A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems

Byoungro So, Mary W. Hall and Pedro C. Diniz

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{bso,mhall,pedro}@isi.edu

ABSTRACT

This paper describes an automated approach to hardware design space exploration, through a collaboration between parallelizing compiler technology and high-level synthesis tools. We present a compiler algorithm that automatically explores the large design spaces resulting from the application of several program transformations commonly used in application-specific hardware designs. Our approach uses synthesis estimation techniques to quantitatively evaluate alternate designs for a loop nest computation. We have implemented this design space exploration algorithm in the context of a compilation and synthesis system called DEFACTO, and present results of this implementation on five multimedia kernels. Our algorithm derives an implementation that closely matches the performance of the fastest design in the design space, and among implementations with comparable performance, selects the smallest design. We search on average only 0.3% of the design space. This technology thus significantly raises the level of abstraction for hardware design and explores a design space much larger than is feasible for a human designer.

Categories and Subject Descriptors

D.3.4 [**Compilers**]: Parallelizing Compilers; B.5.2 [**Designs Aids**]: Automatic Synthesis; D.7.1 [**Types and Design Styles**]: Algorithms implemented in hardware, Gate Arrays

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Design Space Exploration, Data Dependence Analysis, Reuse Analysis, Loop Transformations

PLDI'02, June 17-19, 2002, Berlin, Germany.

1. INTRODUCTION

The extreme flexibility of Field Programmable Gate Arrays (FPGAs) has made them the medium of choice for fast hardware prototyping and a popular vehicle for the realization of custom computing machines. FPGAs are composed of thousands of small programmable logic cells dynamically interconnected to allow the implementation of any logic function. Tremendous growth in device capacity has made possible implementation of complex functions in FPGAs. For example, FPGA implementations can sometimes yield even faster solutions than conventional hardware, up to 2 orders of magnitude on encryption [18]. In addition FPGAs offer a much faster time to market for time-critical applications and allow post-silicon in-field modification to prototypical or low-volume designs where an Application Specific Integrated Circuit (ASIC) is not justified.

Despite growing importance of application-specific FPGA designs, these devices are still difficult to program making them inaccessible to the average developer. The standard practice requires developers to express the application in a hardware-oriented language such as Verilog or VHDL, and synthesize the design to hardware using a wide variety of synthesis tools. As optimizations performed by synthesis tools are very limited, developers must perform high-level and global optimizations by hand. For example, no commercially-available high-level synthesis tool handles multi-dimensional array variables[1] nor automatic selection of loop unroll factors.

Because of the complexity of synthesis, it is difficult to predict a priori the performance and space characteristics of the resulting design. For this reason, developers engage in an iterative refinement cycle, at each step manually applying transformations, synthesizing the design, examining the results, and modifying the design to trade off performance and space. Throughout this process, called *design space exploration*, the developer carries the responsibility for the correctness of the application mapping.

We believe the way to make programming of FPGA-based systems more accessible is to offer a high-level imperative programming paradigm, such as C, coupled with compiler technology oriented towards FPGA designs. In this way, developers retain the advantages of a simple programming

[1]Several claim the support of this feature but only for simulation purposes, not actual hardware synthesis.

model via the high-level language but rely on powerful compiler analyses and transformations to optimize the design as well as automate most of the tedious and error-prone mapping tasks. We make the observation that, for a class of FPGA applications characterized as highly parallel array-based computations (*e.g.,* multimedia codes), many hand optimizations performed by developers are similar to transformations used in parallelizing compilers. For example, developers parallelize computations, optimize external memory accesses, explicitly manage storage and perform loop transformations. For this reason, we argue that parallelizing compiler technology can be used to optimize FPGA designs.

In this paper, we describe an automated approach to design space exploration, based on a collaboration between a parallelizing compiler and high-level synthesis tools. Completely synthesizing a design is prohibitively slow (hours to days) and further, the compiler must try several designs to arrive at a good solution. For these reasons, we exploit estimation from behavioral synthesis to determine specific hardware parameters (e.g., size and speed) with which the compiler can quantitatively evaluate the application of a transformation to derive an optimized and feasible implementation of the loop nest computation. Since the hardware implementation is bounded in terms of capacity, the compiler transformations must also consider space constraints. This compiler algorithm effectively enables developers to explore a potentially large design space, which without automation would not be feasible.

In previous work, we presented an overview of DEFACTO, the system upon which this work is based, which combines parallelizing compiler technology in the Stanford SUIF compiler with hardware synthesis tools [9]. In this paper, we present a detailed algorithm for design space exploration and results demonstrating its effectiveness. While there are a few systems that automatically synthesize hardware designs from C specifications [24], to our knowledge there is no other system that automatically explores the design space in collaboration with behavioral synthesis estimation features. Our current infrastructure largely supports the direct mapping of computations to multiple FPGAs [26]. However, the work in this paper describes an implementation and experimental results for designs that are mapped to a single FPGA and multiple memories. We thus focus on the algorithmic aspects of design space exploration under simpler data and computation partitioning strategies.

This paper makes the following specific contributions.

- Describes the integration of behavioral synthesis tools and parallelizing compiler technology to map computations to FPGA-based architectures. We present a compiler algorithm for design space exploration that relies on behavioral synthesis estimates. The algorithm applies loop transformations to explore a space-time trade-off in the realization of hardware designs.
- Defines a *balance* metric for guiding design space exploration, which suggests when it is profitable to devote more resources to storage or computation. The design space exploration algorithm exploits monotonicity properties of the balance metric to effectively prune large regions of the search space, thereby allowing the compiler to consider a wider range of transformations that otherwise would not be feasible.
- Presents experimental results for five multimedia kernels. Our algorithm derives an implementation that closely matches the performance of the fastest design in the design space, and among implementations with comparable performance, selects the smallest design. We search on average only 0.3% of the design space.

As technology advances increase density of FPGA devices, tracking Moore's law for conventional logic of doubling every 18 months, devices will be able to support more sophisticated functions. With the future trend towards on-chip integration of internal memories, FPGAs with special-purpose functional units are becoming attractive as a replacement for ASICs and for custom embedded computing architectures. We foresee a growing need to combine the strengths of high-level program analysis techniques, to complement the capabilities of current and future synthesis tools. Devices and consequently designs will become more complex, demanding an efficient solution to exploring even larger design spaces.

The remainder of the paper is organized as follows. In the next section we present some background on FPGAs and behavioral synthesis. Section 3 describes the optimization goal of our design space exploration algorithm in mapping loop nest computations to hardware. In Section 4 we discuss the analyses and transformation our algorithm uses. In Section 5 we present the design space exploration algorithm. In Section 6 we present experimental results for the application of this algorithm to 5 image processing computations. We survey related work in Section 7 and conclude in Section 8.

2. BACKGROUND

We now describe FPGAs and synthesis, and compare with optimizations performed in parallelizing compilers. We also discuss features of our target application domain.

2.1 Field-Programmable-Gate-Arrays

FPGAs are a popular vehicle for rapid prototyping or as a way to implement simple logic interfaces. FPGAs are implemented as (re)programmable seas-of-gates with distinct internal architectures. For example, the Xilinx Virtex family of devices consists of $12,288$ device *slices* where each slice in turn is composed of 2 look-up tables (LUTs) each of which can implement an arbitrary logic function of 11 boolean inputs and 6 outputs [15]. Two slices form a configurable logic block (CLBs) and these blocks are interconnected in a 2-dimensional mesh via programmable static routing switches. To configure an FPGA, designers have to download a *bitstream* file with the configuration of all slices in the FPGA as well as the routing. Other programmable devices, for example the APEX II devices from Altera, have a more hierarchical routing approach to connecting the CLBs in their FPGAs, but the overall functionality is similar [6].

As with traditional architectures, bandwidth to external memory is a key performance bottleneck in FPGAs, since it is possible to compute orders of magnitude more data in a cycle than can be fetched from or stored to memory. However, unlike traditional architectures, an FPGA has the flexibility to devote its internal configurable resources either to storage or to computation.

2.2 FPGA Synthesis Flow

Synthesis flow for FPGAs is the term given to the process of translating functional logic specifications to a bitstream description that configures the device. This functional specification can be done at multiple levels. Using hardware

description languages such as VHDL or Verilog, designers can specify the functionality of their datapath circuits (*e.g.*, adders, multipliers, etc.) as a diagram of design blocks. This *structural specification* defines the input/output interface for each block and allows the designers to describe finite state machines (FSMs) to control the temporal behavior of each of the blocks. Using this approach designers can control every single aspect of operations in their datapaths. This is the preferred approach when maximum performance is sought, but requires extremely high design times.

The process that takes a structural specification and targets a particular architecture's programmable units (LUTs in the case of Xilinx devices) is called *RTL-level synthesis*. The *RTL-level synthesis* generates a *netlist* representation of the intended design, used as the input of low-level synthesis steps such as the mapping and place-and-route (P&R) to ultimately generate the device bitstream configuration file.

2.3 Behavioral Synthesis vs. Compilers

Behavioral specifications in VHDL or Verilog, as opposed to lower level structural specifications, express computations without committing to a particular hardware implementation structure. The process of taking a behavioral specification and generating a hardware implementation is called *behavioral synthesis*. Behavioral synthesis performs three core functions:

- **binding** operators and registers in the specification to hardware implementations (*e.g.*, selecting a ripple-carry adder to implement an addition);
- resource **allocation** (*e.g.*, deciding how many ripple-carry adders are needed); and,
- **scheduling** operations in particular clock cycles.

To generate a particular implementation, behavioral synthesis requires the programmer to specify the target design requirements in terms of area, clock rate, number of clock cycles, number of operators, or some combination. For example, the designer might request a design that uses two multipliers and takes at most 10 clock cycles. Behavioral synthesis tools use this information to generate a particular implementation that satisfies these constraints.

In addition, behavioral synthesis supports some optimizations, but relies heavily on the developer to direct some of the mapping steps. For example, current behavioral synthesis tools allow the specification of which loops to unroll. After loop unrolling, the tool will perform extensive optimizations on the resulting inner loop body, such as parallelizing and pipelining operations and minimizing registers and operators to save space. However, deciding the unroll factor is left up to the programmer.

Behavioral Synthesis	Parallelizing Compilers
Optimizations only on scalar variables	Optimizations on scalars and arrays
Optimizations only inside loop body	Optimizations inside loop body and across loop iterations
Supports user-controlled loop unrolling	Analyses guide automatic loop transformations
Manages registers and inter-operator communication	Optimizes memory accesses Evaluates trade-offs of different storage on- and off-chip
Considers only single FPGA	System-level view: multiple FPGAs multiple memories
Performs allocation, binding and scheduling of hardware resources	No knowledge of hardware implementation of computation

Table 1: Comparison of Behavioral Synthesis and Parallelizing Compiler Technologies.

While there are some similarities between the optimizations performed by synthesis tools and parallelizing compilers, in many ways they offer complementary capabilities, as shown in Table 1. The key advantage of parallelizing compiler technology over behavioral synthesis is the ability to perform data dependence analysis on array variables, used as a basis for parallelization, loop transformations and optimizing memory accesses. This technology permits optimization of designs with array variables, where some data resides in off-chip memories. Further, it enables reasoning about the benefits of code transformations (such as loop unrolling) without explicitly applying them. In addition, parallelizing compilers are capable of performing global program analysis, which permits optimization across the entire system.

2.4 Target Application Domain

Because of their customizability, FPGAs are commonly used for applications that have significant amounts of fine-grain parallelism and possibly can benefit from non-standard numeric formats (*e.g.,* reduced data widths). Specifically, multimedia applications, including image and signal processing on 8-bit and 16-bit data, respectively, offer a wide variety of popular applications that map well to FPGAs.

For example, a typical image processing algorithm scans a multi-dimensional image and operates on a given pixel value and all its neighbors. Images are often represented as multi-dimensional array variables, and the computation is expressed as a loop nest. Such applications exhibit abundant concurrency as well as temporal reuse of data. Examples of computations that fall into this category include image correlation, Laplacian image operators, erosion/dilation operators and edge detection.

Fortunately, such applications are a good match for the capabilities of current parallelizing compiler analyses, which are most effective in the *affine* domain, where array subscript expressions are linear functions of the loop index variables and constants [25]. In this paper, we restrict input programs to loop nest computations on array and scalar variables (no pointers), where all subscript expressions are affine with a fixed stride. The loop bounds must be constant.[2] We support loops with control flow, but to simplify control and scheduling, the generated code always performs conditional memory accesses.

3. OPTIMIZATION GOAL AND BALANCE

Simply stated, the optimization criteria for mapping a single loop nest to FPGA-based systems are as follows: (1) the design must not exceed the capacity constraints of the system; (2) the execution time should be minimized; and, (3) for a given level of performance, FPGA space usage should be minimized. The motivation for the first two criteria should be obvious, but the third criterion is also needed for several reasons. First, if two designs have equivalent performance, the smaller design is more desirable, in that it frees up space for other uses of the FPGA logic, such as to map other loop nests. In addition, a smaller design usually has less routing complexity, and as a result, may achieve a faster

[2]Non-constant bounds could potentially be supported by the algorithm, but the generated code and resulting FPGA designs would be much more complex. For example, behavioral synthesis would transform a `for` loop with a non-constant bound to a `while` loop in the hardware implementation.

target clock rate. Moreover, the third criterion suggests a strategy for selecting among a set of candidate designs that meet the first two criteria.

With respect to a particular set of transformations, which are described in the next section, our algorithm attempts to select the best design that meets the above criteria. The algorithm uses two metrics to guide the selection of a design. First, results of estimation provide space usage of the design, related to criterion 1 above. Another important metric used to guide the selection of a design, related to criteria 2 and 3, is *Balance*, defined by the equation.

$$Balance = F/C,$$

where F refers to the data fetch rate, the total data bits that memory can provide per cycle, and C refers to the data consumption rate, total data bits the computation can consume during the computational delay. If balance is close to one, both memories and FPGAs are busy. If balance is less than one, the design is memory bound; if greater than one, it is compute bound. When a design is not balanced, this metric suggests whether more resources should be devoted to improving computation time or memory time.

We borrow the notion of balance from previous work for mapping array variables to scalar registers to balance the floating point operations and memory accesses [5]. Because we have the flexibility in FPGAs to adjust time spent in either computation or memory accesses, we use the data fetch rate and data consumption rate, and compare them under different optimization assumptions.

4. ANALYSES AND TRANSFORMATIONS

This section describes at a high level the code transformations performed by our system, as illustrated by the FIR filter example in Figure 1.

Unroll-and-Jam. The first code transformation, *unroll-and-jam*, involves unrolling one or more loops in the iteration space and fusing inner loop bodies together, as shown in Figure 1(b). Unrolling exposes operator parallelism to high-level synthesis. In the example, all of the multiplies can be performed in parallel. Two additions can subsequently be performed in parallel, followed by two more additions. Unroll-and-jam can also decrease the dependence distances for reused data accesses, which, when combined with scalar replacement discussed below, can be used to expose opportunities for parallel memory accesses.

Scalar Replacement. *Scalar replacement* replaces array references by accesses to temporary scalar variables, so that high-level synthesis will exploit reuse in registers [5]. Our approach to scalar replacement closely matches previous work, which eliminates true dependences when reuse is carried by the innermost loop, for accesses in the affine domain with consistent dependences (*i.e.,* constant dependence distances) [5]. There are, however, two differences: (1) we also eliminate unnecessary memory writes on output dependences; and, (2) we exploit reuse across all loops in the nest, not just the innermost loop. The latter difference stems from the observation that many, though not all, algorithms mapped to FPGAs have sufficiently small loop bounds or small reuse distances, and the number of registers that can be configured on an FPGA is sufficiently large. A more detailed description of our scalar replacement and register reuse analysis can be found in [9].

In the example in Figure 1(c), we see the results of scalar replacement, which illustrates some of the above differences

```
int S[96];
int C[32];
int D[64];
for (j=0; j<64; j++)
  for(i = 0; i<32; i++)
    D[j] = D[j] + (S[i+j] * C[i]);
```
(a) Original code.

```
for (j=0; j<64; j+=2)
  for(i = 0; i<32; i+=2){
    D[j] = D[j] + (S[i+j] * C[i]);
    D[j] = D[j] + (S[i+j+1] * C[i+1]);
    D[j+1] = D[j+1] + (S[i+j+1] * C[i]);
    D[j+1] = D[j+1] + (S[i+j+2] * C[i+1]);
  }
```
(b) After unrolling j loop and i loop by 1 (unroll factor 2) and jamming copies of i loop together.

```
for (j=0; j<64; j+=2) { /* initialize D registers */
  d_0 = D[j];
  d_1 = D[j+1];
  for (i=0; i<32; i+=2) {
    if (j==0) { /* initialize C registers */
      c_0_0 = C[i];
      c_1_0 = C[i+1];
    }
    S_0 = S[i+j+1];
    d_0 = d_0 + S[i+j] * c_0_0; /* unroll(0,0) */
    d_0 = d_0 + S_0 * c_1_0; /* unroll(0,1) */
    d_1 = d_1 + S_0 * c_0_0; /* unroll(1,0) */
    d_1 = d_1 + S[i+j+2] * c_1_0; /* unroll(1,1) */
    rotate_registers(c_0_0, ... ,c_0_15);
    rotate_registers(c_1_0, ... ,c_1_15);
  }
  D[j] = d_0;
  D[j+1] = d_1;
}
```
(c) After scalar replacement of accesses to C and D across both i and j loop.

```
for (j=0; j<32; j++) { /* initialize D registers */
  d_0 = D2[j];
  d_1 = D3[j];
  for (i=0; i<16; i++) {
    if (j==0) { /* initialize C registers */
      c_0_0 = C0[i];
      c_1_0 = C1[i];
    }
    S_0 = S1[i+j];
    d_0 = d_0 + S0[i+j] * c_0_0; /* unroll(0,0) */
    d_0 = d_0 + S_0 * c_1_0; /* unroll(0,1) */
    d_1 = d_1 + S_0 * c_0_0; /* unroll(1,0) */
    d_1 = d_1 + S0[i+j+1] * c_1_0; /* unroll(1,1) */
    rotate_registers(c_0_0, ... ,c_0_15);
    rotate_registers(c_1_0, ... ,c_1_15);
  }
  D3[j] = d_1;
  D2[j] = d_0;
}
```
(d) Final code generated for FIR, including loop normalization and data layout optimization.

Figure 1: Optimization Example: FIR.

from previous work. Accesses to arrays C and D can all be replaced. The D array is written back to memory at the end of the iteration of the j loop, but redundant writes are eliminated. Only loop-independent accesses to array S are replaced because the other accesses to array S do not have a consistent dependence distance. Because reuse on array C is carried by the outer loop, to exploit full reuse of data from C involves introducing extra registers that hold values of C across all iterations of the inner loop. The *rotate* operation shifts the registers and rotates the last one into the first position; this operation can be performed in parallel in hardware.

Loop Peeling and Loop-Invariant Code Motion. We see in Figure 1(c) and (d) that values for the c registers are loaded on the first iteration of the j loop. For clarity it is not shown here, but the code generated by our compiler actually peels the first iteration of the j loop instead of including these conditional loads so that other iterations of the j loop have the same number of memory loads and can be optimized and scheduled by high-level synthesis accordingly. Although at first glance the code size appears to be doubled by peeling, high-level synthesis will usually reuse the operators between the peeled and original loop body, so that the code growth does not correspond to a growth in the design. Memory accesses to array D are invariant with respect to the i loop, so they are moved outside the loop using loop-invariant code motion. Within the main unrolled loop body, only memory accesses to array S remain.

Data Layout and Array Renaming. Another code transformation lays out the data in the FPGA's external memory so as to maximize memory parallelism. Custom data layout is separated into two distinct phases. In the first phase, which we call *array renaming*, performs a 1-to-1 mapping between array access expressions and virtual memory ids, to customize accesses to each array according to their access patterns. Array renaming can only be performed if all accesses to the array within the loop nest are *uniformly generated*. Two affine array references $A(a_1 i_1 + b_1, \ldots, a_n i_n + b_n)$ and $A(c_1 i_1 + d_1, \ldots, c_n i_n + d_n)$, where $a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots c_n, d_1, \ldots, d_n$ are constants and $i_1, \ldots, i_n$ are loop index variables, are uniformly generated if $\forall_{i=1,n} a_i = c_i$. If an array's accesses are not uniformly generated, then it is mapped to a single memory. The result of array renaming is an even distribution of data across the virtual memories.

The second phase, called *memory mapping*, binds virtual memory ids to physical ids, taking into consideration accesses by other arrays in the loop nest to avoid scheduling conflicts. As shown in Figure 1(d), the effect of data layout is that even elements of S and C are mapped to memory 0, and odd elements are mapped to memory 1, with accesses renamed to reflect this layout. D is similarly distributed to memories 2 and 3.

This approach is similar in spirit to the modulo unrolling used in the RAW compiler [3]. However, as compared to modulo unrolling, which is a loop transformation that assumes a fixed data layout, our approach is a data transformation. Further, our current implementation supports a more varied set of custom data layouts. A typical layout is cyclic in at least one dimension of an array, possibly more, but more customized data layouts arise from packing small data types, strided accesses, and subscript expressions with multiple induction variables (i.e., subscripts of the form $a_0 i_0 + a_1 i_1 + \ldots + a_n i_n + b$, where $n > 0$ and more than one a_i is non-zero). A full discussion of the data layout algorithm is beyond the scope of this paper, but further discussion can be found in [9].

Summary. To summarize, the algorithm evaluates a focused set of possible unroll factors for multiple loops in the loop nest. Data reuse is exploited within and across the loops in the nest, as a result of scalar replacement by the compiler, eliminating unnecessary memory accesses. Operator parallelism is exposed to high-level synthesis through the unrolling of one or more loops in the nest; any independent operations will be performed in parallel if high-level synthesis deems this beneficial.

Thus, we have defined a set of transformations, widely used in conventional computing, that permit us to adjust parallelism and data reuse in FPGA-based systems through a collaboration between parallelizing compiler technology and high-level synthesis. To meet the optimization criteria set forth in the previous section, we have reduced the optimization process to a tractable problem, that of selecting the unroll factors for each loop in the nest that leads to a high-performance, balanced, efficient design. In the next section, we present the algorithm in detail.

Although our algorithm focuses on a fixed set of compiler transformations, the notion of using balance to guide the performance-space tradeoff in design space exploration can be used for other optimizations as well.

5. OPTIMIZATION ALGORITHM

The discussion in this section defines terms and uses these to describe the design space exploration algorithm. The algorithm is presented assuming that scalar replacement will exploit all reuse in the loop nest on affine array accesses. The resulting design will store all reused data internally on the FPGA, which is feasible for many applications with short reuse distances, but may require too many on-chip registers in the general case. We address this problem by limiting the number of registers in Section 5.4.

5.1 Definitions

We define a *saturation point* as a vector of unroll factors where the memory parallelism reaches the bandwidth of the architecture, such that the following property holds for the resulting unrolled loop body:

$$\sum_{i \in \text{Reads}} width_i = C_1 * \sum_{l \in \text{NumMemories}} width_l.$$

$$\sum_{j \in \text{Writes}} width_j = C_2 * \sum_{l \in \text{NumMemories}} width_l.$$

Here, C_1 and C_2 are integer constants. To simplify this discussion, let us assume that the access widths match the memory width, so that we are simply looking for an unroll factor that results in a multiple of $NumMemories$ read and write accesses for the smallest values of C_1 and C_2. The saturation set, Sat, can then be determined as a function of the number of read and write accesses, R and W, in a single iteration of the loop nest and the unroll factor for each loop in the nest. We consider reads and writes separately because they will be scheduled separately.

We are interested in determining the saturation point after scalar replacement and redundant write elimination. For the

purposes of this discussion, we assume that for each array accessed in the main loop body, all accesses are uniformly generated, and thus a customized data layout will be obtained; modifications to the algorithm when this does not hold are straightforward, but complicate the calculation of the saturation point. R is defined as the number of uniformly generated read sets. W is the number of uniformly generated write sets. That is, there is a single memory read and single write access for each set of uniformly generated references because all others will be removed by scalar replacement or redundant write elimination.

We define an unroll factor vector as $U = \langle u_1, \ldots, u_n \rangle$, where u_i corresponds to the unroll factor for loop i, and a function $P(U) = \prod_{1 \leq i \leq n} u_i$, which is the product of all the unroll factors. Let $P_{sat} = lcm(gcd(R, W), NumMemories)$. The saturation set Sat can then be defined as a vector whose product is P_{sat}, where $\forall u_i \neq 1$, array subscript expressions for memory accesses are varying with respect to loop i. That is, the saturation point considers unrolling only those loops that will introduce additional memory parallelism. Since loop peeling and loop-invariant code motion have eliminated memory accesses in the main loop body that are invariant with respect to any loop in the nest, from the perspective of memory parallelism, all such unroll factor vectors are equivalent. A particular saturation point Sat_i refers to unrolling the i loop by the factor P_{sat}, and using an unroll factor of 1 for all other loops.

5.2 Search Space Properties

The optimization involves selecting unroll factors for the loops in the nest. Our search is guided by the following observations about the impact of unrolling a single loop in the nest, which depend upon the assumptions about target applications in Section 2.4.

OBSERVATION 1. *The data fetch rate is monotonically nondecreasing as the unroll factor increases by multiples of P_{sat}, but it is also nonincreasing beyond the saturation point.*

Intuitively, the data fetch rate increases as there are more memory accesses available in the loop body for scheduling in parallel. This observation requires that the data is laid out in memory and the accesses are scheduled such that the number of independent memory accesses on each memory cycle is monotonically nondecreasing as the unroll factor increases. Here, the unroll factor must increase by multiples of P_{sat}, so that each time a memory operation is performed, there are $NumMemories$ accesses in the main loop body that are available to schedule in parallel. This is true whenever data layout has successfully mapped each array to multiple memories. (If data layout is not successful, as is the case when not all accesses to the same array are uniformly generated, a steady state mapping of data to memories can guarantee monotonicity even when there are less than $NumMemories$ parallel accesses, but we will ignore this possibility in the subsequent discussion.)

Data layout and mapping data to specific memories is controlled by the compiler. Given the property of array renaming from Section 4, that the accessed data is evenly distributed across virtual memory ids, this mapping derives a solution that, in the absence of conflicting accesses to other arrays, exposes fully parallelizable accesses. To prevent conflicting read or write accesses in mapping virtual memory ids to physical ones, we must first consider how accesses will be scheduled. The compiler component of our system is not directly responsible for scheduling; scheduling memory accesses as well as computation is performed by behavioral synthesis tools such as Monet.

The scheduling algorithm used by Monet, called *As Soon As Possible*, first considers which memory accesses can occur in parallel based on comparing subscript expressions and physical memory ids, and then rules out writes whose results are not yet available due to dependences [10]. In performing the physical memory id mapping, we first consider read accesses, so that we maximize the number of read operations that can occur in parallel. The physical id mapping matches the read access order, so that the total number of memory reads in the loop is evenly distributed across the memories for all arrays. As an added benefit, operands for individual writes are fetched in parallel. Then the physical mapping for write operations is also performed in the same order, evenly distributing write operations across the memories.

With these properties of data layout and scheduling, at the saturation point, we have guaranteed through choice of unroll factor that the data fetch rate increases up to the saturation point, but not beyond it.

OBSERVATION 2. *The consumption rate is monotonically non-decreasing as the unroll factor increases by multiples of P_{sat}, even beyond the saturation point.*

Intuitively, as the unroll factor increases, more operator parallelism is enabled, thus reducing the computation time and increasing the frequency at which data can be consumed. Further, based on Observation 1, as we increase the data fetch rate, we eliminate idle cycles waiting on memory and thus increase the consumption rate. Although the parallelism exploited as a result of unrolling a loop may reach a threshold, performance continues to improve slightly due to simpler loop control.

OBSERVATION 3. *Balance is monotonically nondecreasing before the saturation point and monotonically nonincreasing beyond the saturation point as the unroll factor increases by multiples of P_{sat}.*

That balance is nondecreasing before the saturation point relies on Observation 1. The data fetch rate is increasing as fast as or faster than the data consumption rate because memory accesses are completely independent, whereas operator parallelism may be restricted. Beyond the saturation point, the data fetch rate is not increasing further, and the consumption rate is increasing at least slightly.

5.3 Algorithm Description

The algorithm is presented in Figure 2. Given the above described monotonicity of the search space for each loop in the nest, we start with a design at the saturation point, and we search larger unroll factors that are multiples of P_{sat}, looking for the two points between which balance crosses over from compute bound to memory bound, or vice versa. In fact, ignoring space constraints, we could search each loop in the nest independently, but to converge to a near-optimal design more rapidly, we select unroll factors based on the data dependences, as described below.

The algorithm first selects U_{init}, the starting point for the search, which is in Sat. We select the most promising unroll factors based on the dependence *distance vectors*. A dependence distance vector is a vector $d = \langle d_1, d_2, \ldots, d_n \rangle$ which

represents the vector difference between two accesses to the same array, in terms of the loop indices in the nest [25]. Since we are starting with a design that maximizes memory parallelism, then either the design is memory bound and we stop the search, or it is compute bound and we continue. If it is compute bound, then we consider unroll factors that provide increased operator parallelism, in addition to memory parallelism. Thus, we first look for a loop that carries no dependence (*i.e.,* $\forall_{d \in D} d_i = 0$). All unrolled iterations of such a loop can be executed in parallel. If such a loop i is found, then we set the unroll factor to Sat_i. assuming this unroll factor is in Sat.

If no such loop exists, then we instead select an unroll factor that favors loops with the largest dependence distances, because such loops can perform in parallel computations between dependences. The details of how our algorithm selects the initial unroll factor in this case is beyond the scope of this paper, but the key insight is that we unroll all loops in the nest, with larger unroll factors for the loops carrying larger minimum nonzero dependence distances. The monotonicity property also applies when considering simultaneous unrolling for multiple loops as long as unroll factors for all loops are either increasing or decreasing.

If the initial design is space constrained, we must reduce the unroll factor until the design size is less than the size constraint *Capacity*, resulting in a suboptimal design. The function *FindLargestFit* simply selects the largest unroll factor between the baseline design corresponding to no unrolling (called U_{base}), and U_{init}, regardless of balance, because this will maximize available parallelism.

Assuming the initial design is compute bound, the algorithm increases the unroll factors until it reaches a design that is (1) memory bound; (2) larger than *Capacity*; or, (3) represents full unrolling of all loops in the nest (*i.e.,* $U_{curr} = U_{max}$), as follows.

The function *Increase*(U_{in}) returns unroll factor vector U_{out} such that

$$(1) P(U_{out}) = 2 * P(U_{in}); \text{ and,}$$
$$(2) \forall_i u_i^{in} \leq u_i^{out} \leq u_i^{max}.$$

If there are no such remaining unroll factor vectors, then *Increase* returns U_{in}.

If either a space-constrained or memory bound design is found, then the algorithm will select an unroll factor vector between the last compute bound design that fit, and the current design, approximating binary search, as follows.

The function *SelectBetween*(U_{small}, U_{large}) returns the unroll factor vector U_{out} such that

$$(1) P(U_{out}) = (P(U_{small}) + P(U_{large})/2;$$
$$(2) \forall_i u_i^{small} \leq u_i^{out} \leq u_i^{large}; \text{ and,}$$
$$(3) P(U_{out}) = C * P(U_{init}), \text{ for some constant } C.$$

If there are no such remaining unroll factor vectors, then *SelectBetween* returns U_{small}, a compute bound design.

5.4 Adjusting Number of On-Chip Registers

For designs where the reuse distance is large and many registers are required, it may become necessary to reduce the number of data items that are stored on the FPGA. Using fewer on-chip registers means that less reuse is exploited, which in turn slows down the fetch rate and, to a lesser extent, the consumption rate. The net effect is that, in the first place, the design will be smaller and more likely to fit on chip, and secondly, space is freed up so that it can be used to increase the operator parallelism for designs that are compute bound.

To adjust the number of on-chip registers, we can use loop tiling to tile the loop nest so that the localized iteration space within a tile matches the desired number of registers, and exploit full register reuse within the tile.

```
Search Algorithm:
Input:  Code          /* An n-deep loop nest */
Output: ⟨u_1, ..., u_n⟩ /* a vector of unroll factors */

U_curr = U_init
U_mb = U_max
ok = False
while (!ok) do
 Code = Generate(U_curr)
 Estimate = Synthesize(Code)
 B = Balance(Code,Estimate.Performance)
 /* first deal with space-constrained designs */
 if (Estimate.Space > Capacity) then
    if (U_curr = U_init) then
       U_curr = FindLargestFit(U_base, U_curr)
       ok = True
    else
       U_curr = SelectBetween(U_cb, U_curr)
 else if (B = 1) then ok = True /* Balanced, so DONE! */
 else if (B < 1) then /* memory bound */
    U_mb = U_curr
    if (U_curr = U_init) then ok = True
    else
       /* Balanced solution is between earlier size and this */
       U_curr = SelectBetween(U_cb, U_mb)
 else if (B > 1) then /* compute bound */
    U_cb = U_curr
    if (U_mb = U_max) then
       /* Have only seen compute bound so far */
       U_curr = Increase(U_cb)
    else
       /* Balanced solution is between earlier size and this */
       U_curr = SelectBetween(U_cb, U_mb)
 /* Check if no more points to search */
 if (U_curr = U_cb) ok = True
end
return U_curr
```

Figure 2: Algorithm for Design Space Exploration.

6. EXPERIMENTAL RESULTS

This section presents experimental results that characterize the effectiveness of the previously described design space exploration algorithm for a set of kernel applications. We describe the applications, the experimental methodology and discuss the results.

6.1 Application Kernels

We demonstrate our design exploration algorithm on five multimedia kernels, namely:

- Finite Impulse Response (FIR) filter, integer multiply-accumulate over 32 consecutive elements of a 64 element array.
- Matrix Multiply (MM), integer dense matrix multiplication of a 32-by-16 matrix by a 16-by-4 matrix.
- String Pattern Matching (PAT), character matching operator of a string of length 16 over an input string of length 64.

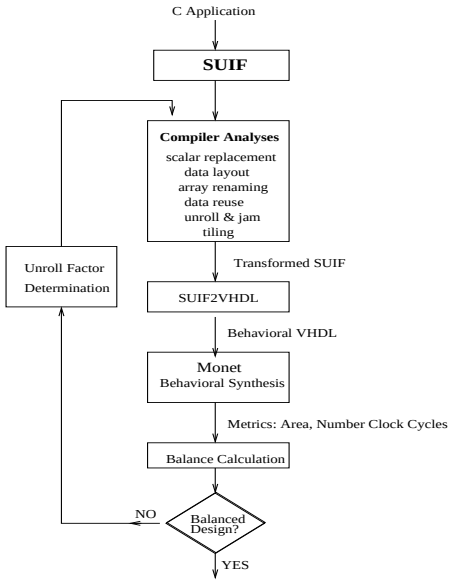


Figure 3: Compilation and Synthesis Flow.

- Jacobi Iteration (JAC), 4−point stencil averaging computation over the elements of an array.
- Sobel (SOBEL) Edge Detection (see *e.g.*, SOBEL [22]), 3-by-3 window Laplacian operator over an integer image.

Each application is written as a standard C program where the computation is a single loop nest. There are no pragmas, annotations or language extensions describing the hardware implementation.

6.2 Methodology

We applied our prototype compilation and synthesis system to analyze and determine the best unrolling factor for a balanced hardware implementation. Figure 3 depicts the design flow used for these experiments. First, the code is compiled into the SUIF format along with the application of standard compiler optimizations. Next, our design space exploration algorithm iteratively determines which loops in the loop nest should be unrolled and by how much. To make this determination the compiler starts with a given unrolling factor and applies a sequence of transformations as described in Sections 4 and 5. Next, the compiler translates the SUIF code resulting from the application of the selected set of transformations to behavioral VHDL using a tool called SUIF2VHDL. The compiler next invokes the Mentor Graphics' `Monet`™ behavioral synthesis tool to obtain space and performance estimates for the implementation of the behavioral specification. In this process, the compiler currently fixes the clock period to be $40ns$. The `Monet`™ synthesis estimation yields the amount of area used by the implementation and the number of clock cycles required to execute to completion the computation in the behavioral specification. Given this data, the compiler next computes the *balance* metric.

This system is fully automated. The implementation of the compiler passes specific to this experiment, namely data reuse analysis, scalar replacement, unroll&jam, loop peeling, and customized data layout, constitutes approximately $14,500$ lines of C++ source code. The algorithm executed in less than 5 minutes for each application, but to fully synthesize each design would require an additional couple of hours.

6.3 Results

In this section, we present results for the five previously described kernels in Figures 4 through 10. The graphs show a large number of points in the design space, substantially more than are searched by our algorithm, to highlight the relationship between unroll factors and metrics of interest. The first set of results in Figures 4 through 7 plots balance, execution cycles and design area in the target FPGA as a function of unroll factors for the inner and outer loops of FIR and MM. Although MM is a 3-deep loop nest, we only consider unroll factors for the two outermost loops, since through loop-invariant code motion the compiler has eliminated all memory accesses in the innermost loop. The graphs in the first two columns have as their x-axis unroll factors for the inner loop, and each curve represents a specific unroll factor for the outer loop.

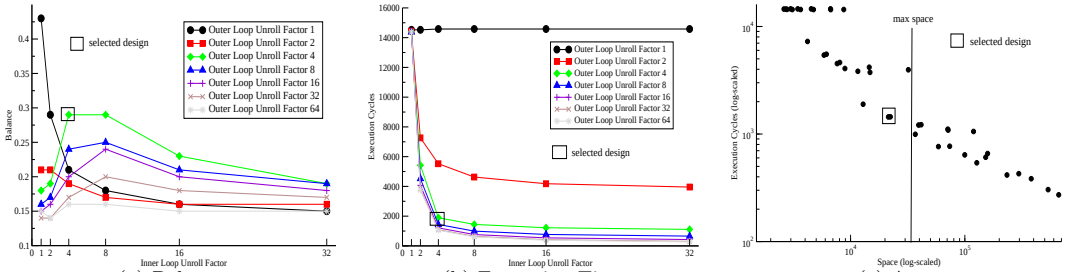
For FIR and MM, we have plotted the results for pipelined and non-pipelined memory accesses to observe the impact of memory access costs on the balance metric and consequently in the selected designs. In all plots, a squared box indicates the design selected by our search algorithm. For pipelined memory accesses, we assume a read and write latency of 1 cycle. For non-pipelined memory accesses, we assume a read latency of 7 cycles and a write latency of 3 cycles, which are the latencies for the Annapolis `WildStar`™ [13] board, a target platform for this work. In practice, memory latency is somewhere in between these two as some but not all memory accesses can be fully pipelined. In all results we are assuming 4 memories, which is the number of external memories that are connected to each of the FPGAs in the Annapolis `WildStar`™ board.

In these plots, a design is *balanced* for an unrolling factor when the y-axis value is 1.0. Data points above the y-axis value of 1.0 indicate *compute-bound* designs whereas points with the y-axis value below 1.0 indicate *memory-bound* designs. A *compute-bound* design suggests that more resources should be devoted to speeding up the computation component of the design, typically by unrolling and consuming more resources for computation. A *memory-bound* design suggests that less resources should be devoted to computation as the functional units that implement the computation are idle waiting for data. The design area graphs represent space consumed (using a log scale) on the target Xilinx Virtex 1000 FPGAs for each of the unrolling factors. A vertical line indicates the maximum device capacity. All designs to the right side of this line are therefore unrealizable.

With pipelined memory accesses, there is a trend towards compute-bound designs due to low memory latency. Without pipelining, memory latency becomes more of a bottleneck leading, in the case of FIR, to designs that are always memory bound, while the non-pipelined MM exhibits compute-bound and balanced designs.

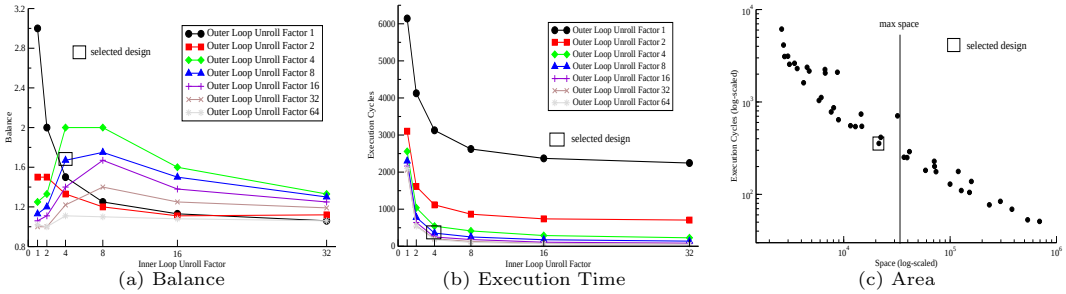
The second set of results, in Figures 8 through 10, show performance of the remaining three applications, JAC, PAT and SOBEL. In these figures, we present, as before, balance, cycles and area as a function of unroll factors, but only for pipelined memory accesses due to space limitations.

We make several observations about the full results. First, we see that *Balance* follows the monotonicity properties described in Observation 3, increasing until it reaches a sat-



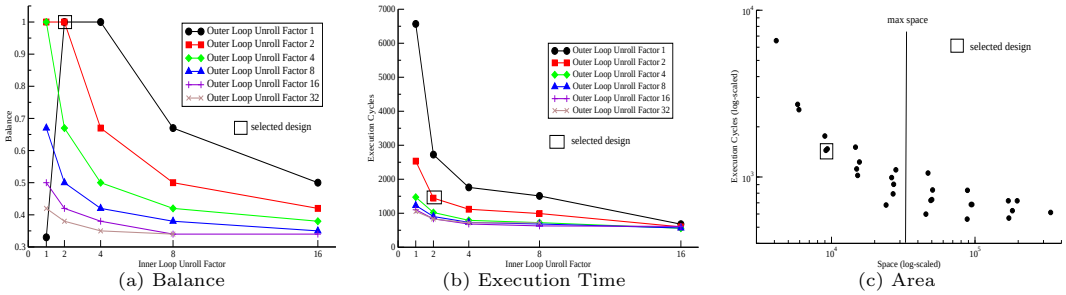
(a) Balance (b) Execution Time (c) Area

Figure 4: Balance, Execution Time and Area for Non-pipelined FIR.



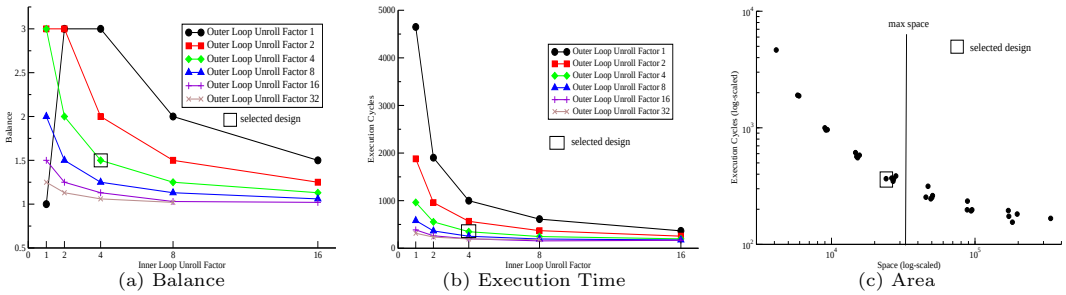
(a) Balance (b) Execution Time (c) Area

Figure 5: Balance, Execution Cycles and Area for Pipelined FIR.



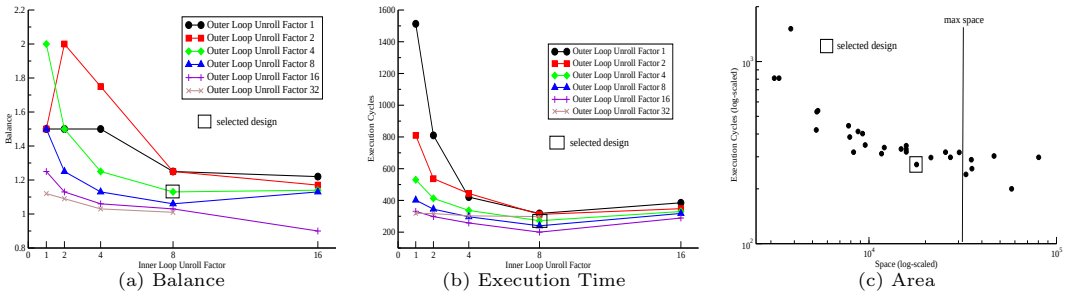
(a) Balance (b) Execution Time (c) Area

Figure 6: Balance, Execution Cycles and Area for Non-pipelined MM.



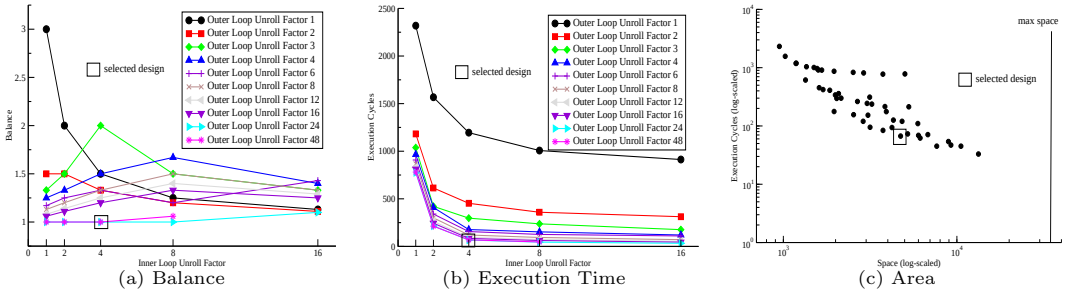
(a) Balance (b) Execution Time (c) Area

Figure 7: Balance, Execution Cycles and Area for Pipelined MM.



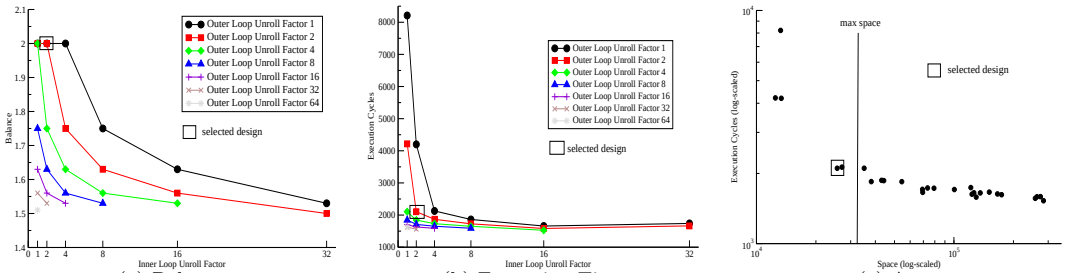
(a) Balance (b) Execution Time (c) Area

Figure 8: Balance, Execution Time and Area for Pipelined JAC.



(a) Balance (b) Execution Time (c) Area

Figure 9: Balance, Execution Cycles and Area for Pipelined PAT.



(a) Balance (b) Execution Time (c) Area

Figure 10: Balance, Execution Time and Area for Pipelined SOBEL.

uration point, and then decreasing. The execution time is also monotonically nonincreasing, related to Observation 2. In all programs, our algorithm selects a design that is close to best in terms of performance, but uses relatively small unroll factors. Among the designs with comparable performance, in all cases our algorithm selected the design that consumes the smallest amount of space. As a result, we have shown that our approach meets the optimization goals set forth in Section 3. In most cases, the most balanced design is selected by the algorithm. When a less balanced design is selected, it is either because the more balanced design is before a saturation point (as for non-pipelined FIR), or is too large to fit on the FPGA (as for pipelined MM).

Table 2 presents the speedup results of the selected design for each kernel as compared to the baseline, for both pipelined and non-pipelined designs. The baseline is the loop nest with no unrolling (unroll factor is 1 for all loops) but including all other applicable code transformations as described in Section 4.

Although in these graphs we present a very large number of design points, the algorithm searches only a tiny fraction of those displayed. Instead, the algorithm uses the pruning heuristics based on the saturation point and balance, as described in section 5. This reveals the effectiveness of the algorithm as it finds the best design point having only explored a small fraction, only 0.3% of the design space consisting of all possible unroll factors for each loop. For larger design spaces, we expect the number of points searched relative to the size to be even smaller.

Program	Non-Pipelined	Pipelined
FIR	7.67	17.26
MM	4.55	13.36
JAC	3.87	5.56
PAT	7.53	34.61
SOBEL	4.01	3.90

Table 2: Speedup on a single FPGA.

6.4 Accuracy of Estimates

To speed up design space exploration, our approach relies on estimates from behavioral synthesis rather than going through the lengthy process of fully synthesizing the design, which can be anywhere from 10 to 10,000 times slower for this set of designs. To determine the gap between the be-

havioral synthesis estimates and fully synthesized designs, we ran logic synthesis and place-and-route to derive implementations for a few selected design points in the design space for each of the applications. We synthesized the baseline design, the selected designs for both pipelined and non-pipelined versions, and a few additional unroll factors beyond the selected design.

In all cases, the number of clock cycles remains the same from behavioral synthesis to implemented design. However, the target clock rate can degrade for larger unroll factors due to increased routing complexity. Similarly, space can also increase, slightly more than linearly with the unroll factors. These factors, while present in the output of logic synthesis and place-and-route, were negligible for most of the designs selected by our algorithm. Clock rates degraded by less than 10% for almost all the selected designs as compared with the baseline, and the speedups in terms of reduction in clock cycles more than made up for this. In the case of FIR with pipelining, the clock degraded by 30%, but it met the target clock of $40ns$, and because the speedup was $17X$, the performance improvement was still significant. The space increases were sublinear as compared to the unroll factors, but tended to be more space constrained for large designs than suggested by the output of behavioral synthesis.

The very large designs that appear to have the highest performance according to behavioral synthesis estimates show much more significant degradations in clock and increases in space. In these cases, performance would be worse than designs with smaller unroll factors. Our approach does not suffer from this potential problem because we favor small unroll factors, and only increase the unrolling factor when there is a significant reduction in execution cycles due to memory parallelism or instruction-level parallelism.

For this set of applications, these estimation discrepancies, while not negligible, never influenced the selected design. While this accuracy issue is clearly orthogonal to the design space algorithm described in this paper, we believe that estimation tools will improve their ability to deliver accurate estimates given the growing pressures for accuracy in simulation for increasingly larger designs.

7. RELATED WORK

In this section we discuss related work in the areas of automatic synthesis of hardware circuits from high-level language constructs and design space exploration using high-level loop transformations.

7.1 Synthesizing High-Level Constructs

The gap between hardware description languages such as VHDL or Verilog and applications in high-level imperative programming languages prompted researchers to develop hardware-oriented high-level languages. These new languages would allow programmers to migrate to configurable architectures without having to learn a radically new programming paradigm while retaining some level of control about the hardware mapping and synthesis process.

One of the first efforts in this direction was the Handel-C[21] parallel programming language. Handel-C is heavily influenced by the OCCAM CSP-like parallel language but has a C-like syntax. The mapping from Handel-C to hardware is compositional where constructs, such as `for` and `while` loops, are directly mapped to predefined template hardware structures [20].

Other researchers have developed approaches to mapping applications to their own reconfigurable architectures that are not FPGAs. These efforts, *e.g.,* the RaPiD [7] reconfigurable architecture and the PipeRench [12], have developed an explicitly parallel programming language and/or developed a compilation and synthesis flow tailored to the features of their architecture.

The Cameron research project is a system that compiles programs written in a single-assignment subset of C called SA-C into dataflow graphs and then synthesizable VHDL [23]. The SA-C language includes *reduction* and *windowing* operators for two-dimensional array variables which can be combined with *doall* constructs to explicitly expose parallel operations in the computation. Like in our approach, the SA-C compiler includes loop-level transformations such as loop unrolling and tiling, particularly when windowing operators are present in a loop. However, the application of these transformations is controlled by *pragmas*, and is not automatic. Cameron's estimation approach builds on their own internal data-flow representation using curve fitting techniques [17].

Several other researchers have developed tools that map computations expressed in a sequential imperative programming language such as C to reconfigurable custom computing architectures. Weinhardt [24] describes a set of program transformations for the pipelined execution of loops with loop-carried dependences onto custom machines using a pipeline control unit and an approach similar to ours. He also recognizes the benefit of data reuse but does not present a compiler algorithm.

The two projects most closely related to ours, the Nimble compiler [19] and work by Babb et. al. [2], map applications in C to FPGAs, but do not perform design space exploration. They also do not rely on behavioral synthesis, but in fact the compiler replaces most of the function of synthesis tools.

7.2 Design Space Exploration

In this discussion, we focus only on related work that has attempted to use loop transformations to explore a wide design space. Other work has addressed more general issues such as finding a suitable architecture (either reconfigurable or not) for a particular set of applications [1].

In the context of behavioral VHDL [16] current tools such as `Monet`™ [14] allow the programmer to control the application of loop unrolling for loops with constant bounds. The programmer must first specify an application behavioral VHDL, linearize all multi-dimensional arrays, and then select the order in which the loops must execute. Next the programmer must manually determine the exact unroll factor for each of the loops and determine how the unrolling is going to affect the required bandwidth and the computation. Given the effort and interaction between the transformations and the data layout options available this approach to design space exploration is extremely awkward and error-prone.

Other researchers have also recognized the value of exploiting loop-level transformations in the mapping of regular loop computations to FPGA-based architectures. Derrien/Rajopadhye [8] describe a tiling strategy for doubly nested loops. They model performance analytically and select a tile size that minimizes the iteration's execution time.

7.3 Discussion

The research presented in this paper differs from the efforts mentioned above in several respects. First the focus of

this research is in developing an algorithm that can explore a wide number of design points, rather than selecting a single implementation. Second, the proposed algorithm takes as input a sequential application description and does not require the programmer to control the compiler's transformations. Third, the proposed algorithm uses high-level compiler analysis and estimation techniques to guide the application of the transformations as well as evaluate the various design points. Our algorithm supports multi-dimensional array variables absent in previous analyses for the mapping of loop computations to FPGAs. Finally, we use a commercially available behavioral synthesis tool to complement the parallelizing compiler techniques rather than creating an architecture-specific synthesis flow that partially replicates the functionality of existing commercial tools. Behavioral synthesis allows the design space exploration to extract more accurate performance metrics (time and area used) rather than relying on a compiler-derived performance model. Our approach greatly expands the capability of behavioral synthesis tools through more precise program analysis.

8. CONCLUSION

We have described a compiler algorithm that balances computation and memory access rates to guide hardware design space exploration for FPGA-based systems. The experimental results for five multimedia kernels reveal the algorithm quickly (in less than five minutes, searching less than 0.3% of the search space) derives a design that closely matches the best performance within the design space and is smaller than other designs with comparable performance.

This work addresses the growing need for raising the level of abstraction in hardware design to simplify the design process. Through combining strengths of parallelizing compiler and behavioral synthesis, our system automatically performs transformations typically applied manually by hardware designers, and rapidly explores a very large design space. As technology increases the complexity of devices, consequently designs will become more complex, and furthering automation of the design process will become crucial.

Acknowledgements. This research has been supported by DARPA contract # F30602-98-2-0113. The authors wish to thank contributors to the DEFACTO project, upon which this work is based, in particular Joonseok Park, Heidi Ziegler, Yoon-Ju Lee, and Brian Richards.

9. REFERENCES

[1] S. Abraham, B. Rau, R. Schreiber, G. Snider, and M. Schlansker. Efficient design space exploration in PICO. Tech. report, HP Labs, 1999.

[2] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua and S. Amarasinghe. Parallelizing Applications into Silicon. In *Proc. of the IEEE Symp. on FPGA for Custom Computing Machines (FCCM'99)*, 1999.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A compiler-managed memory system for raw machines. In *Proc. of the 26th Intl. Symp. on Computer Architecture (ISCA'99)*, 1999.

[4] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proc. of the ACM Conference on Program Language Design and Implementation (PLDI'90)*, pages 53–65, 1990.

[5] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.

[6] Altera Corp. APEX II programmable logic device data sheets. 2001.

[7] D. Cronquist, P. Franklin, and C. Ebeling. Specifying and compiling applications for RaPiD. In *Proc. of the IEEE Symp. on FPGA for Custom Computing Machines (FCCM'98)*, pages 116–125, 1998.

[8] S. Derrien and S. Rajopadhye. Loop tiling for reconfigurable accelerators. In *Proc. of the Eleventh Intl. Symp. on Field Programmable Logic (FPL'2001)*, 2001.

[9] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Bridging the gap between compilation and synthesis in the DEFACTO system. In *Proc. of the Forteenth Workshop on Languages and Compilers for Parallel Computing (LCPC'2001)*, August 2001. To be published as Lecture Notes in Computer Science.

[10] J. P. Elliott. *UnderStanding Behavioral Synthesis: A Practical Guide to High-Level Design*. 1999.

[11] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective. In *Proc. of the ACM Symp. on Field Programmable Gate Arrays (FPGA'2002)*, 2001.

[12] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Intl. Symp. on Computer Architecture (ISCA'99)*, 1999.

[13] Annapolis MicroSystems `WildStar`^{tm} manual, 4.0. 1999.

[14] Mentor Graphics `Monet`^{TM} user's manual (release r42). 1999.

[15] XILINX Virtex-II 1.5V FPGA data sheet. ds031(v1.7). 2001.

[16] D. Knapp. *Behavioral Synthesis*. Prentice-Hall, 1996.

[17] D. Kulkarni, W. Najjar, R. Rinker, and F. Kurdahi. Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems. In *Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines (FCCM'2002)*, 2002.

[18] M. Leong, O. Cheung, K. Tsoi, and P. Leong. A bit-serial implementation of the international data encryption algorithm IDEA. In *Proc. of the IEEE Symp. on FPGA for Custom Computing Machines (FCCM'98)*, pages 122–131, 1998.

[19] Y. Li, T. Callahan, E. Darnell, R.E. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proc. of the Design Automation Conference (DAC '00)*, June, 2000.

[20] W. Luk, D. Ferguson, and I. Page. *Structured hardware compilation of parallel programs*. Abingdon EE&CS Books, 1994.

[21] I. Page and W. Luk. Compiling OCCAM into FPGAs. In *Proc. of the First Intl. Symp. on Field Programmable Logic (FPL'91)*, 1991.

[22] J. Proakis and D. G. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications*. Prentice-Hall, 1995.

[23] R. Rinker, M. Carter, A. Patel, M.Chawathe, C. Ross, J. Hammes, W. Najjar, and W. Bohm. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Trans. on VLSI Systems*, 9(1):130–139, 2001.

[24] M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Proc. of the 1997 Reconfigurable Architectures Workshop RAW'97*. Springer-Verlag, 1997.

[25] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Addison-Wesley, 1996.

[26] H. Ziegler, B. So, M. Hall, and P. Diniz. Coarse-Grain Pipelining for Multiple FPGA Architectures. In *Proc. of the IEEE Symp. on FPGA for Custom Computing Machines (FCCM'02)*, 2002.