGAs*. International Conference on Field Programmable Logic. 1995.

[Gup89] R. Gupta, et al., *An Object Oriented VLSI CAD Framework: A case study in Rapid Prototyping.* Computer, 1989: p. 28-37.

[Gup97] R.K. Gupta and S.Y Liao. *Using a Programming Language for Digital Design Systems*. IEEE Design and Test of Computers, 14(2)72-80 1997.

[Gus97] Dan Gusfield,. *Algorithms in Strings, Trees, and Sequences*. Cambridge University Press. Cambridge UK. 1997.

[Had95] J. D. Hadley and B. L. Hutchings. *Design methodologies for partially reconfigured systems*. Proc. Field-programmable Custom Computing Machines (FCCM) 1995.

[Hal02] Inbal Halperin, Buyong Ma, Haim Wolfson, and Ruth Nussinov. *Principles of Docking: An Overview of Search Algorithms and a Guide to Scoring Functions*. Proteins: Structure, Function and Genetics 47:409-443, 2002

[Ham99] Jeff Hammes, Bob Rinker, Wim Böhm, and Walid Najjar. C*ompiling a High-level Language to Reconfigurable Systems*. Compiler and Architecture Support for Embedded Systems (CASES). 1999.

[Ham01] J. Hammes, A.P.W. Böhm, M. Chawathe., B. Draper, R. Rinker, and W. Najjar. *Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops*. IPDPS Reconfigurable Architecture Workshop. 2001.

[Har94] A.Hartman. *A fundamental construction of 3-designs*. Discrete Mathematics 124: 107-132. 1994.

[Har01] R. Hartenstein. *A decade of reconfigurable computing: a visionary retrospective*. Proc. International conference on Design, Automation and Test in Europe. 2001.

[Har04] Mark Harris, David Leubke, et al. SIGGRAPH 2004 GPGPU Course. http://www.gpgpu.org/s2004/ on 28 July 2005.

[Har05] Mark Harris, David Leubke, et al. SIGGRAPH 2005 GPGPU Course. http://www.siggraph.org/s2005 on 29 July 2005.

[Hau97] J. Hauser, and J. Wawryzynek. *Garp: A MIPS processor with a reconfigurable coprocessor*. Conference on Field-programmable Custom Computing Machines pp. 24–33. 1997.

[Hen03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, third edition*. Morgan Kaufmann, San Francisco. 2003

[Hir96] Jeffrey D. Hirschberg, Richard Hughey, and Kevin Karplus. *Kestrel: A Programmable Array for Sequence Analysis*. Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors. 1996

[Hoa93] Dzung T. Hoang. *Searching Genetic Databases on Splash 2*. IEEE Workshop on FPGAs for Custom Computing Machines 1993

[Hof99] Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. *Fast computation of generalized Voronoi diagrams using graphics hardware*. SIGGRAPH 99.

[Hor01] Edson L. Horta and John Lockwood. PARBIT: *A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs).* Department of Computer Science, Applied Research Lab, Washington University technical report WUCS-01-13.

[Hor04] Edson L. Horta and John W. Lockwood. *Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs*. Proc. Field Programmable Logic.2004.

[Hug95] Richard Hughey and George M. Church. *Parallel Sequence Comparison and Alignment*. Proceedings of the International Conference on Application-specific Array Processors.1995

[Hug00] Jason D. Hughes, Preston W. Estep, Saeed Tavazoie, and George M. Church. *Computational Identification of Cis-regulatory Elements Associated with Groups of FunctionallyRelated Genes in Saccharomyces cerevisiae*. Journal of Molecular Biology 296:1205-1214. 2000.

[Hut95] Brad L. Hutchings and Michael J. Wirthlin. *Implementation Approaches for Reconfigurable Logic Applications*. 5th International Workshop on Field Programmable Logic and Applications, pp 419-428. 1995

[Hut99] B. Hutchings, P. Bellwos, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, M. *A CAD suite for high-performance FPGA design*. In Conference on Field-programmable Custom Computing Machines. 1999.

[Hut01] Brad .L. Hutchings and Brent E. Nelson. *Unifying Simulation and Execution in a Design Environment for FPGA Systems*. IEEE Transactions on CAD 9(1)201-204. 2001

[Hwa01] J. Hwang, B. Milne and N. Shirazi, and J. Stroomer. System Level Tools for DSP in FPGAs. Proceedings of the International Conference on Field Programmable Logic. 2001.

[IBM02] IBM (2002). *High Availability Cluster Multiprocessing for AIX on IBM e-server pSeries*. http://www1.ibm.com/servers/aix/products/ibmsw/high_avail_network/hacmp .pdf on 15 Jan 2003

[IBM02a] IBM (2002). *High Performance Cluster File System*. http://www1.ibm.com/servers/eserver/clusters/software/gpfs.pdf on 15 Jan 2003

[IBM03] IBM (2003). *IBM Delivers Supercomputing On Demand*. http://www.ibm.com/news/us/2003/01/091.html on 15 Jan 2003

[IBM04] IBM. *IBM Rational Suite Family*. http://www3.software.ibm.com/ibmdl/pub/software/rational/web/datasheets/version6/suite.pdf on 17 March 2004

[IEEE97] IEEE (1997). IEEE STD 1076.3-1997: *IEEE Standard VHDL Synthesis Packages*.

[IEEE01] IEEE (2001). IEEE STD 1364-2001: *IEEE Standard Verilog® Hardware Description Language*.

[IEEE02] IEEE (2002). IEEE STD 1076™-2002: *IEEE Standard VHDL Hardware Description Language*.

[IEEE02a] IEEE (2002). IEEE STD 1076.6-1999: *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*.

[Imp05] Impulse Acceleration Technology, Inc. 2005.

[Jac98] P. Jackson et al. *PCR analysis of tissue samples from the 1979 Sverdlovsk anthrax victims*.Proceedings of the National Academy of Science 95:1224-1229. 1998

[Jan00] Axel Jantsch and Ingo Sander. 2000. *On the Roles of Functions and Objects in System Specification*. Proc.Conference on Hardware/Software Codesign 2000. ACM/IEEE

[Jen91] Jensen, Kathleen and Niklasu Wirth. *Pascal User Manual and Report: Revised for the ISO Pascal Standard*. Springer-Verlag, Berlin. 1991

[Joh02] Richard A. Johnson and Dean W. Wichern. Applied Multivariate Statistical Analysis, Fifth Edition. Prentice Hall, Upper Saddle River NJ. 2002.

[Jon90] G. Jones, and M. Sheeran. *Circuit design in Ruby*. Formal Methods for VLSI Design, IFIP 10.5 Lecture Notes, J. Staunstruph, Ed. North-Holland, 1990.

[Jou89] Norman P. Jouppi and David W. Wall. *Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines*. Proc. Architectural Support for Programming Languages and Operating Systems `89. 1989.

[Kam01] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura. *A C-based synthesis system, Bach, and its applications.* Asia-South Pacific Design Automation Conference. 2001.

[Kat93] Ephraim Katzir-Katchalski, Isaac Shariv, Miriam Eisenstein, Asher A. Friesem, Claude Alfalo, and Ilya A. Vakser. *Molecular Surface Recognition: Determination of Geometric Fit between Proteins and their Ligands by Correlation Techniques*. Proc National Academy of Science 89:2195-2199. 1992.

[Kaw00] S. Kawashima and M. Kanehisa, M. *AAindex: amino acid index database*. Nucleic Acids Research. 28, 374 (2000).

[Kea88] Thomas Andrew Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*. Ph. D. dissertation, University of Edinburgh. 1988

[Ker78] Brian w. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall 1978.

[Kim01] Y. Kim, M.-J. Noh, T.-D.Han, S.-D. Kim, and S.-B. Yang. *Finding genes for cancer classification: Many genes and small number of samples*. In 2nd Annual Houston Forum on Cancer Genomics and Informatics. 2001.

[Kle03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained – The Model-Driven Arcitecture: Practice and Promise*. Addison-Wesley, Boston. 2003.

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Leland Stanford Junior University. 1992

[Koh01] Teuvo Kohonen. *Self-Organizing Maps (third edition)*. Springer-Verlag. Berlin. 2001

[Kos01] Timo Koski. *Hidden Markov Models for Bioinformatics*. Kluwer Academic Publishers, Dordrecht. 2001.

[Koz05] Dima Kozakov, Karl H. Klodfelter, Sandor Vajda, and Carlos J. Camacho. *Optimal Clustering for Detecting Near-Native Conformations in Protein Docking*. Biophysical Journal 89:867-875. August 2005.

[Ku90] D. Ku, D. and Micheli. *HardwareC - a language for hardware design*. Tech. Rep. CSL-TR-90-419, Stanford University, 1990.

[Kuh00] Tommy Kuhn and Wolfgang Rosensteil. 2000. *Java Based Object Oriented Hardware Specifications and Synthesis*. Asia and South Pacific Design Automation Conference, 2000. IEEE/ACM

[Lav96] Dominique Lavenier (1996). *SAMBA Systolic Accelerators for Molecular Biological Applications*. http://citeseer.nj.nec.com/lavenier96samba.html on 7 Jan 2003

[Lav98] Dominique Lavenier (1998). *Speeding Up Genome Computation with a Systolic Accelerator*. http://citeseer.nj.nec.com/489695.html on 7 Jan 2003

[Lee97] Hsi-Chieh Lee and F. Ercal. *RMESH algorithms for parallel string matching*. International Symposium on Parallel Architectures, Algorithms, and Networks. 1997

[Lef04] Aaron Lefohn, Ian Buck, John Owens, and Robert Strzodka. *GPGPU: General Purpose Computation on Graphics Processors: IEEE Visualization 2004 Tutorial*. http://www.gpgpu.org/vis2004/gpgpuVis04_courseNotes_fourPerPage.pdf on 29 July 2004

[Lem95] Eric Lemoine and David Merceron. *Run Time Reconfiguration of FPGA for Scannig Genomic Database*. Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM). 1995.

[Len00] C. Lennard, P. Schaumont, and G. de Jong. *Standards for system-level design: Practical reality or solution in search of a question?* Design Automation and Test in Europe. 2000.

[Lia03] J. Liang, R. Tessier, and O. Mencer. *Floating point unit generation and evaluation for FPGAs*. Conference on Field-programmable Custom Computing Machines. 2003.

[Lip85] Richard J. Lipton and Daniel Lopresti. *A Systolic Array for Rapid String Comparison*. Chapel Hill Conference on VLSI, 1985.

[Lip86] Richard Lipton and Daniel Lopresti. *Comparing Long Strings on a Short Systolic Array* in Systolic Arrays (Will Moore, Andrew McCabe, Roddy Uquhart, eds.). Adam Hilger 1986.

[Lis84] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Schiffler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, New York. 1984

[Lis88] Barbara Liskov. *Data abstraction and hierarchy*. SIGPLAN notices 35(5). 1988.

[Liu00] Chamond Liu. *Smalltalk, Objects, and Design*. toExcel, New York. 2000

[Lop87] Daniel P. Lopresti. *P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences*. Computer 20(7)98-99. IEEE. 1987.

[Luk89] Wayne Luk and Geraint Jones. *Computer-Based Tools for Regular Array Design*, in Systolic Array Processors (J. McCanny, J. McWhirther, and E. Swartzlander Jr. Eds) Prentice Hall International pp.589-598. 1989

[Luk98] W. Luk, A. and McKeever. Pebble: *A language for parameterised and reconfigurable hardware design*. International Conference on Field Programmable Logic. 1998.

[Mac04] John S. MacNeil. *Hot Rod Homology Heaven*. Genome Technology 41:22-23, 2004.

[Mad93] Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. ACM 1993

[Man85] Tamio Mano, Fumihiro Maruyama, Kazushi Hayashi, Taeko Kakuda, Nobukai Kawato, and Takao Uehara. *Occam to CMOS: Experimental Logic Design Support System*. Proceedings of the Conference on Computer Hardware Description Languages and their Applications. 1985.

[Man05] Dinesh Manocha. *General Purpose Computations using Graphics Processors*. IEEE Computer 38(8)85-88. August 2005.

[Mar00] T. Maruyama and T. Hoshino. *A C to HDL compiler for pipeline processing on FPGAs*. In Conference on Field-programmable Custom Computing Machines. 2000.

[Mar03] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Pearson Education Inc. Upper Saddle River NJ. 2003.

[Mat03] The Mathworks. *Simulink Reference, Version 5*. Natick, MA. 2003

[McM00] Scott McMillan and Steven A. Guccione. *Partial Run-Time Reconfiguration Using JRTR*. Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications, FPL 2000. Lecture Notes in Computer Science 1896.

[Meg90] G. M. Megson. *Efficient systolic string matching*. Electronics Letters, 26(24)2040-2042. 1990

[Meg01] S. Meguerdichian, F. Koushanfar, A. Mogre, D. Petranovic, and M. Potkonjak, M. *MetaCores: design and optimization technique*. ACM/IEEE Design Automation Conference. 2001.

[Men99] O. Mencer, and M. Platz. *Dynamic circuit generation for boolean satisfiability in an object-oriented design environment*. In Hawaii International Conference on System Sciences (1999).

[Men00] O. Mencer, L. Semeria, M. Morf, and J.-M. Delosme. *Application of reconfigurable CORDIC architectures*. The Journal of VLSI Signal Processing, special issue on Reconfigurable Computing. 2000.

[Men01] Oskar Mencer, Marco Platzner, Martin Morf, and Michael J. Flynn. *Object-oriented Domain Specific Compilers for Programming FPGAs*. IEEE Transactions on VLSI, 2001

[Men02] Oskar Mencer. *PAM-Blox II: Design and Evaluation of C++ Module Generation for Computing with FPGAs*. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), Napa, California, April 2002

[Men03] Oskar Mencer, David J. Pearce, Lee W. Howes, and Wayne Luk. *Design Space Exploration with a Stream Compiler*. IEEE International Conference on Field Programmable Technology (FPT), Tokyo, Dec. 2003

[Men06] Oskar Mencer. ASC: *A Stream Compiler for Computing with FPGAs*. to appear in IEEE Trans. Computer Automated Design 2006.

[Mes01] Francisco Mesa-Martinez, Eric Perlman, and Richard Hughey (2001). *The UCSC Kestrel High Performance SIMD Processor: Present And Future*. http://citeseer.nj.nec.com/503554.html on 8 Jan 2003

[Mey86] Bertrand Meyer. *Genericity versus Inheritance*. Proc. ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages and Applications 1986.

[Mey92] Meyer, Bertrand. *Eiffel: The Language*. Prentice Hall, New York. 1992

[Mit05] Mitrionics AB. *A true software approach to FPGA programming: Deploying the Mitrion Virtual Processor*. http://www.mitrion.com/press/Mitrion_whitepaper_050729.pdf, verified 29 Sep 2005

[Moi01] P. Moisset, P. Diniz, and J. Park. *Matching and searching analysis for parallel hardware implementation in FPGAs*. ACM/SIGDA International Symposium on Field Programmable Gate Arrays. 2001.

[Mor03] Kenneth Moreland and Edward Angel. *The FFT on a GPU*. Eurographics Conference on Graphics Hardware 2003.

[Mor04] Ádám Moravánszky. *Dense Matrix Algebra on the GPU*, in *Shader X^2, Shader Programming Tips and Tricks with DirectX 9*, Wolfgang F. Engel ed. Wordware Publishing, Inc. 1984.

[Muk79] Amar Mukhopadhyay. *Hardware Algorithms for Nonnumeric Computation*. IEEE. Trans. On Computers. C-28(6)384-394

[Muk89] Amar Mukherjee. *Hardware Algorithms for Determining Similarity Between Two Strings*. IEEE Trans. On Computers, 38(4)600-603. 1989.

[Nal05] Nallatech Ltd. *Product Line Card*. www.nallatech.com, 2005.

[Nal05a] Nallatech Ltd. *Alliance Launched in Scotland to Develop the Next Generation of Supercomputers*. http://www.nallatech.com. 2005.

[Neb96] W. Nebel, and G. Schumacher. *Object-oriented hardware modeling - where to apply and what are the objects?* Proceedings of the European Design Automation Conference. 1996.

[Nee70] S. B. Needleman C. D. Wunsch. *A General Method Applicable to the Search for Similarities in the Amino Acids Sequences of Two Proteins*, Journal of Molecular Biology 48:443-453. 1970.

[Nei00] Masatoshi Nei and Sudhir Kumar. *Molecular Evolution and Phylogenetics*. Oxford University Press, Oxford UK. 2000

[Ng00] Pauline C. Ng, Jorja G. Henikoff, and Steven Henikoff. *PHAT: a Transmembrane-Specific Substitution Matrix*. Bioinformatics 16(9) 760-766. 2000.

[NVI04] NVIDIA Corporation. *Cg Toolkit, User's Manual*. NVIDIA Corporation, Santa Clara CA. 2004

[NVI05] NVIDIA Corporation. *NVIDIA GPU Programming Guide, Version 2.4.0*. NVIDIA Corporation, Santa Clara CA. 2005

[NVI05a] NVIDIA Corporation (2005). http://www.slizone.com/page/slizone_learn.html on 10 Aug 2005

[OCP01] OCP International Partnership. *Open Core Protocol Specification Release 1.0*. OCP International Partnership, 2001.

[Ode03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. *A Nominal Theory of Objects with Dependent Types*. Proc. Foundations of Object Oriented Languages (FOOL) 10, January 2003

[Odo02] John O'Donnell. *Overview of Hydra: A Concurrent Language for Synchronous Digital Circuit Design*. Proceedings of the International Parallel and Distributed Processing Symposium 2002.

[Oli02] Oliveira, M., and Hu, A. *High-level specification and automatic generation of IP interface monitors*. ACM/IEEE Design Automation Conference.2002.

[OMG01] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification, Rev 2.5*. 2001.

[OMG03] Object Management Group, Inc. *OMG Unified Modeling Language Specification v1.5*. http://www.omg.org . 2003

[OMG03a] Object Management Group, Inc. *UML™ Profile for Schedulability, Performance, and Time Specification, version 1.0. 2003*. http://www.omg.org/technology/documents/formal/uml.htm on 17 March 2004

[Ori05] Orion Multisystems, Inc. Product information. http://www.orionmulti.com on 28 July 2005

[OSC03] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual Revision 1.0*. Open SystemC Initiative, San Jose, CA, 2003.

[Owe05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A *Survey of General-Purpose Computation on Graphics Hardware*. Eurographics 2005.

[Pag95] Ian Page and Wayne Luk. *Compiling occam into Field-Programmable Gate Arrays*. Proceedings of the International Conference on Field Programmable Logic and Applications (FPL) 1995.

[Pag03] Roderic D. M. Page. *Tangled Trees: Phylogeny, Cospeciation, and Coevolution*. The University of Chicago Press, Chicago. 2003.

[Pag04] Ian Page. *Compiling software to gates*. http://www.embedded.com 20 Dec 2004

[Par01] Paracel Inc. (2001), http://www.paracel.com/publications/hmm_white_paper.html on 8 Jan 2003

[Pee00] Roger M.A. Peel and Barry M. Cook. *Occam on Field Programmable Gate Arrays - Fast Prototyping of Parallel Embedded Systems*. Proc, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). 2000

[Pel05] David Pellerin and Scott Thibault. *Practical FPGA programming in C*. Prentice Hall, Upper Saddle River NJ. 2005.

[Per99] S. Periyacheri, A. Nayak, A. Jones, N. Shenoy, A. Choudhary, and P. Banerjee. *Library functions in reconfigurable hardware for matrix and signal processing operations in MATLAB*. IASTED Parallel and Distributed Computing and Systems. 1999.

[Per00] Charles M. Perou, Therese Sørlie, Michael B. Eisen, Matt van de Rijn, Stefanie S. Jeffrey, Christian A. Rees, Jonathan R. Pollack, Douglas T. Ross, Hilde Johnsen, Lars A. Akslen, Øystein Fluge, Alexander Pergamenschikov, Cheryl Williams, Shirley X. Zhu, Per E. Lønning, Anne-Lise Børresen-Dale, Patrick O. Brown, David Botstein. *Molecular portraits of human breast tumours*. Nature 406:747-752. 17 August 2000.

[Phi99] Philips Electronics North America Corporation. TriMedia TM1100 Data Book. 1999

[Pie02] Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002

[Pla92] P. J. Plauger. *The Standard C Library*. Prentice Hall, Englewood Cliffs NJ. 1992

[Pla02] Platform Computing, Inc. (2002). *A Guide to Harnessing Grid Computing in the Enterprise*. http://www.platform.com/pdfs/whitepapers/EnterpriseGrid_11_28_02.pdf on 5 Jan 2003

[Pre92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press (1992)

[Ran96] Nagarajan Ranganathan and Raghu Sastry. *VLSI Circuit for Determining the Edit Distance Between Strings*. U. S. Patent 5,553,272

[Rap95] D. C. Rapaport. *The Art of Molecular Dynamics Simulations*. Cambridge University Press, Cambridge UK. 1995

[Ran97] N. Ranganathan and R. Motamarri. *A VLSI architecture for computing the optimal correspondence of string subsequences*. Proceedings of the Computer Architecture for Machine Perception. 1997.

[Reu99] A. Reutter and W. Rosensteil. *An Efficient Reuse System for Digital Circuit Design*. Proceedings of the Design Automation and Test in Europe. 1999.

[Rin01] Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, Wim Böhm. *An automated process for compiling dataflow graphs into reconfigurable hardware*. IEEE Trans. on VLSI Systems 9(1)130-139. 2001. 1999.

[Ris72] Edward W. Riseman and Caxton Foster. *The Inhibition of Potential Parallelism by Conditional Jumps*. IEEE Transactions on Computers 21(12):1405-1411

[Ros04] Randi J. Rost. *OpenGL® Shading Language*. Addison-Wesley Professional. 2004

[Row97] J. Rowson, and A. Sangiovanni-Vincentelli. *Interface-based design*. ACM/IEEE Design Automation Conference. 1997

[Rus95] John C. Russ. *The Image Processing Handbook*. CRC Press. 1995.

[Sal98] S.L. Salzberg, D.B. Searls, S. Kasif (eds). *Computational Methods in Molecular Biology*. Elsevier, 1998

[Sas95] Raghu Sastry, N. Ranganathan, and Klinton Remedios. *CASM: A VLSI Chip for Approximate String Matching*. IEEE Trans. On Pattern Matching and Machine Intelligence 18(8)824-830. 1995.

[Sch95] Guido Schumacher and Wolfgang Nebel. 1995. *Inheritance Concepts for Signals in Object-Oriented Extensions to VHDL*. EuroDAC 95 IEEE.

[Sch99] Patrick Schaumont, Radim Cmar, Serge Vernalde, Marc Engles, and Ivo Bolsens. *Hardware Reuse at the Behavioral Level*. Proc. of DAC 99. 1999.

[Sch05] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis. Vienna University of Technology. 2005

[Sel94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994

[Sel98] Bran Selic and Jim Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*, 1998. Rational Inc.

[Sém01] Luc Séméria and Giovanni De Micheli. *Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C*. IEEE Trans. On Computer Aided Design 20(2)213-233. 2001.

[Sil01] Silicon Graphics, Inc. *SGI Bioinformatics Performance Report*. Mountain View, CA, 2001

[Sil04] Silicon Graphics, Inc. *Extraordinary Accleration of Workflows with Reconfigurable Application-Specific Computing from SGI*. www.sgi.com/pdfs/3721.pdf, 2004.

[Sin93] Raj K. Singh, Stephen G. Tell, C. Thomas White, Doug Hoffman, and Vernon L. Chi. *A Scalable Systolic Multiprocessor System for Analysis of Biological Sequences*. Proceedings of the Symposium on Integrated Systems. 1993

[Sin95] S. Singh. *Architectural description for FPGA circuits*. In Conference on Field-programmable Custom Computing Machines. 1995.

[Sin02] S. Singh. *Interface specification for reconfigurable components*. International Conference on Computer-Aided Design. 2002.

[Sin00] V. Sinha, F. Doucet, C. Siska, R., Gupta, S. Liao, and A. Ghosh. *YAML: A tool for hardware design visualization and capture*. In International Symposium on System Synthesis (2000).

[Smi90] D. Smith and E. Pierzchala. *An algorithm and architecture for approximate string matching*. Proceedings of the 33rd Midwest Symposium on Circuits and Systems. 1990.

[Sni01] G. Snider, B. Shackelford, and R. Carter. *Attacking the semantic gap between application programming languages and configurable hardware*. ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 115–124. 2001.

[Sny86] L. Snyder. *Type architectures, shared memory, and the corollary of modest potential*. Annual Review of Computer Science, 289–317. 1986

[Sod98] D. Soderman, and Y. Panchul. *Implementing C algorithms in reconfigurable hardware using C2Verilog*. In Conference on Field-programmable Custom Computing Machines. 1998.

[SRC05] SRC Computers, Inc. http://www.srccomp.com . Verified on 1 Dec 2005.

[Sta04] Starbridge Systems, Inc. *Starbridge Hypercomputer System Hardware Manual*. 2004

[Sta05] Starbridge Systems, Inc. *Viva 2.4.1 User Guide*. 2005

[Ste01] R. Stevens, C. Goble, P. Baker, and A. Brass. *A classification of tasks in bioinformatics*. Bioinformatics 17 (2001)

[Str97] Bjarne Stroustrop. *The C++ Programming Language, 3rd edition*. Addison-Wesley, Boston. 1997

[Str05] E. Strohmaier, J. Dongarra, H. Meuer, and H. Simon. R*ecent Trends in the Marketplace of High Performance Computing*. CTWatch Quarterly, 1(2), February 2005. http://www.ctwatch.org/quarterly/articles/2005/02/recent-trends/ on 5 Aug 2005

[Sud04] Avneesh Sud, Miguel A. Otaduy and Dinesh Manocha. DiFi: *Fast 3D Distance Field Computation Using Graphics Hardware*. Computer Graphics Forum 23 (3): 557-566 (Proc. of Eurographics) 2004

[Sun02] Sun Microsystems, Inc. *Briefing in Computational Biology*. Palo Alto, CA, 2002

[Sun02a] Sun Microsystems, Inc. *Sun ONE Grid Engine Administration and User's Guide*. http://www.sun.com/products n solutions/hardware/docs/pdf/816 2077 12.pdf on 15 Jan 2003

[Sun02b] Sun Microsystems, Inc. *Sun ONE Grid Engine and Sun ONE Grid Engine, Enterprise Edition*. http://www.sun.com/products n solutions/hardware/docs/pdf/816 4767-11.pdf on 15 Jan 2003

[Sun05] Sun Microsystems, Inc. *Sun Grid Compute Utility*. http://www.sun.com/service/sungrid/SUN_Grid_Datasheet_20054.pdf on 5 Aug 2005.

[Sun05a] Sun Microsystems, Inc. *Java™ 2 Platform, Enterprise Edition 5.0 (J2EE™ 5.0) Specification*. 2005

[Swa87] E. Swartzlander. *Systolic Signal Processing Systems*. Marcel Drekker, Inc. 1987

[Swe02] Peter Sweeney. *Error Control Coding, from Theory to Practice*. John Wiley and Sons, Ltd., Chichester. 2002.

[Syn05] Synective Labs AB. *Pure Computational Power: The Computing Node Platform — CNP*. Goteborg, Sweden, URL: www.synective.se, 2005.

[Tai02] Makoto Taiji, Tetsu Narumi, Yousuke Ohno, Noriyuki Futatsugi, Atsushi Suenaga, Naoki Takada, and Akihiko Konagaya. *Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations*. Genome Informatics 13: 461-462. 2002.

[Ten04] Tensilica Inc. Xtensa LX Microprocessor Overview Handbook. 2004.

[Tho04] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*, Second Edition. Pragmatic Bookshelf. 2004

[Tim02] TimeLogic Corporation (2002). http://www.timelogic.com/decypher_intro.html on 8 Jan 2003

[Tim02a] TimeLogic Corporation (2002). *The Value of Accelerated Computing in Bioinformatics*. http://www.timelogic.com/whitepapers/decypher_benefits_e.pdf on 8 Jan 2003

[Tim02b] TimeLogic Corporation (2002). *ASIC vs. FPGA Engineering Tradeoffs*. http://www.timelogic.com/technology_fpga.html on 8 Jan 2003

[Tim05] Time Logic Corp. Web Site. www.timelogic.com, 2005.

[Tom05] K. Tomko. *Feasibility of FPGA co-processor acceleration of FDTD codes*. Tech. Rep. GSA Contract No. DAAD05-01-C-0033, High Performance Computing Modernization Program, 2004.

[Top03] http://www.top500.org/sublist/stats/index.php on 15 Jan 2003

[Tra05] Pedro Trancoso and Maria Charalambous. *Exploring Graphics Processor Performance for General Purpose Applications*. Euromicro Conference on Digital System Design. 2005.

[Tre04] N. Tredennick, *Reconfigurable systems emerge*. Keynote Talk, Int. Conf. on Field Programmable Logic and Applications, August 2004.

[Ull00] Michael Ullner. *Dynamic Configurable System of Parallel Modules Comprising Chain of Chips Comprising Parallel Pipeline Chain of Processors with Master Controller Feeding Command and Data*. U. S. Patent 6,112,288. 2000.

[Ung87] David Ungar and Randall B. Smith. *Self: The Power of Simplicity*. Proc. ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages and Applications 1987, ACM.

[Vah98] F. Vahid, and T. Givargis. *Incorporating cores into system-level specification*. International Symposium on System Synthesis. 1998.

[Vak94] Ilya A. Vakser and Claude Alfalo. *Hydrophobic Docking: A Proposed Enhancement to Molecular Recognition Techniques*. Proteins: Structure, Function, and Genetics 20:320-329. 1994.

[Van03] T. VanCourt, M. Herbordt, and R. Barton. *Case study of a functional genomics application for an FPGA-based coprocessor*. In Proceedings of the International Conference on Field Programmable Logic and Applications (2003), pp. 365–374.

[Van04] T. VanCourt, Herbordt, M., and R. Barton *Microarray data analysis using an FPGA-based coprocessor*. Microprocessors and Microsystems 28, 4 (2004), 213–222

[Van04a] T. VanCourt, M. Herbordt. *Families of FPGA-based algorithms for approximate string matching*. In Proceedings of the Application-Specific Systems, Architectures, and Processors 2004.

[Van04b] T. VanCourt, Y. Gu, and M. Herbordt. *FPGA Acceleration of Rigid Molecule Interactions*. Proceedings of the Field Programmable Logic and Applications, p.862-867. 2004.

[Van05] T. VanCourt, and M. Herbordt. *Three Dimensional Template Correlation: Object Recognition in 3D Voxel Data*. In Proceedings of the Computer Architecture for Machine Perception 2005.

[Van06] Tom VanCourt and Martin C. Herbordt. *Families of FPGA-Based Accelerators for Approximate String Matching*. Special Issue on FPGA-based Reconfigurable Computing, Journal of Microprocessors and Microsystems, to be published in 2006.

[Van06a] Tom VanCourt, Yonfeng Gu, Vikas Mundada, and Martin Herbordt. *Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws*. EURASIP Journal on Applied Signal Processing, to be published in 2006.

[Ver00] E. Vermeulen, F. Catthoor, D. Verkest, and H. De Man. *Formalized Three-Layer System-Level Reuse Model and Methodology for Data-Dominated Applications*. DATE 2000. IEEE/ACM

[Voe01] J. Voeten. *On the fundamental limitations of transformational design*. ACM Transactions on Design Automation of Electronic Systems 6(4)533–552. 2001

[VSI97] VSI Alliance. *Architecture Document Version 1.0*. VSI Alliance, Wakefield MA, 1997.

[W3C04] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/TR/2004/REC-xml-20040204/ (verified on 12 Sep 2005) February 2004.

[Wai97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Marua, J. Babb, S. Amarasinghe, and A. Agrawal, A. *Bring it all to software: Raw machines*. Computer pp. 86–93. Sept. 1997.

[Wal93] David W. Wall. *Limits of Instruction-Level Parallelism*. Digital Western Research Report 93/6. 1993.

[War03] Joa Warmer and Anneke Kleppe. *The Object Constraint Language, second edition*. Addison-Wesley, Boston. 2003

[Wat76] M. S. Waterman, T. F. Smith, and W. A. Beyer. *Some Biological Sequence Metrics*. Advances in Mathematics (Academic Press) 20(3)367-387 . 1976.

[Weg87] Peter Wegner. *Dimensions of Object-Based Programming Languages*. ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages and Applications 1987, pp.168-182

[Wir82] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, New York. 1982

[Wir95] Niklaus Wirth. *Circuit Design for Computer Science Students*. Springer-Verlag, Berlin. 1995

[Wir97] Michael J. Wirthlin and Brad L. Hutchings. *Improving Functional Density through Run-Time Constant Propagation*. Proceedings of the Field Programmable Gate Arrays, 1997.

[Wul71] W. A. Wulf, D. B. Russell, and A. N. Habermann. *BLISS: A Language for Systems Programming*. CACM 14(12):780-790. 1971

[Xil02] Xilinx, Inc. *Constraints Guide.* San Jose CA, 2002.

[Xil02a] Xilinx, Inc. *Development System Reference Guide – ISE 5*. San Jose CA, 2002

[Xil02b] Xilinx, Inc. *Xilinx Synthesis Technology (XST) User Guide*. San Jose CA, 2002.

[Xil02c] Xilinx, Inc. *Libraries Guide*. San Jose CA, 2002.

[Xil02d] Xilinx, Inc. *Xilinx System Generator™ for DSP*, version 3.1, User Guide. San Jose CA, 2002.

[Xil03] Xilinx, Inc (2003). *JBits 3.0 for Virtex II*. http://www.xilinx.com/labs/downloads/jbits/index.htm on 22 July 2005.

[Xil03a] Xilinx, Inc. *Forge High Level Language Compiler – Source Style Guide*. Columbia MD, 2003

[Xil03b] Xilinx, Inc. *Synthesis and Verification Design Guide*. 2003

[Xil04] Xilinx Inc. *Virtex-II Pro™ FPGAs: The Highest System Performance The Lowest System Cost*. Xilinx Inc. 2004

[Xil05] Xilinx Inc. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/overview/index.htm (verified on 10 Aug 2005).

[Ye00] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. *Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit*. International Symposium on Computer Architecture. 2000.

[Yu88] Kwang-I Yu, Shi-Ping Hsu, Lee Z. Hasiuk, and Peggy M. Otsubo. *Fast Search Processor*. U.S. Patent 4,760,523. 1988.

[Yu03] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong. *A Smith-Waterman Systolic Cell*. Proceedings of the 13th Field-Programmable Logic and Applications. Springer, Berlin. 2003.

[Yu05] Yi-Kuo Yu and Stephen F. Altschul. *The Construction of Amino Acid Substitution Matrices for the Comparison of Proteins with Non-Standard Compositions*. Bioinformatics 21(7)902-911. 2005.

VITA

Thomas David VanCourt

3 Marshall Place
Charlestown MA 02129

Formal Education

1978	BS Engineering, College of Engineering, Cornell University
2001	MS Computer Science, Metropolitan College, Boston University Thesis: *Reverse Engineering of Design Patterns from Compiled Programs*. Was granted the Metropolitan College Award for Excellence in Graduate Study

Refereed Journal Articles

Tom VanCourt and Martin C. Herbordt (2005). *Families of FPGA-Based Accelerators for Approximate String Matching*. Special Issue on FPGA-based Reconfigurable Computing, Journal of Microprocessors and Microsystems, accepted for publication in 2006.

Tom VanCourt, Yongfeng Gu, Vikas Mundada, and Martin Herbordt (2005). *Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws*. EURASIP Journal on Applied Signal Processing, accepted for publication in 2006.

Yongfeng Gu, Tom VanCourt, and Martin C. Herbordt. *Accelerating Molecular Dynamics Simulations with Configurable Circuits*. IEE Proceedings on Computers & Digital Techniques, accepted for publication in 2006.

T. VanCourt, M. C. Herbordt and R. J. Barton (2004), *Case Study of a Functional Genomics Application for an FPGA-Based Coprocessor*. Microprocessors and Microsystems 28(4)213-222, special issue on FPGA Applications, Algorithms, and Tools.

Articles in Refereed Conference Proceedings

T. VanCourt and M.C.Herbordt. *LAMP: A Tool Suite for Families of FPGA-based Computation Accelerators*. Proceedings of Field Programmable Logic and Applications (FPL), August 2005

Y. Gu, T. VanCourt, and M.C. Herbordt (2005). *Accelerating Molecular Dynamics Simulations with Configurable Circuits*, Proc. FPL, August 2005

Y. Gu, T. VanCourt, D. DiSabello, and M.C. Herbordt (2005). *FPGA Acceleration of a Molecular Dynamics Application*, Symposium on Field Programmable Custom Computing Machines (FCCM) 2005

T. VanCourt, Y. Gu, and M. C. Herbordt (2005). *Three-Dimensional Template Correlation: Object Recognition in Voxel Data* (preliminary version), Proceedings of Computer Architecture for Machine Perception (CAMP), July 2005

T. VanCourt and M.C. Herbordt (2004). *Families of FPGA-Based Algorithms for Approximate String Matching*, Proceedings of Application-Specific Systems, Architectures, and Processors (ASAP), September 2004

T. VanCourt, Y. Gu, and M.C. Herbordt (2004). *FPGA Acceleration of Rigid Molecule Interactions*, Proc. FPL, September 2004. Lecture Notes in Computer Science, 3203, J. Becker, et al., editors, Springer Verlag

A. Conti, T. VanCourt, and M.C. Herbordt (2004). *Processing Repetitive Structures with Mismatches at Streaming Rate*, Proc. FPL, September 2004. Also in Lecture Notes in Computer Science, 3203, J. Becker, et al., editors, Springer Verlag

T. VanCourt, Y. Gu, and M.C. Herbordt (2004). *FPGA Acceleration of Rigid Molecule Interactions* (preliminary version), Proc. FCCM, April 2004.

T. VanCourt, M.C. Herbordt, and R.J. Barton (2003). *Case Study of a Functional Genomics Application for an FPGA-Based Coprocessor*, Proc. FPL, September 2003 pp. 365-374. Also in Lecture Notes in Computer Science, 2778, P.Y.K. Cheung, et al., editors, Springer Verlag.

Workshop Presentations without Proceedings

T. VanCourt and M. C. Herbordt. *Requirements for any HPC/FPGA Application Development Tool Flow*. Fourth Boston Area Computer Architecture Workshop, January 2006

T. VanCourt and M. C. Herbordt, *Making FPGAs a Cost-Effective Computing Architecture*, Third Boston Area Computer Architecture Workshop, January 2005.

T. VanCourt and M. C. Herbordt, *Processor-Memory Networks Based on Steiner Systems*, presented at Second Boston Area Computer Architecture Workshop, January 2004.

Book, Supplementary Material

E. Braude (2000), Software Engineering, an Object-Oriented Perspective, John Wiley & Sons Inc. Created the sample program (~40 Java files, ~8000 commented source lines) used as the book's case study, including tests, graphics, and maintenance documentation.

Invited Seminars

FPGA Application Accelerators: More than Logic Design. Imperial College UK. July 2005.

Opportunities for the Application of Special-Purpose Computing to Protein-Ligand Docking, Mercury Computer System Inc., September 2004

Families of FPGA-Based Algorithms for Approximate String Matching: A Case Study in Flexible Design, University of Houston, September 2004

Invited Articles

Template matching in three dimensions: FPGAs make it practical. SPIE Newswatch, to be published 2006

Reverse Engineering Design Patterns from Compiled Programs. Boston University, Metropolitan College, Department of Computer Science Technical Report BUMETCS TR-2001-001, based on MS thesis.

Appointments

Fall 2002 – present. Research Assistant. Computer Architecture and Automated Design Laboratory. Boston University, Electrical and Computer Engineering

Spring 2001 – Spring 2006. Instructor. Department of Computer Science, Metropolitan College, Boston University. Taught or will teach MET courses CS575, CS665, CS673, TC650

Spring 2004 – Fall 2004. Consultant. Mercury Computing Systems

Spring 2002. Senior Software Engineer (founder). Silicon Keep Inc., Lexington MA

2001 –2002. Senior Software Engineer. InterTrust Technologies Corp., Billerica MA

1993 – 1997. Principal Engineer. PixelVision Inc., Acton MA

1991 – 1993. Principal Engineer. PictureTel Corporation, Danvers MA

1983 – 1991. Senior Software Engineer. Apollo/Hewlett Packard Company, Chelmsford MA

1978 – 1983. Software Engineer. Digital Equipment Corporation, Maynard MA

Summer 1977. Intern. IBM Corporation, Poughkeepsie NY

Patent Activity

T. VanCourt and M. C. Herbordt (2004). Provisional Patent Application No. 60/549,946. *System and Method for Programmable-Logic Acceleration of Bioinformatics and Computational Biology Applications*.

Service

Technical review: revised edition of “UML and the Unified Process”, J. Arlow and I. Neustadt, Pearson Education Ltd.

Reviewed submissions: Computer Architecture for Machine Perception 2005

Reviewed submissions: Field Programmable Logic and Applications 2003

Standards Development: Submissions to ANSI X3T9.5 (FDDI) standards committee, 1987.

Member, Software Engineering Curriculum Committee 2001-2005: Computer Science Department, Metropolitan College, Boston University.

Book reviews: Middle school and high school mathematics texts for The Textbook League

Reader for Recording for the Blind and Dyslexic 2000-2003. Approved for specialist readings in math, statistics, and computing.