

Behavioral Level Guidance Using Property-Based Design Characterization

by

Lisa Marie Guerra

B.S. (Stanford University) 1990

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering-Electrical Engineering

and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Jan Rabaey, Chair

Professor A. Richard Newton

Professor Paul Wright

Fall 1996

The dissertation of Lisa Marie Guerra is approved:

Chair

Date

Date

Date

University of California, Berkeley

Fall 1996

Behavioral Level Guidance Using Property-Based Design
Characterization

Copyright 1996

by

Lisa Marie Guerra

Abstract

Behavioral-Level Guidance Using Property-Based Design Characterization

by

Lisa Marie Guerra

Doctor of Philosophy in Engineering —
Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Jan M. Rabaey, Chair

The growing importance of optimization, short time to market windows, and exponentially growing design complexity are just a few of the factors shaping the state-of-the-art synthesis process. In particular, optimization at the early stages of design is crucial — at the system and behavioral levels, orders of magnitude performance improvement in key design metrics such as throughput, power, and area can be attained. This requires, however, strategic and coordinated application of design techniques best suited for a target design. The problem, however, is the number of options currently available is overwhelming, and as a result, design exploration is often conducted in a qualitative, ad-hoc manner.

To address these challenges, this thesis introduces a new design methodology for guiding the exploration process to quickly find effective sequences of design optimizations. The building blocks of the methodology are quantitative design characterization and a

library of characterized optimization techniques. Design characterization is done using a set of techniques to automatically extract the “essence” of a design description. The library of characterized optimization techniques encapsulates knowledge about the effectiveness, scope, and interdependencies of various optimizations. These two building blocks enable analysis of optimization alternatives, and have been encapsulated in an interactive guidance environment.

The guidance environment suggests and ranks potential optimizations, both in terms of immediate and longer-term impact. It also provides evaluations of the design and of the likely effects each optimization will have on performance. Using the provided guidance, designers can make decisions in a more informed manner and can explore the space more effectively, thus resulting in shorter design time and more highly optimized designs.

A core contribution of this thesis is the design characterization. The essence of the design is captured using property metrics that are shown to be related to the quality of algorithm-architecture mappings. The following properties and their quantifications are presented: size, topology, timing, concurrency, uniformity, locality, and regularity. As well as being a key component of the guidance methodology, this work demonstrates the effectiveness of using property metrics in algorithm selection, performance estimation, and architectural synthesis.

Jan M. Rabaey

Committee Chairman

Table of Contents

Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Contribution of Thesis	3
1.3 Organization of Thesis	4
Chapter 2 Background and Motivation.....	6
2.1 Scope	6
2.1.1 Applications	6
2.1.2 Flowgraph Representation	7
2.1.3 Architecture Model	8
2.2 Mapping from Algorithm to Architecture: Architectural Synthesis	9
2.3 Performance Estimation	11
2.4 Behavioral-Level Optimization and Exploration	14
2.5 Summary	19
Chapter 3 Property-Based Design Characterization	20
3.1 Overview	20
3.2 Size	24
3.3 Topology	25
3.4 Timing	27
3.5 Concurrency and Uniformity	29
3.5.1 Concurrency of Operations	30
3.5.2 Concurrency of Accesses to Registers	32
3.5.3 Concurrency of Data Transfers	34
3.5.4 Concurrency of Variables	35
3.5.5 Uniformity	41
3.6 Locality	42
3.6.1 Temporal Locality	42
3.6.2 Spatial Locality	43
3.7 Regularity	48
3.7.1 Quantification of Regularity	49
3.7.2 Complexity	51
3.8 Other Property Metrics	58
3.9 Summary	60

Chapter 4 Applications of Design Characterization	62
4.1 Case Study: Avenhaus Filter Structures	62
4.2 Algorithm Selection	69
4.3 Performance Estimation	69
4.3.1 Execution Unit Power Estimation: Size	69
4.3.2 Execution Unit Area Modeling: Operator and Register Access Concurrency	70
4.3.3 Register Area Modeling: Variable Concurrency	72
4.3.4 Early Controller Area Estimation	75
4.3.5 Relationship Between Regularity and Interconnect Structure	75
4.4 Architectural Synthesis	78
4.4.1 Assignment for Interconnect Reduction: Exploiting Regularity	78
4.5 Summary	81
Chapter 5 Guided Design Exploration	82
5.1 Related Works	82
5.2 Global Overview	84
5.3 Methodology Features and Benefits	86
5.4 Design Guidance System	89
5.5 Optimization Characterization	91
5.6 Ranking	95
5.7 Interface and Implementation	97
5.8 Design Example	98
5.9 Summary	107
Chapter 6 Conclusion	108
6.1 Directions for Future Research	108
6.1.1 Improvements to the Guided Optimization Environment	108
6.1.2 Design Characterization in Other Domains	110
6.1.3 New Applications of Design Characterization	110
6.2 Summary	110
Chapter 7 Bibliography	113
APPENDIX A: Architecture Model — Rules and Constraints	121
APPENDIX B: Summary of Property Metrics	123
APPENDIX C: Library of Optimization Characterizations	126

Acknowledgments

Words cannot sufficiently express my gratitude for the guidance, instruction, support, and friendship that I have received over the years. Without it, this thesis would not be where it is today. And nor would I be where I am today.

I must first thank my research adviser, Jan Rabaey, who has been a great force behind this research. Jan, you are a great teacher, adviser, and role model whom I greatly admire. Thank you for your patience with me when I was dragging, and for your encouragement while I was rolling. This research has also been shaped heavily by our collaboration with Miodrag Potkonjak whom I have had the pleasure of working together with ever since my first year at Berkeley. Miodrag you are an unending source of ideas and knowledge and an excellent teacher and motivator. I will never forget the hour, day, and week-long blocks that we spent together learning (well, primarily me learning from you), brainstorming, and working. Miodrag, you have been a technical inspiration. And you have become a very great friend whom I know will continue to challenge me not only in research, but more importantly in life.

I would also like to thank Professors Edward Lee and John Rice for providing guidance as members of my qualifying committee, Professor Paul Wright for serving on my dissertation committee, and Professor Richard Newton for serving on both. Thank you also, Professor Bob Brodersen, for your leadership and vision in the Infopad project. Regarding funding, I am grateful to the Office of Naval Research and to AT&T for providing me with graduate fellowships, thus allowing me to more freely pursue the research that most interested me.

There are so many co-workers and members of the BJ research group and EE department that have been an important part of my graduate career. Thanks to Phu Hoang for his early help in understanding the Hyper system and to Dev Chen for his help in the Paddi project. Thanks to Sean Huang for imparting his expertise in Hyper and in transformations. Renu Mehra, it has been a pleasure sharing an office and working with you during these last couple years. Thanks also to Miguel Corazao and Marwan Khalaf, with whom I have been fortunate to have had the opportunity to work with on template matching. Anantha Chandrakasan, thank you for your career advice, and for all those lunches on the northside. Arthur Abnous, thank you for your feedback on my research, and for being a good friend and listener. Jeff Gilbert, what would I have done without all the coffee? And also without your great edits on this thesis. John Reekie, your feedback on this thesis and on my papers have been excellent. You haven't quite converted me on some of your language style issues, but definitely on others. Ole Bentz, you have been a great office-mate, and a great model of focus, diligence and hard work. Naji Ghazal, thanks for your feedback on this work, and for bringing added personality to the office. Thanks to Sam Sheng for his music tips, restaurant tips, and oh yeah, hard-core technical tips. Kevin Zimmerman, I appreciate the cheerfulness with which you always helped us in fixing computing resources.

To David Lidsky, I don't know where to begin. You know that I could go on for days trying to express my appreciation for all that you have done. You have had a strong impact on the fruition of this research. You are an invaluable colleague. And, you are most definitely an invaluable friend and inspiration. Sovarong Leang, you have been a true friend all along. All the advice and guidance that you provided for me at Berkeley in the classroom, and in research I thank you for. All the years together dancing - the classes, practices, demos, and competitions - I will never forget.

There are also a lot of people that have helped make my graduate years a lot more enjoyable and who more importantly have been extremely supportive. Just a few include Keith Onodera, Arnold Feldman, Todd Weigandt, Marlene Wan, Luben Stoilov, Tom Burd, and Kevin Stone. I also want to acknowledge my friends who were constantly checking up on me to make sure I'd graduate some day: Konstantin Guericke, Lisa Martin, Seth Sakamoto, Liz Matsumoto, Eric Tomacruz, and Shelley Teruya, thank you all. Lisa Buckman, you have been a great room-mate and confidante. Nada Aleksic, I appreciate the generosity with which you always welcomed me into your home, and also for helping me survive some of Miodrag's questioning. Finally, Chris Rudell, your hard work and perseverance has been an inspiration to me. Your jokes and stories have livened up the office, and your advice, encouragement, and friendship, I cannot thank you enough for.

I would like to express my deepest appreciation for my family who has always been caring and has always had faith in me. Grandma Kawamoto, I'm finally done! Jenica, Kelsi, and Tony thanks for all the letters and phone calls. Mom and dad, you never told me what to do, you never pushed me to be better. It was your unconditional love that inspired me to grow and work hard. I would like to dedicate this thesis to you.

Introduction

1

1.1 Motivation

The development of new CAD tools and methodologies has not been keeping pace with the increase in complexity of VLSI designs. Figure 1, compiled by the Semiconductor Industry Association [SIA95], illustrates the widening gap between design complexity and designer productivity. In 1996, complexity is on the order of 10 million logic transistors per chip, while designer productivity is a thousand logic transistors per month. With complexity increasing at a projected 58% per year compounded growth rate and productivity increasing at a 21% growth rate, by the year 2010, designs will be close to 1 billion transistors while productivity will be just over 10 K transistors.

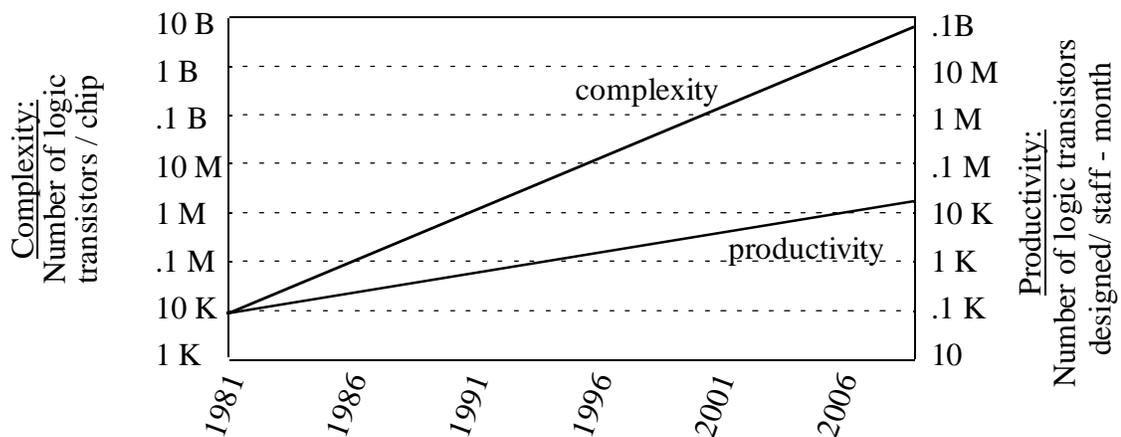


Fig. 1. Complexity of designs versus productivity supported by CAD tools [SIA95].

To aid in narrowing this complexity-productivity gap, tools and methodologies to better manage design complexity and thus increase designer productivity are needed. In particular, methodologies and tools to aid in more effective design at *high* levels of abstraction — such as at the behavioral level — have become a necessity. As will be discussed in the following two subsections, design aids for performing *guided* high-level design and optimization will be a key factor in increasing productivity.

Effectiveness of High-Level Decisions

Case studies indicate that high-level decisions regarding selection and optimization of algorithms and architectures can improve design performance metrics by many orders of magnitude [Wuy94, Pot93, Lid94]. In the video vector quantizer of [Lid94], for example, behavioral and architectural-level changes resulted in power reductions by factors of approximately 30 and 17, respectively. Keutzer and Vanbekbergen's [Keu94] survey similarly reports order-of-magnitude power improvements by existing digital CAD optimizations at the high levels, while logic and layout level techniques result in improvements of just 15 - 20%. By addressing high-level trade-offs to obtain these large improvements, productivity can be greatly increased since design objectives will be met more quickly.

Need for Additional High-Level Design Aids

Automated behavioral-level synthesis [Wal91, Gaj92, Rab91a], optimization, and performance estimation [Kur89, Rab91b, Jai92] are important in reducing high-level design and exploration time. However, they are not sufficient. At these aforementioned high levels of abstraction, the degrees of design freedom are often so great as to make full analysis of design trade-offs impractical. Comprehensive analysis of designs is often prohibitively time-consuming, as is developing a sufficient understanding of the various synthesis and optimization techniques. As a result, designers often search the design space in an ad-hoc manner, repeatedly applying various optimization and synthesis techniques, then evaluat-

ing their effect with estimators. Decisions are made subjectively and prematurely, based on partial analysis as well as factors such as convenience and familiarity. Consequently, much time may be squandered trying to tweak the design at lower-levels of abstraction.

1.2 Contribution of Thesis

To address these challenges (large design space, short time to market windows) design aids that support the decision making process, without neglecting existing tools, can greatly improve design productivity and quality. This thesis proposes techniques for integrating design assistance, from here on called “design guidance,” into the design exploration process. In addition, a means of design characterization by extracting key property metrics of a design, as well as the utilization of these metrics, will be presented.

Behavioral-Level Design Guidance

Design guidance provides the designer with feedback to facilitate effective decision making during the design process. A methodology and environment which provides interactive design guidance for optimization, and also encapsulates existing synthesis tools and estimators, is proposed.

Two core mechanisms behind the approach are design characterization and optimization characterization. For the latter, a database of encapsulated knowledge about the various optimizations is used. The environment suggests and ranks potential optimizations, taking into consideration not only immediate effects, but longer-term effects due to successive optimizations. It provides evaluations of the design and of the likely effects each optimization will have on metrics such as power, cost, and performance. Using this form of guidance, designers maintain a more global view of the exploration space, can make decisions in a more quantitative and informed manner, and thus can more easily and

quickly discover effective trajectories of optimizations. Consequently, higher quality designs and shorter design times can result.

Design Characterization

A key step in realizing the performance potential attainable using high-level design and optimization involves developing a better understanding of the correspondence between application algorithms and various architectures. A core contribution of this work is the identification of the design characteristics that are most directly related to the quality of algorithm-architecture mappings. This set of characteristics, or properties, can essentially be thought to capture the design’s “essence.” In this thesis, the following properties and their quantifications are presented: size, topology, timing, concurrency, uniformity, locality, and regularity.

Information such as design characterization, that captures the relationships between algorithms and architectures can be used in a number of different ways. As well as being a key component of the guidance system, this work demonstrates their effectiveness in *algorithm selection*, *performance estimation*, and *architectural synthesis*.

1.3 Organization of Thesis

This thesis is organized into six chapters. This chapter has provided a motivation for design exploration at the high levels of design abstraction, and for methodologies and tools to aid in the exploration. Chapter 2 defines the scope of applications and architectures that are assumed for this thesis. It also presents background material on high-level design and optimization.

Design characterization has been a major focus of this research; it will be presented in Chapter 3. Subsequent chapters present applications which are based on the characteriza-

tions. Chapter 4 demonstrates the use of properties in algorithm selection, performance estimation, and architectural synthesis. Chapter 5 presents the property-based methodology and environment for integrating guidance into the design exploration process. Conclusions and directions for future work are presented in Chapter 6.

Background and Motivation

2

This chapter presents background information for this thesis, and in so doing also motivates some of the new problems that this thesis addresses. In Section 2.1, the scope of applications and implementations is described. Sections 2.2 through 2.4 provide background for the areas in which a property-based approach can be applied: architectural synthesis, estimation, and optimization. Section 2.2 presents the basic architecture synthesis steps to map an application to architecture. Section 2.3 describes performance estimation. In Section 2.4 the use of optimizations to improve designs is illustrated, as is the effect of combining optimizations.

2.1 Scope

In order to validate the ideas presented in this thesis, a specific application domain and its associated representations and tools was chosen. It is likely that much of this work can be positively applied, with modifications, to other domains. This section presents the domain that has been focused on.

2.1.1 Applications

The application domain is real-time (fixed-throughput) DSP and other numerically intensive applications. Typical examples include various filters (FIR, IIR, non-linear), transforms (FFT, DCT, convolution, wavelet, Hilbert), sorting, elementary functions, and

more complex systems (DFE, echo cancellation, dynamic programming, speech coding, etc.).

2.1.2 Flowgraph Representation

The user specifies an algorithm in either the Silage applicative language [Hil85] or a C++ subset [Wan94]. This description is parsed and compiled into a hierarchical dataflow graph representation. This representation is central to this work, as the design characterization is extracted directly from it. In the flowgraph, nodes represent operators, and directed edges indicate the dependencies between them [Rab91a, Hoa92 - Chap.4, Ver94]. Data edges represent the transfer of data in addition to the data dependencies; control edges express additional relations not already imposed by the data precedence relations. For example, a control edge could be used between array read and write accesses to assure either read-after-write or write-after-read execution. Strictly speaking, the edges within the flowgraph representation are “hyperedges” since they may fanout to several locations.

The algorithms operate on pseudo-infinite streams of input samples to produce pseudo-infinite streams of output samples. The state of the algorithm is represented in the flowgraph by sample delay operators which are initialized to a user-specified value. A sample delay operation delays the stream of data on its sole input port by one sample. Intuitively, one can think of sample delay operators as representing registers holding state, and the other operators as combinational logic.

The semantics of the graphs are most similar in function to the synchronous dataflow model of Lee [Lee87]. Under this model, operators consume at every input, and produce at every output, a fixed number of samples on every execution. Extension to handle dynamic dataflow graphs [Lee93] can also be done by using maximum and expected numbers of iterations for non-deterministic loops and using expected probabilities for branches.

Hierarchy within flowgraphs is used to represent loop and subroutine structures. For example, a “for loop” is represented by a hierarchy node annotated with the associated number of iterations, and a subgraph containing the loop body. Hierarchical representations are more compact than unrolled or flattened representations, and thus enable more efficient synthesis and optimization. In the handling and analysis of hierarchical descriptions, a single thread of control at the hierarchy level is assumed. As a result, while parallelism can be exploited in the scheduling and execution of operations at the leaf level, the hierarchical operations themselves are executed sequentially.

2.1.3 Architecture Model

The target implementations are semi-custom application-specific integrated circuits (ASICs). While there exists a wide range of ASIC architecture models, this thesis focuses on the model shown in Figure 2 and detailed in Appendix A. Under this architecture model, there is an unlimited amount of parallelism. Further, any number of operations

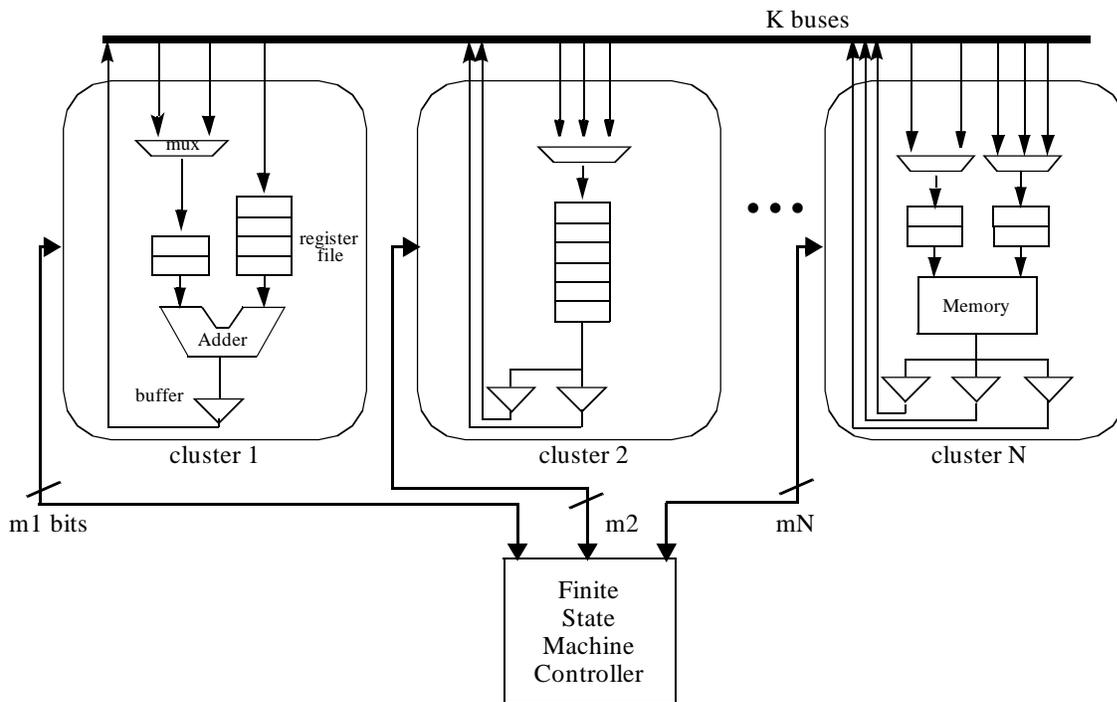


Fig. 2. Sample architecture instance.

from the flowgraph can be mapped to a single execution unit. Registers are clustered into register files, which are tied to the input ports of an execution unit block. Broadcasting is not allowed from the register files. An execution unit block is either a single execution unit from the library (e.g., an adder), or a set of chained units (e.g., a multiplier directly feeding an adder). During execution, operands are fetched from the registers by the execution unit block, operated on, and the results written back to register files. An execution unit block along with its associated multiplexors, register files, and buffers is considered a cluster. Definition of an architecture instance involves selecting the number and types of datapath clusters as well as their interconnection. Datapath cluster definition involves selecting the execution unit block, the number of registers in each register file, the multiplexor implementation, and the number of buffers.

2.2 Mapping from Algorithm to Architecture: Architectural Synthesis

Mapping involves deriving an instance of the specified architecture model to implement a given application. Mapping can be performed manually, or by architectural synthesis. Both forms of mapping have been used in the work described in this thesis. Manual mapping was performed during exploration of new ideas regarding improvements to the existing synthesis tools (Section 4.4). Synthesis, with the Hyper behavioral-level synthesis system [Rab91a], was used during the development of new estimators (Section 4.3) and for general experimentation.

The Hyper system, developed at the University of California at Berkeley, provides an automated path from a high-level language to an architectural netlist. The system targets datapath-intensive DSP applications such as those described in Section 2.1.1 and the architecture model described in Appendix A.

Application description, constraints, parameters, objective

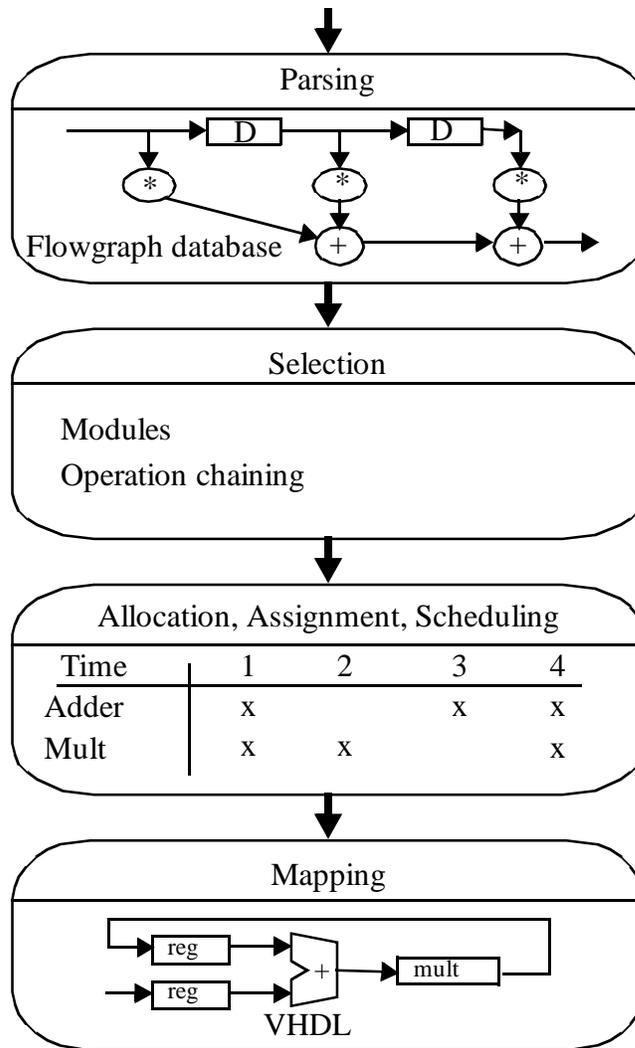


Fig. 3. Basic architectural synthesis steps.

Figure 3 presents an overview of basic architectural synthesis steps within the Hyper system. The user specifies the application, the performance constraints, the parameters, and an objective. A sample performance constraint is “required throughput”; sample parameters are the “clock period” and “supply voltage”; a sample objective is “area minimization.” The parser starts by converting the application description into the flowgraph representation. The flowgraph serves as the working structure on which synthesis and

optimization are performed and to which results are stored. Next, the module selector selects appropriate hardware elements from a library to implement each operation. The hardware library [Bur94] contains parameterized (e.g., in terms of the resource's bitwidth) performance models of the area, delay, and capacitance of each resource. This performance information is annotated onto the flowgraph for use in subsequent synthesis steps. The module selector also determines which operations should be chained [Cor93]. The allocation, assignment, and scheduling tools determine the number of resources to allocate and the assignment of each operation to a particular hardware unit and time slot [Pot89]. The resulting schedule is non-overlapped, in that it is derived for a single iteration of the computation, and is repeated for the subsequent iterations. The last step, hardware mapping [Ben93], generates a finite state machine to control the datapath, and emits a description of the architecture in VHDL [Lip89] or SDL [Bro92]. The SDL format is suitable for silicon compilation to layout using the Lager system [Bro92]. More details on the Hyper system and architectural synthesis in general can be found in [Rab91a, Wal91, Gaj92].

2.3 Performance Estimation

Rarely does the initial user-defined specification meet the design objectives. Rather than evaluating performance at the architectural level after hardware mapping, performance estimates can be run before proceeding to the allocation, assignment, and scheduling steps (Figure 4).

Common performance metrics include throughput, area, power, latency, testability, and fault tolerance. Our focus has been on the first three: throughput, area, and power dissipation. The high-level synthesis literature contains a variety of techniques for computing expected or lower bounds on these performance metrics for various architecture models

Application description, constraints, parameters, objective

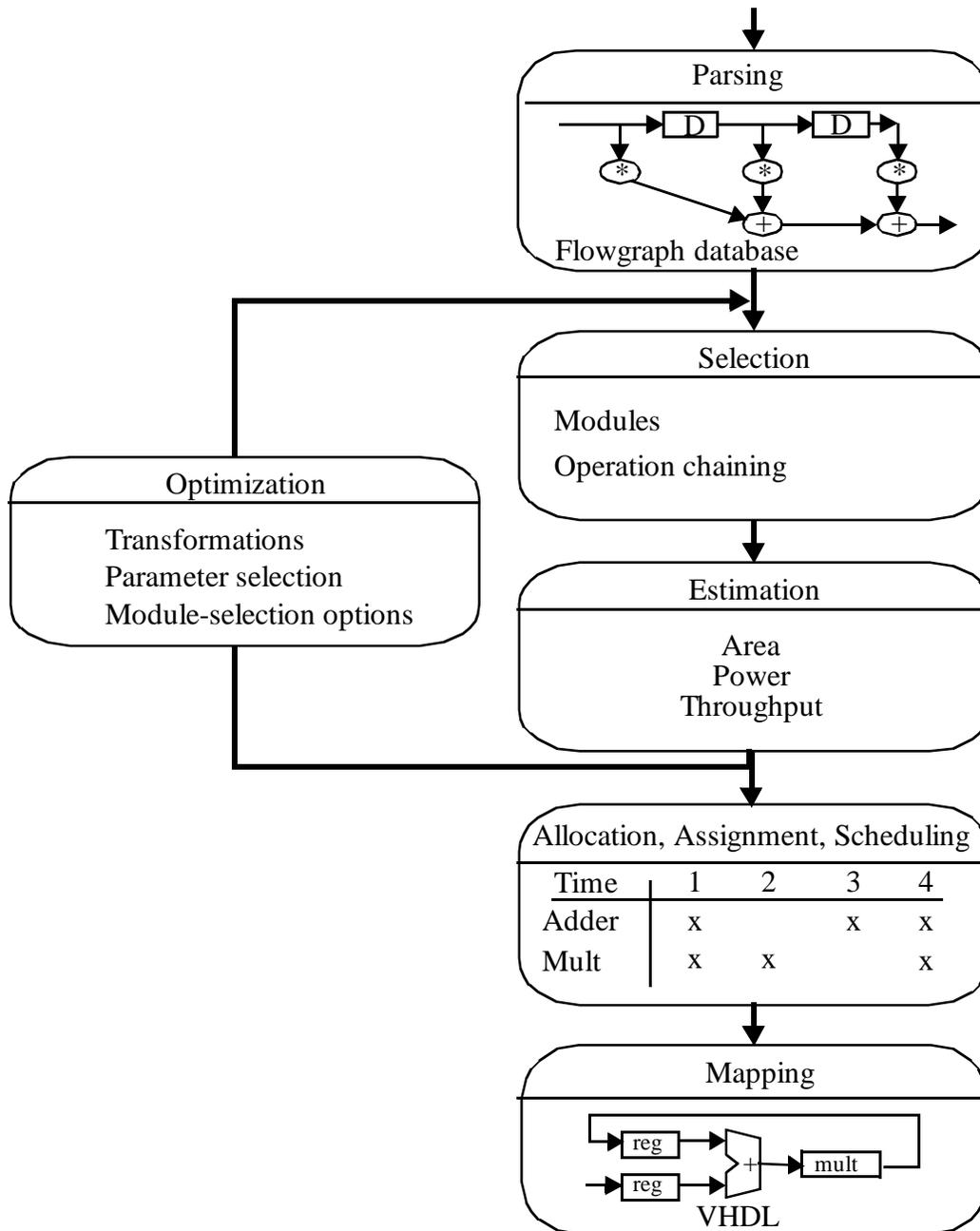


Fig. 4. Sample design flow of the basic synthesis steps, estimation, and optimization.

[Kur89, Rab91b, Jai92, Sha93, Cha95]. The following paragraphs provide an overview of the estimation techniques for the architecture model which is detailed in Appendix A.

Of the three metrics of interest, maximum throughput is the most straightforward to estimate. An implementation's *throughput* is the rate at which it accepts and processes data samples. The inverse of the throughput is the sample period, which is the time between the arrival of successive input samples. The computation's critical path determines the minimum sample period, and hence also the maximum throughput. It is assumed that all inputs and all previous states are available at the same time (at the start of each iteration). Under this assumption, the critical path is defined as the weighted length of the longest path from any primary input or state to any state or output. The overall register-to-register operator delays are determined through a combination of the models in the hardware library and the timing models of [Chu92]. The register-to-register operator delay takes into account not only the operator delays, but also the control decode delay, the register access delay, and the clock duty cycle.

Estimation of the chip's area is performed through a combination of techniques. Both empirical [Gue94] (Section 4.3) and deterministic min-bound [Rab91b] techniques are used for the estimation of execution units, registers, and interconnect counts. This is supplemented with additional empirical models to provide an estimate of total chip area, which takes into account the control and wiring area [Meh94].

Power dissipated in digital CMOS circuits is composed of dynamic, short-circuit, and leakage power. The latter two, however, are influenced mainly by the circuit design style used, and can be designed to be less than 15% of the total chip power [Vee84]. For analysis at the algorithm and architectural levels, therefore, power can be adequately described by its dynamic component:

$$Power = C_{eff} \cdot (V_{SW} \cdot V_{DD}) \cdot f$$

where f is the frequency of operation, V_{SW} is the switched voltage, V_{DD} is the supply voltage, and C_{eff} is the effective switched capacitance. C_{eff} is dependent on C , the physical capacitance being charged or discharged, and α , the activity factor: $C_{eff} = \alpha \cdot C$. Just as with area estimation, power estimation [Meh94] is done using a combination of techniques. Deterministic models are used for the power estimation of execution units and registers. Empirical models are used for control, interconnect, and clock power estimation.

2.4 Behavioral-Level Optimization and Exploration

After estimation, optimizations can be applied to improve the design's area, power, and throughput. An optimization may involve changing a parameter such as the clock frequency [Cor96] or voltage [Cha95, Rag95], performing chaining [Cor96], selecting new hardware resources from the hardware library [Jai90], or applying a transformation. Transformations involve changes in the structure of an algorithm without changing the input-output behavior. Many behavioral-level optimizations used in CAD have been adopted from compiler technology. An overview of compiler optimizations as well as an extensive bibliography can be found in [All75, Aho77, Rob87, Bac94]. Further, Parhi [Par95] presented a survey on numerous transformations for DSP systems.

Behavioral-level transformations include algebraic transformations (using the associative, distributive, and commutative identities), common sub-expression elimination, constant propagation, retiming, pipelining, loop transformations (loop jamming, loop unfolding, software pipelining), and replacement of constant multiplications with additions and shifts.

Multitudes of optimization techniques have been developed for applying optimizations. Some are restricted to optimization of a specific class of computation such as linear computations [Pot92, Sri95a]. Some focus on power optimization while others target area

or throughput. Furthermore, some concentrate on reduction of just a specific sub-metric; for example, different power optimization techniques have been used to reduce capacitance, to reduce activity, and to enable voltage scaling [Mus95, Rag95, Cha95, Cat94].

Optimization Examples

Figures 5 and 6 illustrate the effects of 2 common transformations: distributivity and retiming. In Figure 5a, assuming that each operation has a delay of one clock cycle and an

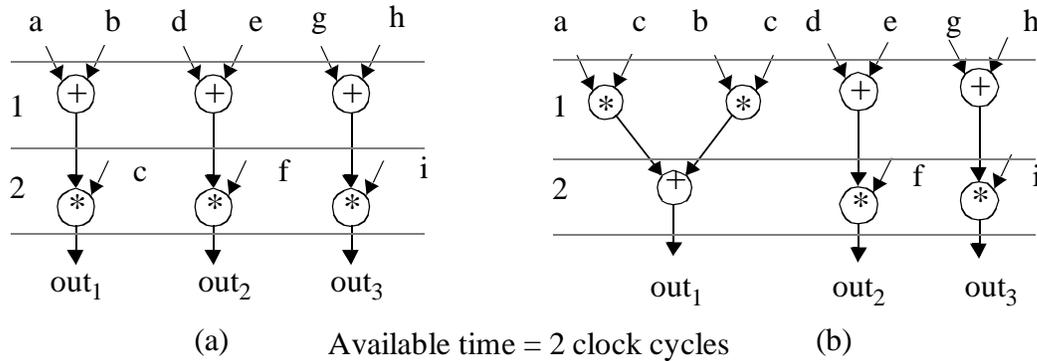


Fig. 5. Transformations: using the distributive identity to reduce the required hardware resources.

available time for computation of two clock cycles, 3 adders and 3 multipliers are needed. If 5a is transformed into 5b using the distributive identity, $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$, however, the resulting structure requires just 2 adders and 2 multipliers. Using the distributive identity, the functional unit requirements are reduced by 50%. Consider next the example of Figure 6a, whose critical path is 12 clock cycles. Using retiming [Lei83] to transform 6a into 6b results in a critical path of just 10 clock cycles. Retiming has reduced the critical path to allow operation at a higher throughput.

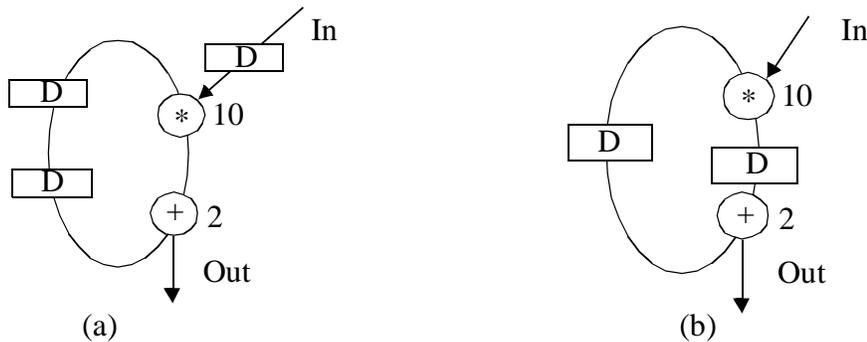


Fig. 6. Transformations: using retiming to improve maximum throughput.

In the optimization of a design, a single optimization technique is rarely sufficient: a sequence of techniques is typically used for greatest performance. Different sets of optimizations and their orderings, however, can result in dramatically different results. Figure 7 shows the effects of applying 2 different optimization sequences on a 20-tap transpose-form Hamming low-pass FIR filter. For a straightforward Hyper implementation of the original design (using the Berkeley low-power library), the estimated power is 427 mW. After applying the set of optimizations of Figure 7a, the power is reduced to 85 mW. Application of the same optimizations, in a slightly different order (Figure 7b) results in a factor of 4 greater improvement, to 21 mW!

- | | |
|--|--|
| <ul style="list-style-type: none"> • maximum pipelining • replace constant multiplications with additions and shifts • select clock frequency and voltage | <ul style="list-style-type: none"> • replace constant multiplications with additions and shifts • maximum pipelining • select clock frequency and voltage |
|--|--|

(a) Sequence 1

(b) Sequence 2

Design version	Power (mW)
Original FIR	427
After sequence 1	85
After sequence 2	21

(c)

Fig. 7. The effect of different orderings of transformations.

While it is clear that an effective sequence of optimization techniques is important, as the next example illustrates, finding an appropriate sequence is often not easy.

Example Motivating the Difficulty of Finding an Effective Sequence of Optimizations

To provide an introduction to the complex interdependencies that exist between optimizations, consider the example of Figure 8. The flowgraph representation of the example algorithm is shown in Figure 8a. The objective is area minimization: for conceptual simplicity, consider only execution units. Each operation takes a single clock cycle to execute and the timing constraint is 3 clock cycles.

Figure 8a shows the initial computation which has three multiplications and two additions. For this structure, two multipliers and one adder are needed to meet the timing constraint. If the computation is pipelined as shown in Figure 8b, a solution requiring just a single multiplier and a single adder results.

Consider next what would happen if the timing constraints were changed from 3 cycles to 2. Under these new timing constraints, area optimization is not so straightforward. In Figure 8a, since all operations are on the critical path, the implementation requires three multipliers and two adders. Pipelining reduces this to two multipliers and an adder, which is better, but not ideal. Using a collection of techniques, greater improvements can be obtained. One such technique that often aids in improving operation distribution and thus resource utilization is distributivity. However, the distributive identity cannot be applied since the result of multiplication m_2 is used in two locations. So, multiplication m_2 can first be replicated as shown in Figure 8c. This transformation results in a structure whose implementation requires 4 multipliers and 2 adders, even more hardware than was initially required. However, now the distributive identity can be applied either once or twice, as shown in Figures 8d and 8e, respectively. It may now seem that the computation of Figure 8d is the preferred solution. Indeed, for the implementation of Figure 8d only 2 multipliers and 1 adder are required, while for the computation shown in Figure 8e,

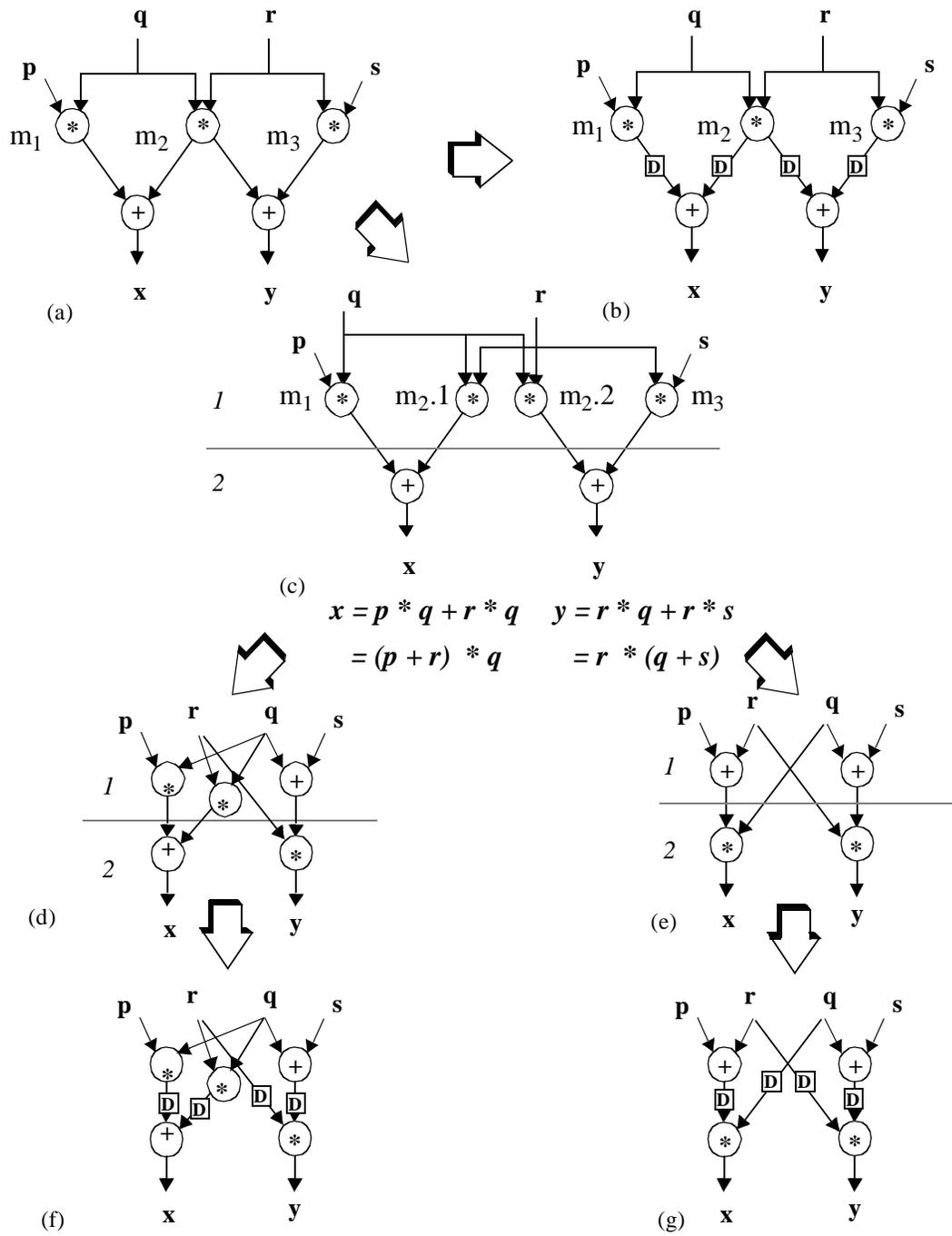


Fig. 8. Ordering of transformations: for an available time of 2 clock cycles, while the initial computation (a) requires 3 multipliers and 2 adders, the final (g) requires only 1 multiplier and 1 adder.

2 multipliers and 2 adders are required. But, if functional pipelining is now applied to both structures, as shown in Figure 8f and 8g, respectively, then 8f requires 1 adder and 2 multipliers while 8g requires just 1 adder and 1 multiplier.

As was shown in the previous example, each small change in constraints, computation structure, or optimization goals can alter the design space greatly. These changes will become even more pronounced when dealing with modern applications, which will require many orders of magnitude greater complexity than the previous example. As a result, high quality designs will require customized optimization obtained through exploration of a variety of potential optimization sequences. In the Hyper paradigm (which is prototypical of other paradigms), an iterative interactive approach is taken. At each stage, estimations are used to determine the placement of the current implementation in the area-throughput-power design space. Using the estimations as guidance, the designer can choose to apply an optimization to improve the solution. After application of the optimization, module selection and estimation are rerun. Exploration proceeds with repeated iteration through optimization, selection, and estimation (Figure 4). During this process, the user is left to analyze the design manually, and to determine which optimizations to apply at each stage. Given the size and complexity of the targeted design spaces, the resulting exploration is often qualitative, ad-hoc, and ineffective.

In Chapter 5, a new property-based methodology and interactive environment for guiding the design exploration process is presented. Using the provided guidance, designers can explore options and make decisions in a more systematic and informed manner.

2.5 Summary

This chapter has presented background information for this thesis. It described the scope of applications and implementations and the flowgraph representation from which

design characterization will be derived. Additionally, the chapter introduced basic ideas within several areas targeted for improvement using a property-based approach — architectural synthesis, estimation, and optimization.

Property-Based Design Characterization

3

Design characterization involves creating an abstraction of the design that contains a specific set of desired information. The underlying idea behind the property-based approach is that a design can be characterized by a small set of relevant, measurable property metrics; these metrics are related to potential performance, and can be used to provide guidance during the optimization and synthesis of the design. Rather than dealing with the design in its entirety, the property metrics provide a simpler, more manageable representation. Design characterization involves identifying and extracting these metrics. In this chapter, the identified metrics are presented. The following two chapters present examples of their use in aiding high-level design and optimization.

3.1 Overview

Property metrics can be used to represent a design's key performance-related characteristics. For example, a quantitative metric such as an algorithm's operation count, albeit a very simple measure, is an effective and often-used indicator of algorithm complexity. To a first order approximation, operation count gives an indication of the required hardware resources or power dissipation. The number of multiplications, in particular, is a common metric used in comparing DSP algorithms [Opp89].

As an example of some other properties, and their effectiveness, consider the following two examples. Figure 9 shows a symmetric direct-form FIR filter before and after the distributive identity is applied to one of the addition-multiplication pairs. In both graphs, the critical path is 4 (assuming each operation has a delay of 1 clock cycle), and all operations lie on the critical path. The original filter structure (Figure 9a) has 7 additions and 4 multiplications, while the transformed one (Figure 9b) has the same number of additions but an additional multiply operation. Given a target throughput of 4 clock cycles, the original filter requires 4 adders and 4 multipliers, while the transformed filter requires just 3 adders and 3 multipliers. Although the transformed filter has a greater number of operations, it has a structure which can be implemented with fewer execution units. A *concurrency* metric can aid in selecting between the two structures, or could be used to guide the transformation from the first to the second.

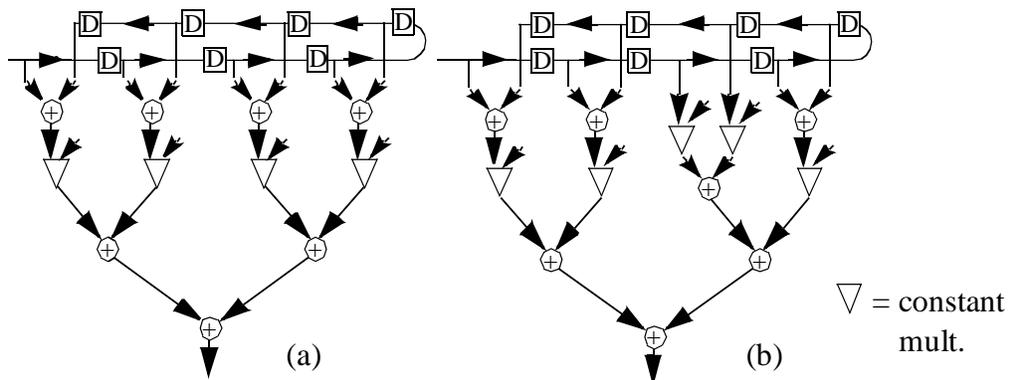


Fig. 9. Size versus operator concurrency.

Consider now the flowgraphs of Figure 10a and 10b, both of which have the same numbers of nodes and edges. The available time is 4 clock cycles. Both graphs have all nodes on the critical path. Nevertheless, while implementation of the flowgraph of Figure 10a requires simultaneous storage of only 2 variables that of Figure 10b requires storage of 4 variables. The edges corresponding to the variables that are alive after the second

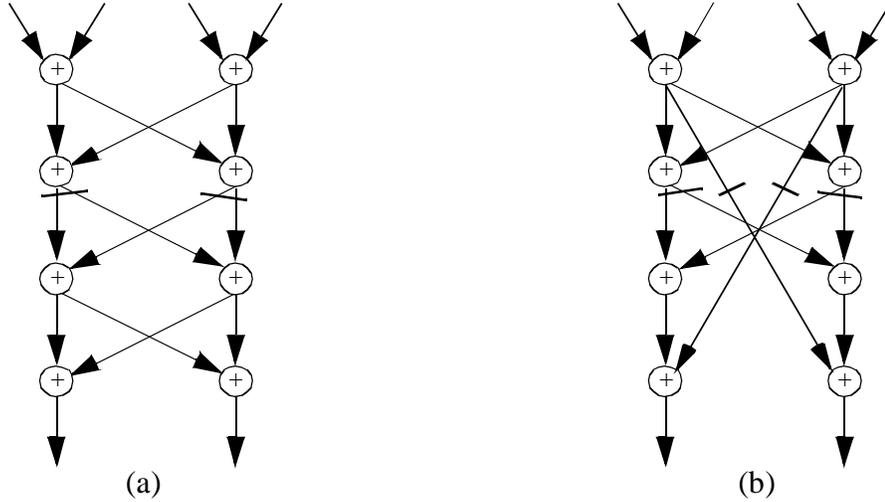


Fig. 10. Size versus variable concurrency.

clock cycle are marked in each figure. The *variable concurrency* metric can be used to indicate the storage needs for implementation of various algorithm structures.

List of Property Classes

The property metrics are organized into the following property classes:

- *Size* metrics characterize the amount of computation that must be executed. They include quantities such as the number of data transfers, operations (functional, input-output, and memory access), and variables (intermediate data words produced by an operation).
- *Topology* metrics characterize the relative arrangement of operations, without regard to their functionality.
- *Timing* metrics include metrics such as the latency, the ratio of sample rate to the critical path, and statistics of the scheduling slack [Wal91].
- *Concurrency* metrics capture the ability for resource accesses to be made in parallel.
- The *uniformity* metrics of a computation capture the degree to which resource accesses are evenly distributed in time over the course of the computation.

-
- *Temporal locality* metrics characterize the persistence of the computation's variables.
 - *Spatial locality* metrics characterize the degree to which the computation is partitionable into clusters of computation.
 - The *regularity* metrics of a computation capture the degree to which common patterns appear.
 - Other property metrics include characterizing the computation as linear or non-linear, and quantifying the degree of non-linearity.

The specific property metrics proposed in this thesis are presented in the following sections and tabulated in Appendix B. While some metrics have been defined elsewhere, others are defined for the first time in this thesis. The usefulness and predictive accuracy of many of these metrics will be demonstrated in Chapters 4 and 5. While some metrics are purely algorithmic (Appendix B, Table 11), for others, the more information the designer provides about the architecture, design constraints, design parameters, and hardware library¹, the more accurate the definitions are (Appendix B, Table 12). For example, the *longest path* through the algorithm is a reasonable indicator of timing. If the operation delays are known, a more accurate *critical path* can be identified.

Selection and definition of property metrics are based on the following requirements and goals. First, it is desired to find a minimal set of metrics that is able to characterize all relevant parts of the algorithm space. Second, the heuristic quantification of the metrics should be of low computational complexity; the aim is for polynomial complexity in the number of nodes and edges of no greater than the second or third power. Thirdly, the met-

1. The design constraints include the required throughput, cost (e.g., area), or power. Relevant design parameters include the supply voltage, clock period, and hardware resource bitwidths. The hardware library specifies the performance (delay, cost, power) of the hardware resources and the underlying set of primitive operations. The library may include simple primitives (e.g., addition, subtraction), complex primitives (mult-add, add-compare-select), and multi-purpose primitives (e.g., ALU).

rics should be relevant in that they are related to potential performance, and can be used to provide guidance during the optimization and synthesis of the design.

Interpretation of the metrics involves linking them to specific design guidance tasks. This research has commenced the study of the interactions between the metrics and their impact on the fundamental cost measures: area, time and power. Although this thesis concentrates on a specific architectural model, the metrics can in general be linked to performance on various architectural models. For example, the critical path and iteration bound are important indicators of speed on an architecture with parallelism, while they are not relevant for a single datapath uni-processor implementation. On a uni-processor, however, the types of operations and operator counts are relevant speed indicators.

Design characterizations have previously been used in a number of ways and areas to improve the mapping of applications onto architectures. In addition to the previous works relating to individual properties described in the following sections, several works have dealt with collections of properties. In the area of parallel computing, Jamieson [Jam87] has proposed characterizations for parallel algorithms and homogeneous parallel architectures to be used in attaining improved mappings between the two. The basic characterization approach presented in this work is similar in motivation to the approach that she has taken. In addition, Papaefstathiou [Pap93] described a framework for parallel software performance prediction based on algorithm characterizations. The use of sets of metrics has also been used in the hardware-software codesign research, as guidance for partitioning [e.g., Tho93, Gon94, Kal93].

3.2 Size

This class of metrics quantifies a computation's size. Size metrics such as bitwidths and operation counts are commonly used indicators of performance. Intuitively, the larger

the computation, the more area, power, and delay one expects. First-cut comparisons of various FFT algorithms, for example, are often based on multiplication counts [Opp89]. In counting operations, a distinction is made between constant and variable multiplications since optimizations can have different effectiveness on each.

In addition to operation counts and bitwidths, there are numerous other size metrics. The number of inputs and outputs gives an indication of the required input-output bandwidth. Since constants must be preserved for the lifetime of the computation, the number of constants gives an indication of the amount of storage that must be allocated for them. Array and variable accesses are linked to the amount of memory bandwidth and storage needed. The number and type of operator classes (e.g., addition, shift, multiplication) and data transfer classes (e.g., shift-to-shift, shift-to-multiply) give an indication of types of hardware and interconnections needed.

3.3 Topology

This class of metrics quantify several features relating to the flowgraph's interconnection of nodes, without regard to functionality. One metric is the node average fanout. High average fanout may indicate the need for increased buffering stages. The next two metrics could be considered part of different property classes but since they don't use information about operation functionality are also topological in nature. One such metric is the flowgraph longest path. The longest path can be found using a modified shortest-paths algorithm in linear time (in the number of nodes and edges) [Cor90], and is a rough indicator of overall timing. Another such metric is the ratio of the number of operations to the length of the longest path, which is a rough indicator of the average concurrency.

A number of other topology metrics are related to feedback or recursion. Consider the biquadratic IIR filter of Figure 11a. The difference equation for this example is:

$$s[n] = a \cdot s[n-1] + b \cdot s[n-2] + u[n]$$

$$y[n] = s[n] + c \cdot s[n-1] + d \cdot s[n-2]$$

In this computation, the variable s at time n is a function of values of the variable s at previous times $(n-1)$ and $(n-2)$. The flowgraph has two feedback cycles, one of which is delimited in the figure by the dashed arrow.

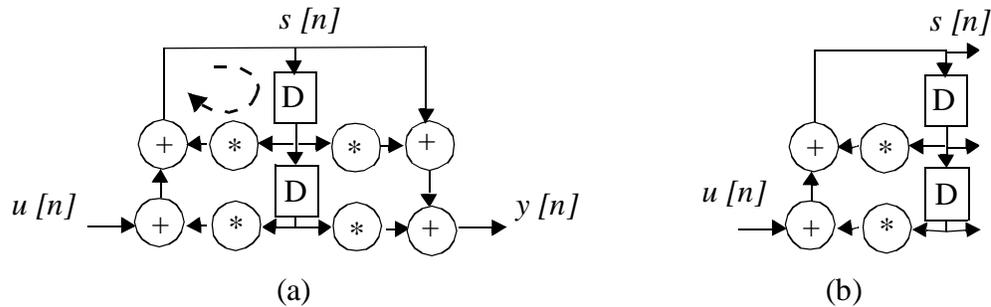


Fig. 11. Topology: a) biquad filter with two cycles, b) corresponding non-trivial SCC.

Useful metrics related to a computation's cyclic structure include the percentage of operations that are in feedback cycles and the number of strongly connected components (SCCs) within the computation. An SCC is defined such that for any pair of operations A and B within a SCC, there exists a path from A to B, and a path from B to A. All operations in non-trivial SCCs (those with more than one operation), are part of a feedback cycle. The example of Figure 11a has a single non-trivial SCC, shown in Figure 11b.

While finding the number of cycles in a flowgraph is computationally expensive, finding the number of SCCs can be done in linear time with a depth-first search [Cor90]. Once the non-trivial SCCs are identified, finding the percentage of operations within cycles is simple, since this value is the same as the percentage of operations in SCCs. Both metrics can be used to determine whether or not the computation has feedback. Additionally, they both give an indication of the degree to which the computation is cyclic. Cyclic sections of a computation warrant special attention since feedback poses a barrier to optimization: it

is well known that any section without feedback can be easily optimized (in terms of throughput) [Mess88]. In order to aid in locating problem areas within a computation, a number of the property metrics presented throughout this chapter are computed not only for the computation as a whole, but also for each individual SCC. These metrics are indicated on the table in Appendix B.

Several other metrics, which characterize the degree to which the computation has natural clusters of computation, are also topological. Since they form an important class of their own, spatial locality, their introduction is delayed to Section 3.6.2.

3.4 Timing

There are a number of useful timing metrics. The metrics assume that each operator has an associated delay of execution as explained in Section 2.3. The hardware library may also be user-defined to contain fictitious relative cost values, relative delays (in clock cycles), and relative power numbers. This allows the designer the opportunity to experiment without having to actually design modules. If a hardware library has not yet been chosen, unit delays are assigned to each operator. In the following sections, $Duration_i$, will represent the clock cycle duration of an operator i .

The critical path is an important timing metric, which determines the time it takes to complete an iteration of the computation. The critical path can be found in linear time with a modified shortest-paths algorithm [Cor90]. In the example of Figure 12, the critical path is 6 clock cycles, or 270 ns. The inverse of critical path gives the maximum throughput at which the computation can be directly implemented (without optimization).

Another metric, the latency of an output, is the delay between the arrival of a set of input samples, and the production of the corresponding output. This is measured as $k \cdot T_s +$

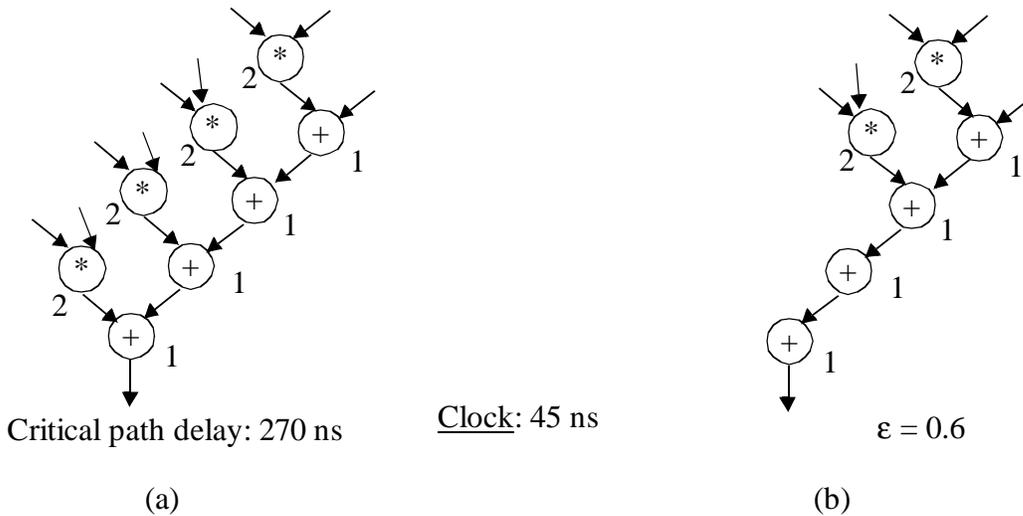


Fig. 12. Example to illustrate timing metrics.

the length of the longest computation path from any primary input or state to the primary output, where k is the number of pipeline stages that have been added and T_s is the sample period. Srivastava and Potkonjak have presented a detailed treatment of the definitions of and the relationships between throughput and latency [Sri95b].

Several timing metrics give a measure of how constrained the system is, such as the critical path to sample period ratio, metrics related to the ϵ -critical network [Sin88], and metrics related to the scheduling slack. The ϵ -critical network is a subset of the flowgraph containing all nodes and edges on paths which are "almost" critical. More precisely, paths of length L or greater are included, where $L = (1 - \epsilon) \cdot CP$, and CP is the critical path length. This is measured for a range of ϵ values, $0 \leq \epsilon \leq 1$. For example, Figure 12b shows the ϵ -critical network assuming an ϵ of 0.6. Metrics such as the percentage and types of nodes on the ϵ -critical network give an indication of the computation's structure and its potential for optimization. For example, different techniques are generally used to speed up a computation with a single long path versus one with many long paths. A num-

ber of the property metrics presented throughout this chapter are computed not only for the computation as a whole, but also for the ε -critical network.

The slack metrics are the average, quartiles, and variance of the slack. Quartiles divide the data into four parts at 25, 50, and 75%. Thus, 25% of the observations are less than or equal to the first quartile and 75% are less than or equal to the third quartile. Notice that the second quartile is also the median. These slack metrics are measured for each operation type. They give an idea of how much flexibility the scheduler will have, and can thus be used in predicting scheduler effectiveness since some scheduling heuristics work better than others under tight constraints.

Other important metrics are the iteration bound and the ratio of the iteration bound to the critical path. The iteration bound is defined as the maximum, over all feedback cycles, of the ratio of the total execution time of all operations in the cycle, $T(c)$, divided by the number of sample delays in that cycle, $D(c)$:

$$c \hat{I} \underset{\text{cycles}}{\text{Max}} \frac{T(c)}{D(c)}.$$

The iteration bound for non-recursive computations is 0. Iteration bound metrics are good predictors of the potential improvement in throughput by application of transformations such as pipelining, retiming, and time-loop unfolding. An efficient algorithm for its calculation can be found in [Ito95]. A final timing metric is the maximum operator delay. This metric is useful, for example, when performing clock selection.

3.5 Concurrency and Uniformity

In both the general-purpose and scientific-computing community, concurrency has been studied extensively and its exploitation has resulted in greatly improved performance

in terms of metrics such as throughput and latency. Studies to determine the amount of available concurrency have been conducted, and results have influenced both architecture and compiler design [Kum88, Nic84, Tja70, Jou89]. In this section, metrics for quantifying the concurrency and uniformity of dataflow graphs are presented. The concurrency metrics are presented for operations, register accesses, data transfers, and variables in sections 3.5.1 - 3.5.4. The uniformity metrics characterize the degree to which these resource accesses are evenly distributed over the course of the computation, and are presented in Section 3.5.5.

3.5.1 Concurrency of Operations

The operator concurrency quantifies the expected number of operations that will be simultaneously executed. The first step in its calculation is construction of a distribution graph, which quantifies an expected number of operations to be executed in each clock cycle. Paulin first proposed and used information from such graphs for force-directed scheduling [Pau89]. Consider the computation of Figure 13a, in which each operator is assumed to require 1 clock cycle to execute. In order to complete the computation in 4 clock cycles, two multiplications must be done in each of the first two clock cycles, two additions in the third clock cycle and one addition in the last cycle. Figure 13b displays the distribution graph for multiplication operations. The x-axis specifies the time slots, and the y-axis the number of operations executing in each. In this example, the execution time slot of each operation is exactly specified by the dependency and timing constraints.

In general, since the exact times in which each operation will be executed is not known until after scheduling, it is necessary to determine the probability that the operation will be executed in a particular time step. To calculate the true probabilities, it is necessary to enumerate all possible schedules. Since this is a computationally formidable problem, approximations of these probabilities must be derived. There are several ways in which to

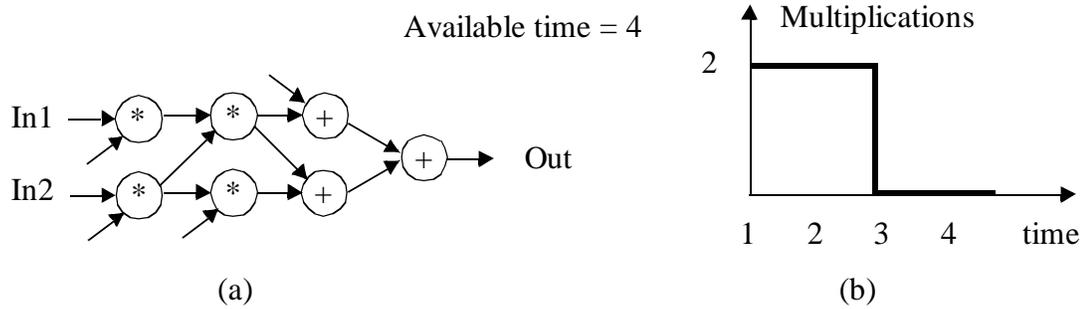


Fig. 13. Concurrency: a) flowgraph, b) distribution graph of multiplication operations.

calculate the approximated distribution graphs, with different trade-offs between computational complexity and accuracy. The solution proposed by Paulin is to assume a uniform probability that a particular operation, i , will be scheduled in each of its potential time slots between its as-soon-as-possible ($ASAP_i$) and as-late-as-possible ($ALAP_i$) times [Pau89]. A possible alternative is a triangular probability distribution. In the case of a uniform probability distribution, an operation i makes the contribution $\frac{1}{ALAP_i - ASAP_i + 1}$ for each time slot k , $t \leq k < t + Duration_i$, for each t , $ASAP_i \leq t < ALAP_i$. The values of the distribution graph are the accumulation of the contributions from all operations of the specified type.

From the distribution graphs, several operation concurrency metrics can be measured. For example, the maximum height of the distribution graph, $maximumheight_j$, is an approximation of the maximum number of simultaneous operations and therefore the required number of resources of type j . Another metric is

$$maximumheight_{\Psi} = \text{Max}(1, maximumheight_j).$$

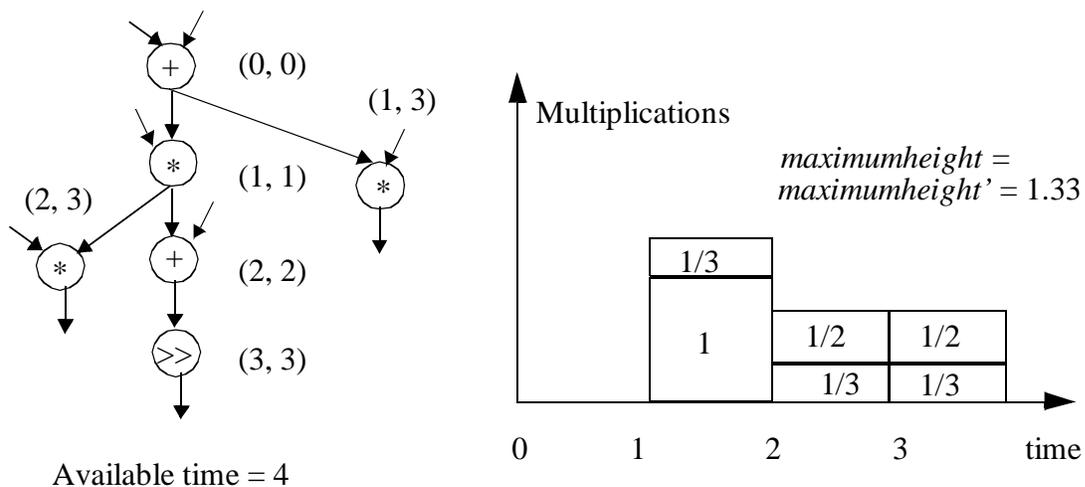


Fig. 14. Sample flowgraph and corresponding distribution graph.

This is used, since while $maximumheight_j$ may be less than one, there must always be at least one unit of a given type. Two combined metrics over all operator types are also defined. The first is the sum of the maximum heights of the individual distribution graphs. The second is the maximum height of the sums of the individual graphs.

An example showing the above concepts is given in Figure 14. The available time is assumed to be four clock cycles, each operation takes one clock cycle, and the ASAP and ALAP scheduling times are shown in parentheses for each node. The distribution graph for multiplications is also shown. Its maximum height is 1.3.

3.5.2 Concurrency of Accesses to Registers

Another design characteristic of interest is the concurrency of accesses to the various register files. Under the target architecture model, only write accesses are an issue since read accesses are guaranteed. Lack of sufficient register write bandwidth can be a bottleneck leading to reduced performance or the need for additional resources (execution units or additional register file ports). Consider the example of Figure 15a. For an available time

of 3, it appears, at first glance, that a single adder resource is sufficient. The adder could execute a single add operation in each of the 2nd and 3rd time steps. This is not the case, however, for the hardware model of Appendix A, which uses a dedicated register file model with single-ported registers. For this computation, 2 variables must simultaneously be written into the adder's second input port register file at the end of the first time step. These data transfers are dashed in the figure. Also, the adder port numbers are indicated. To meet timing constraints, 2 adders are thus needed. As the example of Figure 15a has illustrated, indication of required execution units may depend not only on concurrency of operations but also concurrency of register write accesses. Note that an operator's operands can in some cases be commuted to improve register access concurrency. Consider the application of the commutative identity to addition a_1 of Figure 15a to generate Figure 15b. While the computation of Figure 15a requires 2 adders to support register write accesses that of Figure 15b requires just 1.

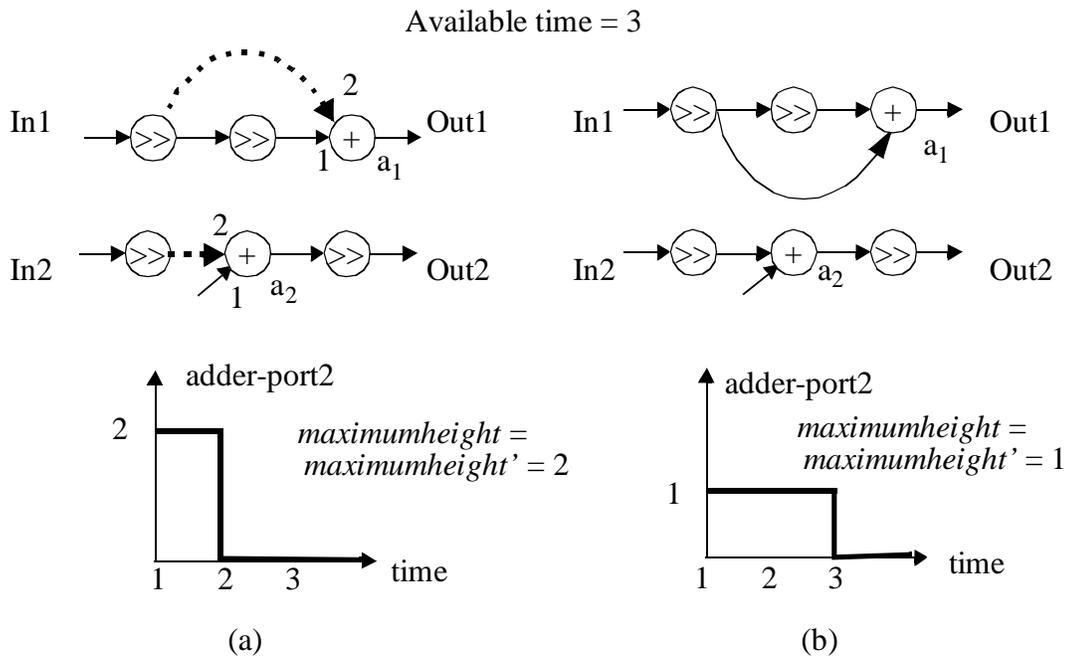


Fig. 15. Register access concurrency: a) original computation and adder-port2 distribution graph, b) after application of the commutative identity to adder a_1 .

To measure register access concurrency, a separate distribution graph is built for each input port of each resource type. In other words, for adders, a separate graph is constructed for adder-port1 and adder-port2. Construction of these distribution graphs are similar to those for operators. A uniform probability that a particular operation, i , will be scheduled in each of its potential time slots between its as-soon-as-possible ($ASAP_i$) and as-late-as-possible ($ALAP_i$) times is assumed. In this case, an operation i that fans out to the register

file of interest makes the contribution $\frac{1}{ALAP_i - ASAP_i + 1}$ to the time slot $(t + Duration_i -$

$1)$, for each t , $ASAP_i \leq t < ALAP_i$. If a variable fans out to multiple resources of the same type, a single write to the corresponding resource is assumed. The values of the distribution graph are the accumulation of the contributions from all operations that fanout to the specified type of register file. For each resource type j , register access concurrency metrics $maximumheight_j$ and $maximumheight'_j$ are found. These heights are determined by the resource's input port with the greatest concurrency: for a two-ported resource j , $maximumheight_j = Max(maximum height_{j-port 1}, maximum height_{j-port 2})$. For example if the adder right input has maximum height of distribution graph of 15, and the left input has value 5, the register access $maximumheight$ metric for adders is $\max \{15, 5\} = 15$. The metric $maximumheight'_j$ is the maximum of 1 and $maximumheight_j$.

3.5.3 Concurrency of Data Transfers

Another measure of interest is the concurrency of data transfers over time. Under the target architecture model, data transfers takes place in the cycle that the data production completes. Since the exact execution time of the operations is not yet known, the contribu-

tion $\frac{1}{ALAP_s - ASAP_s + 1}$ is made to each time slot $(t + Duration_s - 1)$ for each t , $ASAP_s$

$\leq t < ALAP_s$. The values of the distribution graph are the accumulation of the contribu-

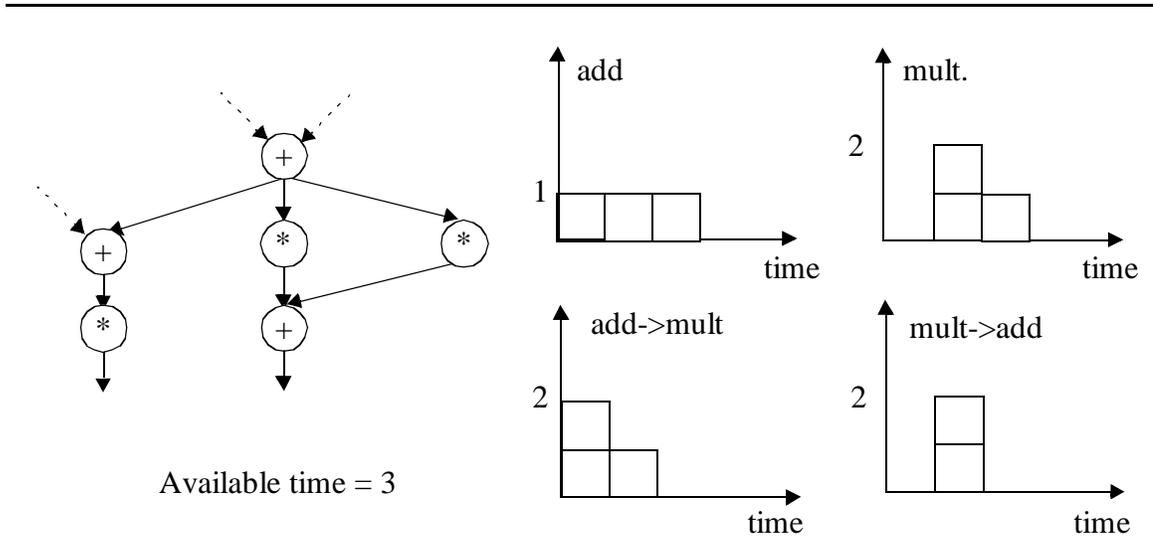


Fig. 16. Operation and data transfer distribution graphs.

tions from accesses of the specified type, where each source-destination pair is considered a distinct data transfer resource type. For each resource type j , data transfer concurrency metrics $maximumheight_j$ and $maximumheight_j'$, are found.

Figure 16 presents an example to illustrate the presented concepts. It is assumed that all operators have a duration of one. The upper two distribution graphs correspond to addition and multiplication accesses. The lower two correspond to add->mult and mult->add interconnection accesses.

3.5.4 Concurrency of Variables

This section presents techniques to measure the concurrency of variable lifetimes. Variable concurrency is slightly different than the previous three cases considered so far. While in the previous cases the main concern was an unclear execution time, in this case, not only are the creation times of each variable unclear, but also the consumption times. In this section, a classification of variables will first be presented (Section 3.5.4.1). Next

three techniques for estimating the lifetimes of variables will be given (Section 3.5.4.2). Finally the construction of the variable distribution graph is presented (Section 3.5.4.3).

3.5.4.1 Formal Classification of Variables

All variables can be classified into three groups with respect to their lifetimes: provably local, provably global, and scheduling-dependent. Let us thus assume that a variable V has a single producer node P , and multiple fanout nodes C_i . The variable V is thus associated with multiple fanout edges (P, C_i) . In these definitions, we refer to two types of edges: delay edges are those which pass through a sample delay, and non-delay edges are those which do not. The available time is specified by the variable T .

A provably local variable is one for which it is certain that the variable will be produced and then consumed in the immediate following clock cycle in all feasible schedules. For this to hold, all of the fanout edges must be provably local. A fanout edge is provably local if the following necessary and sufficient conditions are met:

$$\text{Non-Delay:} \quad \text{ASAP}_P = \text{ALAP}_P \text{ and } \text{ASAP}_{C_i} = \text{ALAP}_{C_i}.$$

$$\text{Delay:} \quad \text{ASAP}_P = \text{ALAP}_P = T + 1 - \text{Duration}_P \text{ and} \\ \text{ASAP}_{C_i} = \text{ALAP}_{C_i} = 1.$$

A provably global variable is one for which it is certain that the variable cannot be consumed immediately because of data or control dependencies, and thus must be alive for more than one cycle. This condition is true as long as one of its fanout edges are provably global. Such edges are termed feed-forward edges for a non-cyclic graph. The following conditions are necessary and sufficient to recognize a fanout edge as provably global:

$$\text{Non-Delay:} \quad \text{ASAP}_{C_i} > \text{ASAP}_P + \text{Duration}_P.$$

$$\text{Delay:} \quad T - \text{ALAP}_P + \text{ASAP}_{C_i} > \text{Duration}_P.$$

The last class of variables are called scheduling-dependent, and consist of those variables which do not fall into one of the previous two classes.

An example of all three types of fanout edges can be found in the graph of Figure 17. The available time is assumed to be three clock cycles, each operation takes one clock cycle, and the ASAP and ALAP scheduling times are shown in parentheses for each node. Note that V_1 and V_2 are provably global variables and V_3 is a provably local variable.

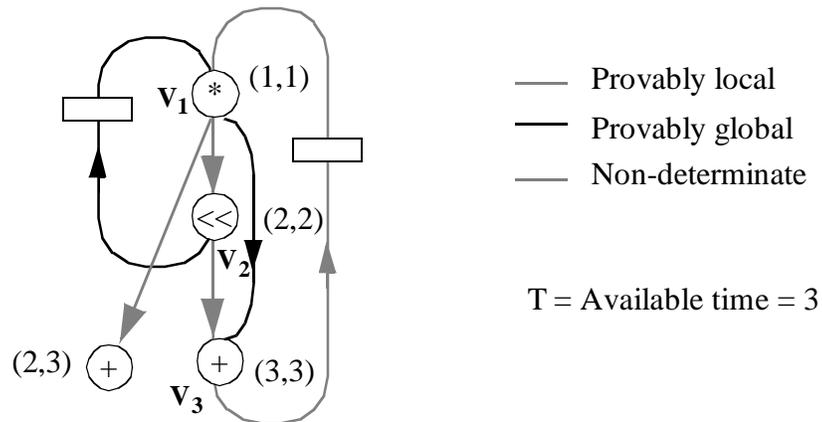


Fig. 17. Classification of edges and variables.

3.5.4.2 Calculations of Lifetime Distributions for Variables

The expected lifetimes and the exact time slots in which the variables will be stored can be easily found for provably local variables. These variables have an expected lifetime of 1; start and end times can be easily derived since the exact execution times of its producer and consumers are known. For provably global and non-determinate variables, however, the execution times of the operations are not known, as they are not determined until scheduling is performed. It is thus necessary to develop techniques to estimate the times in which a variable is expected to be alive. This subsection describes three measures that this thesis proposes for predicting the expected lifetime distributions of the variables in the flowgraph.

Midpoint Approximation

In this approach, execution times of all operations are predicted to occur in the midpoint of their ASAP - ALAP range. For each variable, v , the predicted start and end times are thus calculated as follows:

$$\text{Start}_v = 0.5 \cdot (\text{ASAP}_x + \text{ALAP}_x) + \text{Duration}_x,$$

$$\text{End}_v = \text{Max}_{y_i \in y} (0.5 \cdot (\text{ASAP}_{y_i} + \text{ALAP}_{y_i}) + \text{Duration}_{y_i})$$

where x is the producer node, and $y_i \in y$ are the consumer nodes. Figure 18 illustrates this calculation.

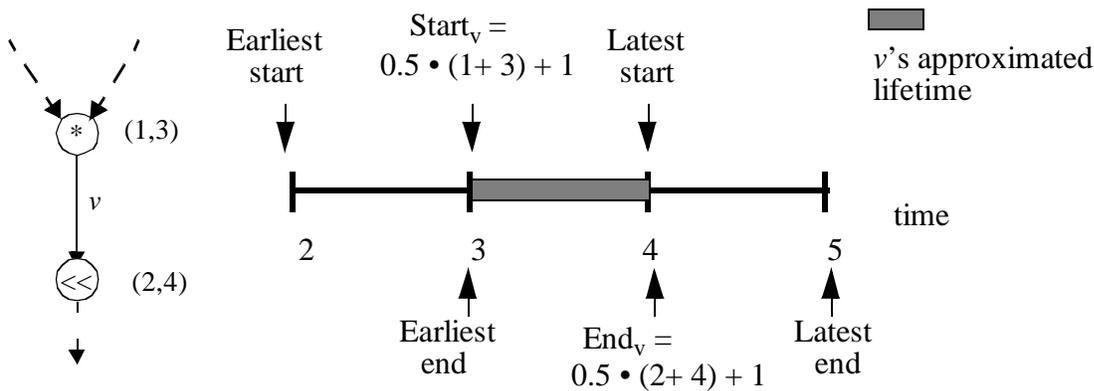


Fig. 18. Midpoint approximation of variable lifetime.

Distributed Approximation

Another approach involves calculating conditional probabilities, $P_v(t)$, that the variable v will be alive during time slot t . This is calculated locally taking into account the slacks of the consumer and producer of the variable, and the dependency between them. If a variable fans out to several operations, these values are independently calculated for each fanout edge, then the maximum probability of existence is used at each time slot.

Depending on producer and consumer ASAP and ALAP times, two cases are defined for the computation: i) the variable is produced and consumed in the same sample period; ii) the variable is produced and consumed in different sample periods.

Case 1: $ASAP_{y_i} > ASAP_x$

"t: 0 $\leq t < T$

if $(ASAP_x + Duration_x \leq t < ALAP_{y_i} + Duration_{y_i})$

$$P_v(t) = \left(\sum_{j=ASAP_x}^{t-Duration_x} P_x(X=j) \cdot P_{y_i}(Y_i > t-Duration_{y_i} | X=j) \right)$$

else

$$P_v(t) = 0.$$

Case 2: $ASAP_{y_i} \leq ASAP_x$

"t: 0 $\leq t < T$

if $(ASAP_x + Duration_x \leq t < ALAP_x + Duration_x)$

$$P_v(t) = \left[\sum_{j=t-Duration_x-1}^{ASAP_x} P_x(X=j) \sum_{k=t-Duration_x+1}^j P_{y_i}(Y_i=k) \right] + \sum_{z=ASAP_x}^{t-Duration_x} P_x(X=z)$$

else if $(ALAP_x + Duration_x \leq t < ASAP_{y_i} + Duration_{y_i})$

$$P_v(t) = 1$$

else if $(ASAP_{y_i} + Duration_{y_i} - 1 < t < ASAP_x + Duration_x)$

$$P_v(t) = 1 - \sum_{j = ASAP_{y_i}}^t P_{y_i}(Y_i = j)$$

else

$$P_v(t) = 0.$$

Figure 19 shows a simple example of these calculations. Note that this approximates the probability that a variable is alive in each time slot, and the area under the curve should not necessarily be 1.

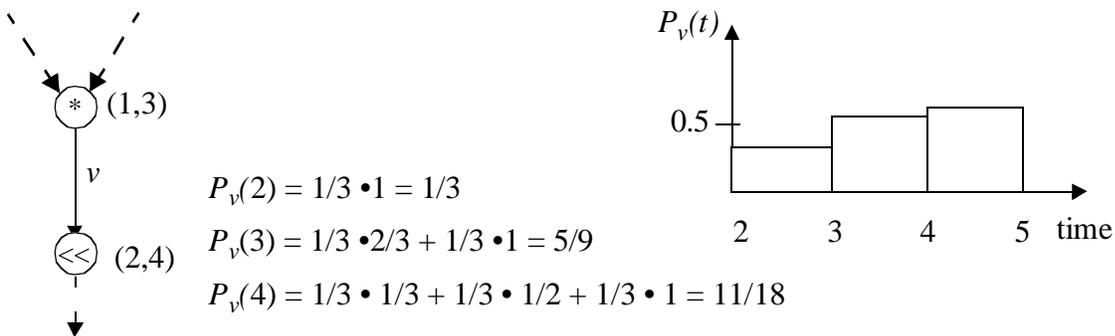


Fig. 19. Approximation of probability that the variable v is alive at time t .

Distributed Approach with Split Variables

In the previous two sections, the expected lifetime of each variable was computed, independent of its fanout operation types. Under the dedicated register file model, however, a variable copy must be stored in the register file at the input of each destination hardware unit. For example, a copy of the variable v in Figure 20 would be stored at the adder input register and at the shifter input register (assuming that both shifters are mapped to the same shift hardware unit). A modification to the analysis of the previous section is thus proposed, to reflect the variable copies that are generated in implementa-

tion. For the purpose of lifetime and storage prediction, each variable is split into N parts, where N is the number of distinct operator fanout types. Figure 20 shows variable v split into v_1 and v_2 .



Fig. 20. Splitting variables: a) original graph, b) graph after splitting variable v into v_1 and v_2 .

3.5.4.3 Variable Distribution Graphs

The distribution graph of variable lifetimes can be constructed from the calculations of the previous section. In the case when the midpoint approximation is used to determine the lifetimes of all variables, each variable contributes a value of 1 to the time slots it is expected to be alive. In the case when the distributed approach is used, each variable contributes $P_v(t)$ to each time slot t . The maximum concurrency metric in both cases is the maximum height of the resulting distribution graph:

$$Maxconcurrency = \text{Max}_{t \in [0 \dots T]} \left(\sum_{v=1}^{NumVariables} P_v(t) \right)$$

The variable maximum concurrency metrics estimate the maximum number of variables expected to be alive at any given time.

3.5.5 Uniformity

The uniformity metrics characterize the degree to which resource accesses are evenly distributed over the course of the computation.

A metric to quantify uniformity is the variance of the distribution graph. Recall that the distribution graphs show how well operations are distributed over time. In general, the greater the variance in these graphs, the less uniform the structure is, and hence the more susceptible it is to improvement by transformations which alter ASAP and ALAP times. The retiming-for-area transformation [Pot91] is one such optimization. Section 4.1 shows examples of using this metric in this way.

3.6 Locality

In the architecture and compiler areas, the concept of locality has been heavily studied and utilized during the last three decades [e.g., Kuc78, Pat90]. In particular, it has greatly influenced the design of memory hierarchies [Pat90]. Locality is the qualitative property of typical programs that 90% of execution time is spent on 10% of the instructions [Knu71]. In those domains, temporal locality is described as the tendency for a program to reuse data or instructions which have been recently used. Spatial locality is the tendency for a program to use data or instructions neighboring that which was recently used. Recently, techniques for enhancing locality in parallel distributed programs have attracted a great deal of attention [Wol89]. In the VLSI signal processing and architecture communities, research on the exploration of locality started in the late seventies. The classification of recursive algorithms, as having either inherently local or global communications has been used as an indicator of an algorithm's suitability for VLSI implementation [Kun88].

In the following sections, metrics for quantifying temporal locality and spatial locality for ASIC implementations are introduced and defined. Note that they take on different meanings from those used in the computer architecture domain.

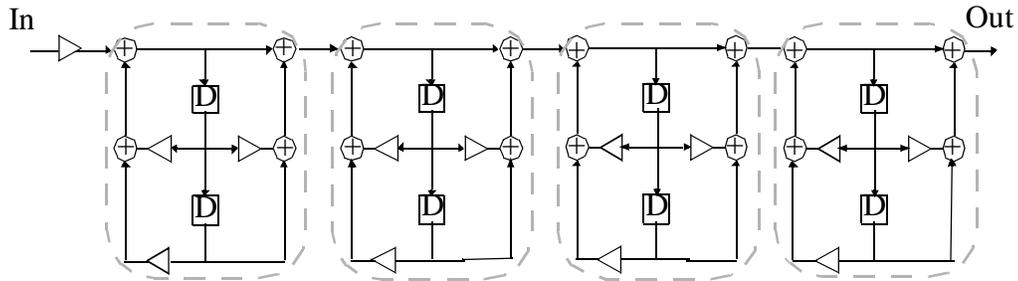


Fig. 21. Cascaded biquad with four clusters of computation.

3.6.1 Temporal Locality

Temporal locality characterizes the persistence of the computation's variables. More specifically, a computation is temporally local if its variables have short predicted lifetimes. Variables with short predicted lifetimes are good candidates for storage in foreground (register) memory. Temporal locality metrics include the average, the quartiles, and the variance of the variable lifetimes derived using the midpoint technique (Section 3.5.4.2).

3.6.2 Spatial Locality

Spatial locality indicates the degree to which a computation has natural isolated clusters of operations with few interconnections between them. For example, a cascaded biquad IIR (Figure 21) has isolated clusters of computation (each biquad), with only a single connection between each one. In the decimation-in-time FFT algorithm, on the other hand, no clear clusters exist. Identification and use of spatially local sub-structures can be used to guide hardware partitioning (inter and intra-chip) and minimize global buses. This can lead to lower area and to lower bus capacitance, and thus lower interconnect power.

There are several metrics that quantify a computation's spatial locality. Two such metrics are the number of isolated components in the computation and the number of bridges

[Cor90]. A bridge is an edge whose removal disconnects the graph. For this metric, each edge with fanout is represented by a collection of edges, one for each fanout. A third metric is the *connectivity* ratio:

$$connectivity = \frac{\text{number of actual internal data transfers}}{\text{number of possible internal data transfers}}.$$

An *internal* data transfer occurs between two operations. Inputs, outputs, and constant data are examples that do not meet this criteria. The number of possible internal data transfers is equal to the total number of input ports over all operations. For example, each of the computations in Figure 22 have 7 inputs: 2 for each of 3 additions and 1 for the sample delay operation. A connectivity ratio of 0 means the graph is fully unconnected. A ratio of 1 means the graph is fully connected. Figure 22 provides examples of each of these metrics.

Another approach to characterizing spatial locality is based on spectral methods. Spectral methods use eigenvalues and eigenvectors of matrices derived from the flowgraph. In particular, work in spectral methods has shown that the eigenvectors of the Laplacian matrix of the graph [Moh88] represent normalized one-dimensional placements of graph nodes which minimize their mean-squared connection length [e.g., Hal70] (presented below). The 2nd smallest eigenvector placement provides a representation of the computation in which distances between consecutively located nodes indicate their affinity, and thus highlight the computation's spatial structure. Given the eigenvector placement, a specific metric that can be measured is the number of "large gaps": the number of distances that exceed $\mu + a \cdot \sigma$, where μ is the average distance between consecutively placed operations, σ is the standard deviation in distances, and a is a constant equal to 1, 2, or 3. The application of spectral methods to the flowgraphs targeted in this thesis has been done in collaboration, and is also presented in [Meh96a].

	<u>#isolated</u>	<u>#bridges</u>	<u>Connectivity</u>
	3	1	$1/7 = .14$
	1	3	$3/7 = .43$
	1	2	$4/7 = .57$
	2	0	$4/7 = .57$
	1	0	$4/7 = .57$
	1	0	$6/7 = .85$

Fig. 22. Example of locality metrics.

As an example, consider the one-dimensional node placement derived from the 2nd smallest eigenvector of the Laplacian of the 4th-order parallel IIR shown in Figure 23. The structure consists of 2 biquad clusters, each of whose nodes are grouped together on the left and right sides of the placement. The two multiplications (labeled a and b in the figure) are both located in the middle of the placement.

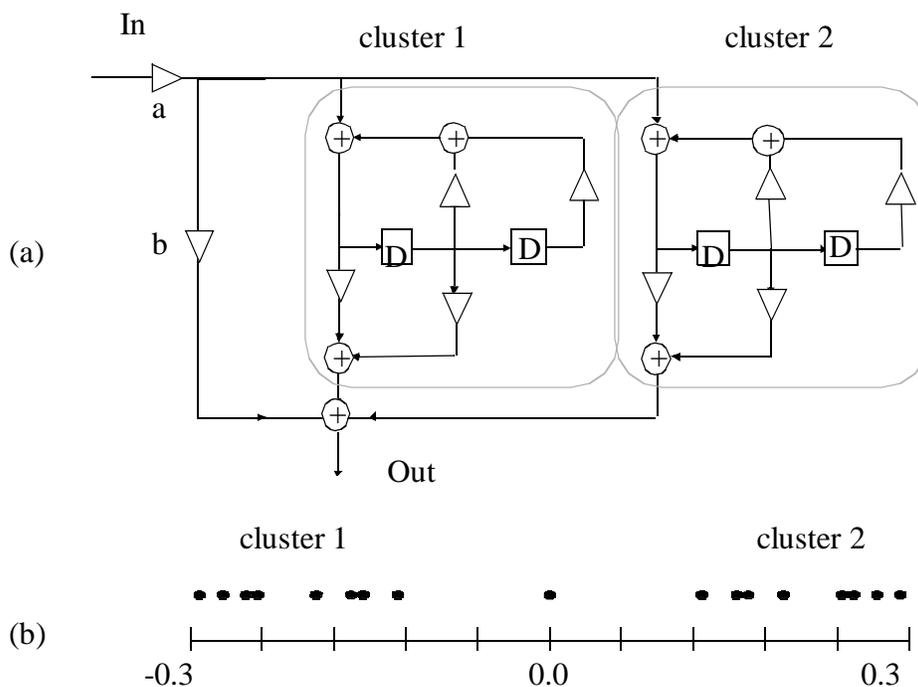


Fig. 23. Fourth-order parallel IIR: a) flowgraph, b) node placement given by the second smallest eigenvector of the graph Laplacian.

Hall's derivation

The derivation presented by Hall [Hal70], and reproduced here, showed that the eigenvectors of the Laplacian matrix solve the one dimensional quadratic placement problem. The problem involves finding the vector $\bar{x} = (x_1, x_2, \dots, x_n)$ of the nodes of a given weighted-edge non-directed graph that minimizes the weighted sum of squares of the edge lengths:

$$z = \frac{1}{2} \sum_i^n \sum_j^n (x_i - x_j)^2 A_{ij}$$

subject to the constraint:

$$|\bar{x}| = \sqrt{(\bar{x})^T \bar{x}} = 1.$$

The constraint is specified to normalize the placement between -1 and 1.

Let A be the weighted adjacency matrix of the graph, where A_{ij} is the weight of the edge between nodes i and j : $A_{ij} = 0$ if there is no edge between i and j . Define a degree matrix, D , as the diagonal matrix in which each diagonal element is the sum of the weights of all the edges connecting to the corresponding node. The cost function z can be rewritten as $\bar{x}^T (D - A)\bar{x}$. The matrix $Q = (D - A)$ is the Laplacian of the graph. The constrained cost function is given by the Lagrangian, L as

$$L = \bar{x}^T Q \bar{x} - \mathbf{1}(\bar{x}^T \bar{x} - 1)$$

Setting the derivative of the Lagrangian, L , to zero gives the following equation:

$$(Q - \mathbf{1}\mathbf{1}^T)\bar{x} = 0,$$

whose solutions are those where λ is the eigenvalue and \bar{x} is the corresponding eigenvector. The smallest eigenvalue, 0 , gives a trivial solution with all nodes at the same point. The eigenvector corresponding to the second smallest eigenvalue minimizes the cost function while giving a non-trivial solution.

Graph (hyperedge) representation

The graph used to form the Laplacian is assumed to be undirected and to contain edges without fanout. Recall that the edges within the assumed flowgraph representation are “hyperedges” since they may fanout to several locations. A conversion of hyperedges to edges without fanout is thus necessary.

Figure 24 shows several techniques that replace the hyperedge by edges. All of them replace the hyperedge by edges between pairs of nodes to form a clique but differ in the weight assigned to the resulting edges. The first is the uniformly weighted clique model

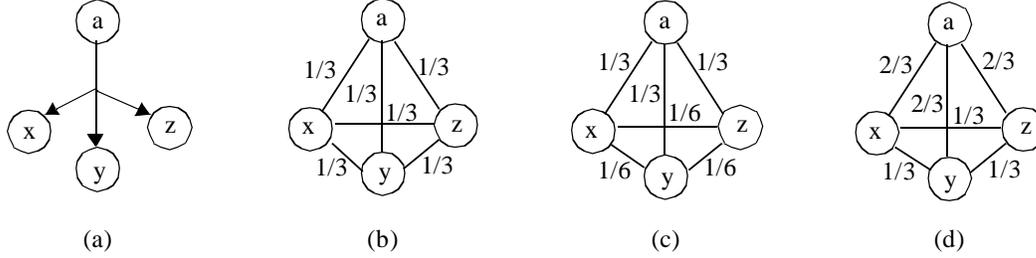


Fig. 24. Hyperedge models: (a) initial hyperedge, (b) uniformly weighted clique model, (c) clique model with lower weights on edges between destination nodes, (d) clique model with higher weights on edges connected to the source node [from Meh96a].

[Len90], which has been widely used for layout partitioning. This model assigns a weight of $1/(k-1)$ to all edges in the clique, where k is the number of nodes in the clique (Figure 24a and Figure 24b). Additional models proposed by Mehra *et al.* [Meh96] weight the edges between nodes x , y , and z less than the edges connecting a to the nodes x , y , or z . In the model shown in Figure 24c, a lower weight, $1/2(k-1)$, is assigned to edges between the destination nodes leaving the weights on the other edges unchanged. In the model of Figure 24d, the weight of the edges that join the source to the destination nodes is increased to $2/(k-1)$, while the weights on edges between the destination nodes is unchanged.

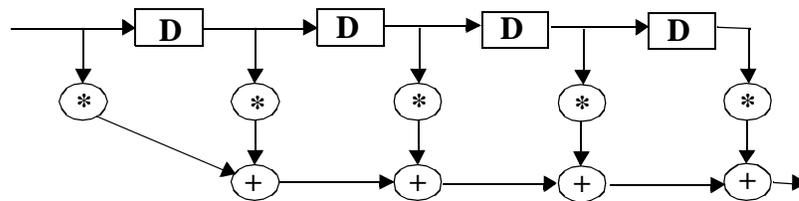
3.7 Regularity

Another important computation characteristic is *regularity*. Regularity has been addressed in a wide range of domains and contexts. For example, in complex VLSI system designs, structured design methodologies resulting in regular designs have gained widespread acceptance [Mea80, Seq83]. The development of systolic and wave-front processors [Kun84, Kun88] can be largely attributed to efforts to effectively exploit computation regularity. Vector processors [Hwa84] are often used to exploit loop level regularity. Bhattacharyya and Lee [Bha93] exploited regularity in their development of a looped scheduler, which synthesizes code for programmable DSPs with reduced program and memory requirements. In the high-level synthesis literature, Note *et al.* [Not91] and

Geurts *et al.* [Geu93] from IMEC, explored regularity for the module selection problem in high level synthesis. Rao and Kurdahi [Rao92, Rao93] demonstrated the use of regularity in a variety of high level synthesis tasks, including partitioning and scheduling. Common to all is the identification and use of recurring “patterns” in the design. In this section, approaches for quantifying the hitherto intuitive notion of regularity are presented.

3.7.1 Quantification of Regularity

One metric of regularity is the percentage operation coverage by common patterns or by a set of common patterns. For example, coverage by the set of patterns including all chained pairs of associative operators can give an indication of the potential for improvement using the associative identity. In computing the coverage, all operations that occur in at least one match of the pattern are counted. For example, the mult-add pattern matches in 5 places in the FIR of Figure 25, covering all multiplication and addition operators. This results in a coverage of 9 out of 13 operations, or 70%. Template matching is done using the technique of [Cor96].



Coverage

add-add pattern: 31%

mult-add pattern: 70%

set containing add-add and mult-add patterns: 70%

Fig. 25. Regularity: percentage coverage by patterns.

Another quantification of regularity is given by the following metric:

$$\text{regularity} = \text{size} / (\text{descriptive complexity})$$

The *size* is the total number of operations and data transfers executed in the computation. The *descriptive complexity* is a heuristic measure of the shortest description from which the graph can be reproduced (Section 3.7.2).

Intuitively, since a graph with repeated patterns is easy to describe, the inverse proportional relation between regularity and descriptive complexity is defined. In comparing two graphs with the same complexity but different sizes, the larger would be considered more regular. The proportional relation between regularity and size is thus established. The regularity is a quantity in the range $[1, \text{size}]$. The lowest value of regularity is 1, which occurs when the complexity is equal to the size. Figure 26 shows an example of how the regularity scales with size. The first graph has a size of 25 since it has 10 operations and 15 data transfers. The sizes of the next two graphs are 49 and 97, respectively. The complexity of each of the three graphs is approximately the same: 16, 18, and 20, respectively (the approach for measuring complexity is described in the next section).

3.7.2 Complexity

Motivated by the underlying ideas in descriptive complexity of strings, a measure of graph complexity has been developed. First, Section 3.7.2.1, presents the background theoretical work on string complexity. Then in Section 3.7.2.2, a methodology for determining heuristic graph complexity is presented.

3.7.2.1 Theoretical Kolmogorov Complexity of a String

In the mid-sixties, three researchers Kolmogorov, Chaitin, and Solomonoff independently proposed a theoretical measure of the algorithmic information content of sequences. This work is known today under a variety of names including K-complexity, algorithmic complexity, program-size complexity, Kolmogorov-Chaitin randomness, and

Kolmogorov complexity. Surveys on the theory and applications of Kolmogorov complexity can be found in [Li90, Cov91].

The main idea is that “the amount of information in a finite object such as a string, is the size, in bits, of the smallest program that, starting with blank memory, outputs the string then terminates” [Li90]. Although there are many programs which can generate a given finite sequence, the shortest one gives a good indication of the information content of that sequence. A more formal definition is: the Kolmogorov complexity $K_U(x)$ of a string x with respect to a universal computer U is:

$$K_U(x) = \min_{p:U(p)=x} \text{length}(p)$$

where $U(p)$ is the output of the computer U given the program p .

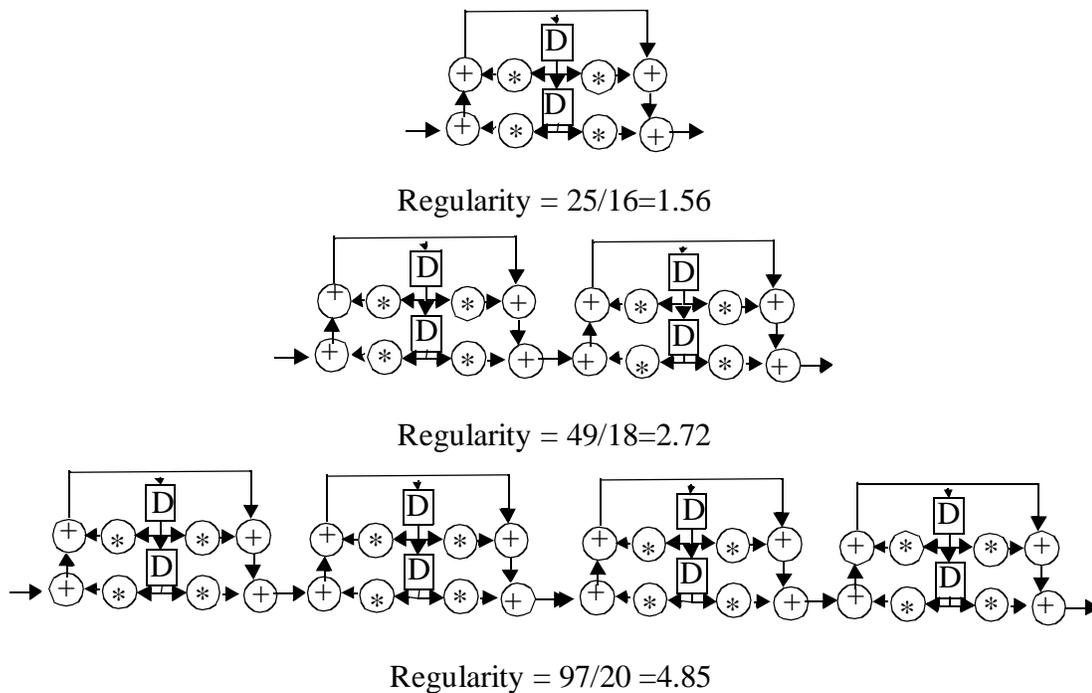


Fig. 26. Example of how regularity scales with size.

The Kolmogorov complexity is computer independent, except for an additive constant. Determining the Kolmogorov complexity of an arbitrary string is non-computable. This can be seen by realizing that finding the shortest program in general involves trying all short programs to find the shortest one that prints the string then halts. However, as the halting problem is non-computable, finding a program to compute the Kolmogorov complexity of an arbitrary string is also non-computable.

Kolmogorov complexity is of interest for this work only to the extent that it provides intuitive motivation for the development of a heuristic measure of graph complexity. The following quote from Cover and Thomas [Cov91] nicely sums this up: “One does not use the shortest computer program in practice because it may take infinitely long to find such a minimal program. But one can use very short, not necessarily minimal, programs in practice.

3.7.2.2 Heuristic Complexity of a Graph

Like a string, a graph is composed of operations from a finite symbol set. Unlike a string, the symbols within the graph can have an arbitrary number of neighbors, depending on the amount of fanout. As a graph has more degrees of freedom (represented by its interconnect structure) than a string, finding a complexity measure of a graph (in the spirit of Kolmogorov complexity) is at least non-computable. This has led to the development of the heuristic graph complexity measure presented in this section.

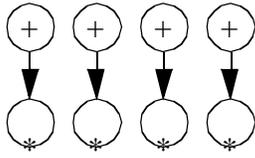
Given an encoding scheme (language by which to describe the graph and a measure of the length of a program), the complexity of a graph is defined as the length of the shortest program which can represent the graph. The three main components in determining the complexity of a graph are 1) a language by which to specify the graph, 2) quantification of the “length” of a program in this language, and 3) a technique to find a short program.

Encoding (Language and Length)

There are a number of programs that can describe a given graph, some being more compact than others. Consider Figure 27, which shows various template-mapped versions of the graph of Figure 27a; in each, a different number of patterns has been detected for use in its description. Description of each graph version involves describing the types and numbers of nodes, and the interconnection between them. A pseudo-descriptive language was developed for this purpose. Although the information is in the graph, the language is used as an aid to conceptualizing the descriptive complexity. The pseudo language has two primary instructions: *instantiate* and *connect*. There is an *instantiate* instruction for each unique type of operation primitive and template. The *instantiate* instruction takes two parameters: the primitive or template to instantiate, and the number of instances. There is a *connect* statement for each connection. The contents of each unique template is then itself described in the same way using *instantiate* and *connect* statements.

For each graph version in Figure 27, the descriptive program is shown. Consider the graph and corresponding description shown in Figure 27a. Instantiations of four additions and four multiplications are made, followed by four *connect* statements to describe their interconnection. In describing interconnections, the “L” and “R” postfixes specify the left and right input ports, respectively. In Figure 27b, instantiation of additions, multiplications, and an addition-multiplication is done. Three *connect* statements are made for connections between the primitive additions and multiplications. Then, a description of the addition-multiplication template is given.

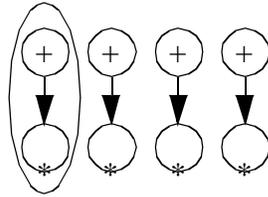
The length of a program in this pseudo-descriptive language is defined as the total number of *instantiate* and *connect* statements. This scheme was chosen through experimentation to be intuitively appealing and consistent.



```
.define graph
instantiate(add,4)
instantiate(multiply,4)

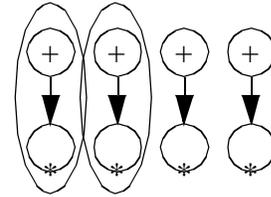
connect(a[1], m[1].L)
connect(a[2], m[2].L)
connect(a[3], m[3].L)
connect(a[4], m[4].L)
.end_define graph
```

(a) Length = 6



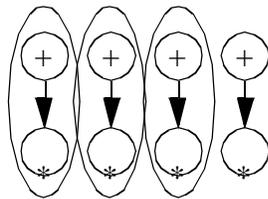
```
.define graph
instantiate(add,3)
instantiate(multiply,3)
connect(a[1], m[1].L)
connect(a[2], m[2].L)
connect(a[3], m[3].L)
instantiate(add_mult, 1)
.define add_mult
instantiate(add,1)
instantiate(multiply,1)
connect(a[i], m[j].L)
.end_define add_mult
.end_define graph
```

(b) Length = 9



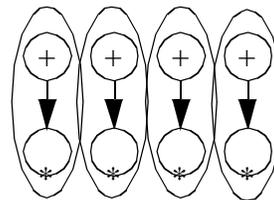
```
.define graph
instantiate(add,2)
instantiate(multiply,2)
connect(a[1], m[1].L)
connect(a[2], m[2].L)
instantiate(add_mult, 2)
.define add_mult
instantiate(add,1)
instantiate(multiply,1)
connect(a[i], m[j].L)
.end_define add_mult
.end_define graph
```

(c) Length = 8



```
.define graph
instantiate(add,1)
instantiate(multiply,1)
connect(a[1], m[1].L)
instantiate(add_mult, 3)
.define add_mult
instantiate(add,1)
instantiate(multiply,1)
connect(a[i], m[j].L)
.end_define add_mult
.end_define graph
```

(d) Length = 7



```
.define graph
instantiate(add_mult, 4)
.define add_mult
instantiate(add,1)
instantiate(multiply,1)
connect(a[i], m[j].L)
.end_define add_mult
.end_define graph
```

(e) **Length= 4**

Fig. 27. Example of language and program length.

For the example, the original length of the graph if no patterns are detected is 6 (Figure 27a). The length if only a single pattern is detected increases the cost to 9. For two and three patterns, the lengths are 8 and 7, respectively. Finding all of the patterns as shown in

Figure 27e results in the lowest length of 4. The heuristic complexity of the graph is the length of the shortest program found, which in this case is 4.

The example of Figure 28 emphasizes the fact that the majority of complexity in graph descriptions is due to the description of the interconnect structure. Both graphs of Figure 28a and 28b have the same types and numbers of nodes. The flat description of either graph is 10. The interconnect structure of Figure 28a is much more regular however, and

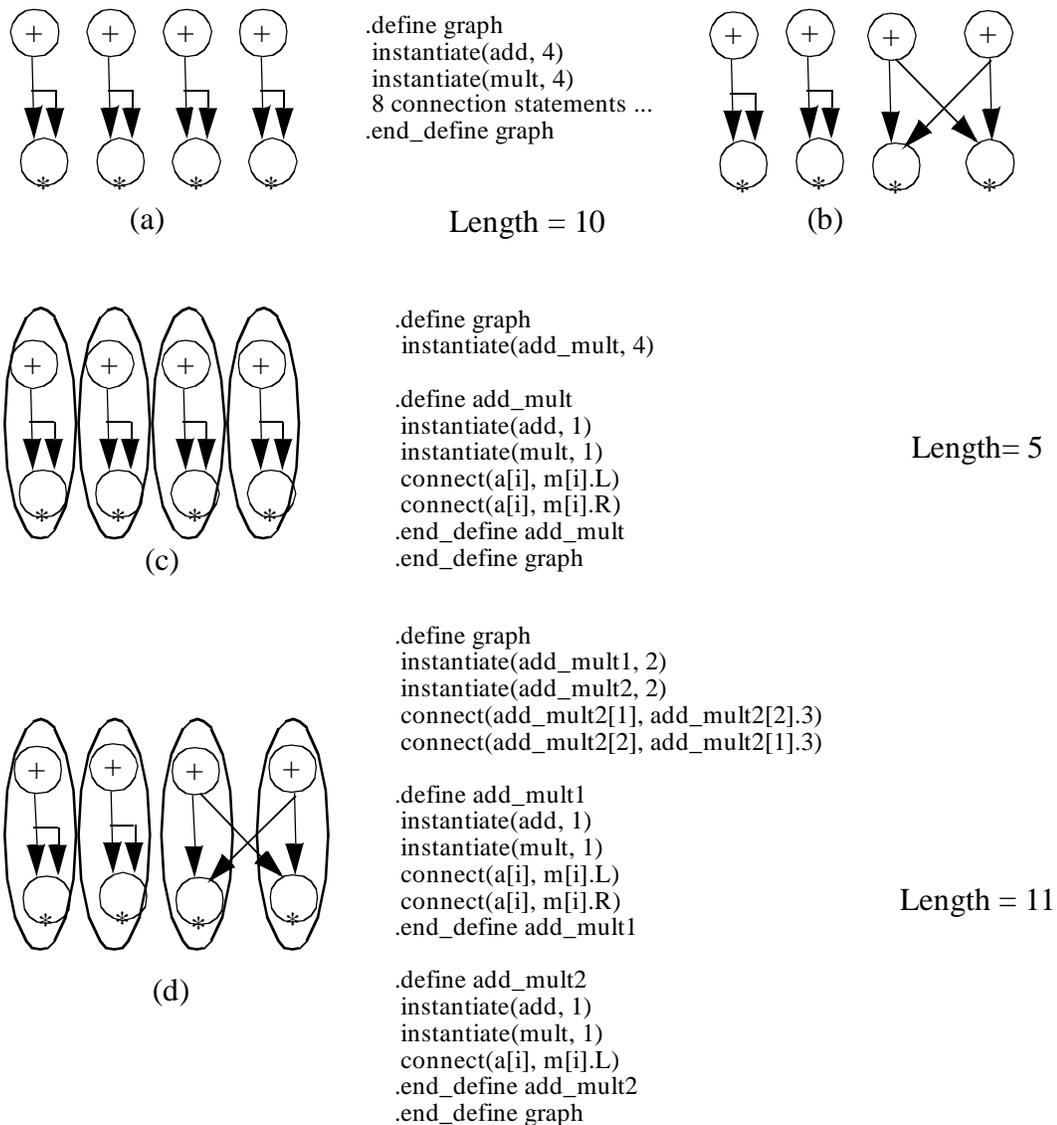


Fig. 28. Encoding example: majority of complexity is in interconnect structure.

thus allows the efficient description of the graph. The complexity of the graph of Figure 28a is 5, while that of 28b is 10. This results in regularity values of 3.2 and 1.6, respectively.

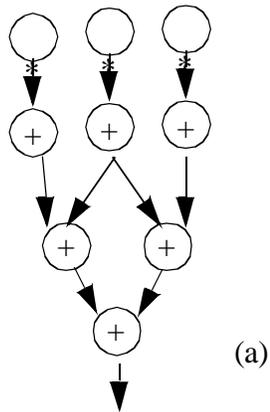
Another example is shown in Figure 29. The computation without patterns detected requires a description of length 11. Identification of the patterns in 29b and 29c do not result in reduced description lengths. From this example, it is clear that description length is only reduced when there are a significant number of common patterns.

Given a template-matched graph, the descriptive complexity is measured by the routine outlined in the following pseudo-code:

```
GetDescriptiveComplexity (Graph) {
  subgraphcomplexity = edgecount = operatorypes = 0;
  for each node n {
    if primitive (n) {
      if not seen before {
        operatorypes++;
        markseen(n);}
    else {
      if not seen before {
        subgraphcomplexity += GetDescriptiveComplexity (n);
        operatorypes++;
        markseen (n);}}
  }
  for each edge e {
    edge++;
  }
  encoding = operatorypes + edges + subgraph complexity;
  return (encoding);
}
```

Technique to Find the Graph Complexity

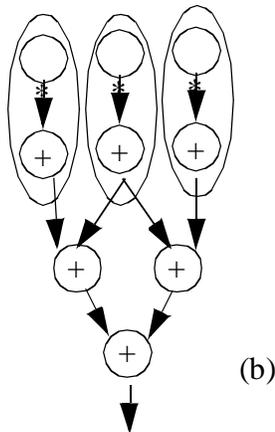
Detecting the common patterns that enable a concise description of a graph involves both template matching and template selection. Template matching locates the patterns



(a)

```
.define graph
instantiate(add,6)
instantiate(mult,3)
connect(m[1], a[1].L)
connect(m[2], a[2].L)
connect(m[3], a[3].L)
connect(a[1], a[4].L)
connect(a[2], a[4].R)
connect(a[2], a[5].L)
connect(a[3], a[5].R)
connect(a[4], a[6].L)
connect(a[5], a[6].R)
.end_define graph
```

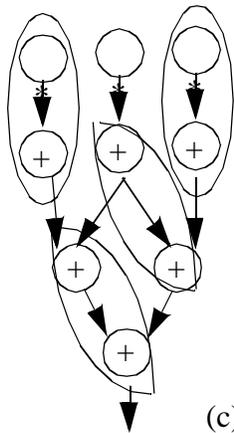
Length = 11



(b)

```
.define graph
instantiate(ma, 3)
instantiate(add,3)
connect(ma[1], a[4].L)
connect(ma[2], a[4].R)
connect(ma[2], a[5].L)
connect(ma[3], a[5].R)
connect(a[4], a[6].L)
connect(a[5], a[6].R)
.define ma
instantiate(mult,1)
instantiate(add,1)
connect(m[i], a[i].L)
.end_define ma
.end_define graph
```

Length = 11



(c)

```
.define graph
instantiate(ma, 2)
instantiate(aa, 2)
instantiate(mult, 1)
connect(ma[1], aa[2].1)
connect(aa[1].1, aa[2].2)
connect(aa[1].2, aa[2].3)
connect(ma[2], aa[1].2)
connect(m[1], aa[1].1)
.define ma
instantiate(mult,1)
instantiate(add,1)
connect(m[i], a[i].L)
.end_define ma
.define aa
instantiate(add,2)
connect(a[i-1], a[i].L)
.end_define aa
.end_define graph
```

Length = 13

Fig. 29. Encoding example: sufficient number of patterns is needed to reduce length.

that exist in a computation. In the computation of Figure 29a, for example, there are, among others, 3 mult-add patterns, 6 add-add patterns, and 4 add-add-add patterns. Template selection involves selecting a subset of these patterns to cover the computation. For example, in Figure 29c 2 mult-add and 2 add-add patterns were selected.

The template matching and selection is done hierarchically, in order to capture the regularity at different levels of granularity. At each hierarchy level, a mutually exclusive set of template matches is selected (no overlaps occur between selected template matches). Consider the example of Figure 30. If only a single level of matching and selection is done, where the 8 multiplication-additions or the 2 biquads are selected, then the resulting program lengths are 26 (Figure 30b) or 20 (Figure 30c). The hierarchical approach captures regularity at both levels, giving a program length of 18 (Figure 30d).

In particular, this work considers a 2-level hierarchy for template matching and selection. At the higher level of hierarchy, patterns are both identified and selected using the user-specified input description; the subroutines and loop structures defined in the description are used as the template patterns. This is done to leverage off of the designer's ability to identify "large-grain" regularity. At the lower level of hierarchy, the template matching routines of Corazao *et al.* [Cor93] are used to identify patterns. For this, the pre-defined library of templates enumerated in [Cor96] is used. These are a collection of common patterns (e.g. add-add, mult-add) that have been observed in many of the targeted applications (Section 2.1.1). For template selection, the routines of [Cor94] are used. These routines target area minimization, which has goals similar to encoding minimization.

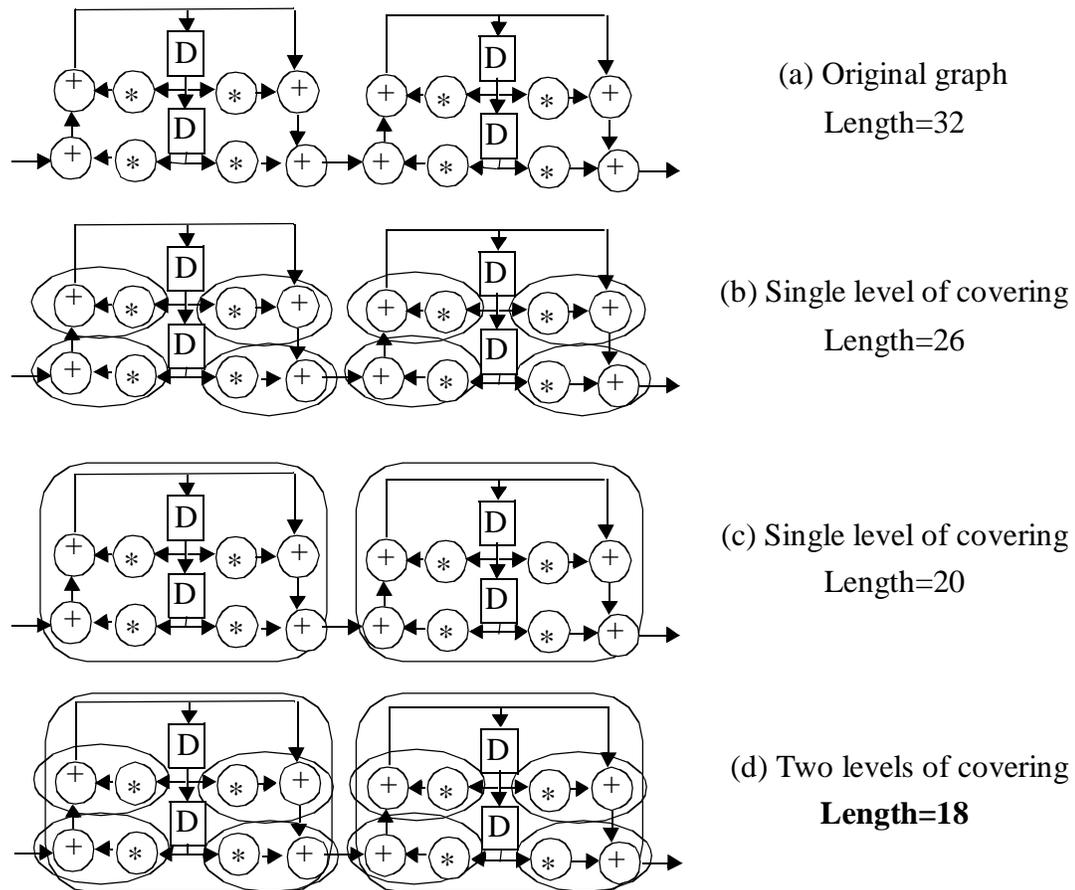


Fig. 30. Hierarchical template matching.

3.8 Other Property Metrics

There are a number of other important metrics of a design which do not fall into the above-mentioned classes. Note however that they are by no means any less important than the already presented metrics. The first metric is a classification of the computation as being either linear or non-linear. The non-linear computations are further classified as either feedback linear or not. Linear systems are a standard topic in fields such as control theory and digital signal processing [e.g., Kai80, Opp89]. Linear computations follow the principle of superposition, according to which linear combinations of inputs map into lin-

ear combinations of outputs. Linear systems are defined over a field F , the additive operator \oplus and the multiplicative operator \otimes . Typically, F is the field over the real numbers, \oplus is arithmetic addition, and \otimes is arithmetic multiplication. This need not be the case, however. For example, systems can exhibit linearity over *min* or *max* as the additive operator, and arithmetic addition as the multiplicative operator [Fet90].

The class of feedback linear computations [Pot92] is a more general class including not only all linear computations, but also a subset of non-linear computations. Feedback linear computations can be defined by the following equations:

$$s[n+1] = A \cdot s[n] + B \cdot x[n]$$

$$Y[n] = C \cdot s[n] + D \cdot x[n],$$

where $s[n]$ is a vector of feedback states, $x[n]$ is the vector of primary inputs, and $y[n]$ is the vector of primary outputs. A is a matrix whose entries are functions which do not depend on the feedback states $s[n]$. There is no restriction on the functions forming the entries of the matrices B , C , and D . Informally, all computations which do not have multiplications between variables which belong to feedback cycles fall within this class.

For computations that are non-linear but not feedback linear, another metric is defined for quantifying its degree of non-linearity. This metric, the formal degree, is the degree of the largest polynomial computed. For example, while a linear computation has a formal degree of 1, the computation $y = x^3 + x^2$ has a formal degree of 3.

Other metrics include a characterization of the computation's conditional structure and nesting of for-loops. For conditionals, the number and types of operations in the "if" and "else" bodies is measured. For for-loops, the number of levels of nesting is measured.

3.9 Summary

This section has presented an approach to design characterization. The idea is to extract a set of quantified design metrics that represent the design's "essence." The properties that have been identified include size, topology, timing, concurrency, uniformity, locality, and regularity. Metrics for quantifying these have been presented. The metrics form the basis for a number of applications such as estimation and design guidance. These applications will be presented in the following two chapters.

Applications of Design Characterization

4

The information captured by design characterization can be used in a number of different ways. This chapter demonstrates its use in algorithm selection, performance estimation, architectural synthesis, and guiding design optimization. These ideas will be introduced through a case study on various trade-offs in the design of a collection of filter structures (Section 4.1). Further examples of each of the first three applications will then be presented in each of the subsequent sections (Section 4.2 - 4.4). A detailed presentation of property-based, guided design optimization will be given in Chapter 5.

4.1 Case Study: Avenhaus Filter Structures

In order to illustrate various ways in which property metrics can aid in high-level design and optimization, consider the eighth-order bandpass filter structure first proposed by Avenhaus [Ave72], and later studied by Crochiere and Oppenheim [Cro75] and by Potkonjak and Rabaey [Pot93]. The candidate algorithms for the implementation of the filter include the direct form, cascade, parallel, continued fraction, ladder, and wave digital structures.

Given a desired signal-to-noise ratio, the required bitwidth for each of the algorithms can be derived either automatically or using fixed point simulation. The traditional means

of comparing various filter structures is to evaluate size metrics like the number of operations of each type and the required bitwidths (Table 1). This section demonstrates that these metrics have an important impact, but depending upon the situation, can very often be misleading as well. For emphasis, italics will be used to highlight the property metrics used in each situation.

Structure	Number of multiplications	Number of additions and subtractions	Total number of operations	Required bitwidth
Direct form	16	16	40	21
Cascade	13	16	34	12
Parallel	18	16	39	11
Continued fraction	18	16	35	23
Ladder	17	32	50	14
Wave digital	11	30	47	12

Table 1: Size metrics of the Avenhaus filter structures.

Size and Timing Metrics are Dominant Performance Indicators

The graph of Figure 31 shows the active area of semi-custom ASIC implementations of the filter structures, synthesized using the low-power library of the Hyper system for a range of sampling rates (without applying any algorithmic optimizations). The first point to notice is that the required *bitwidth* has a strong impact on the area of implementation. The main reason why the *bitwidth* is the dominant area parameter is that the multiplier area grows quadratically with the number of bits (while the delay grows linearly). The direct form and continued fraction form require 21 and 23 bits respectively, while all other designs require less than 14 bits. These two designs are correspondingly the most costly

designs, requiring up to 4.5 times larger area. At low rates, the bitwidths alone can be used as an indicator of the relative areas, since all structures have a single instance of each type of execution unit. This is not the case at all sample rates, though. While the parallel form is the most cost efficient design for many rates, between 1.1 and 1.7 MHz, the cascade structure is up to 25% cheaper. This can be attributed to the fact that the parallel structure has more *multiply operations*, and thus requires an additional multiplier at those speeds. Another point that can be clearly seen from Figure 31 is that each structure has a maximum sampling rate at which it can operate (without considering optimizations). The *critical path* of each structure aids in determining attainable sample rates.

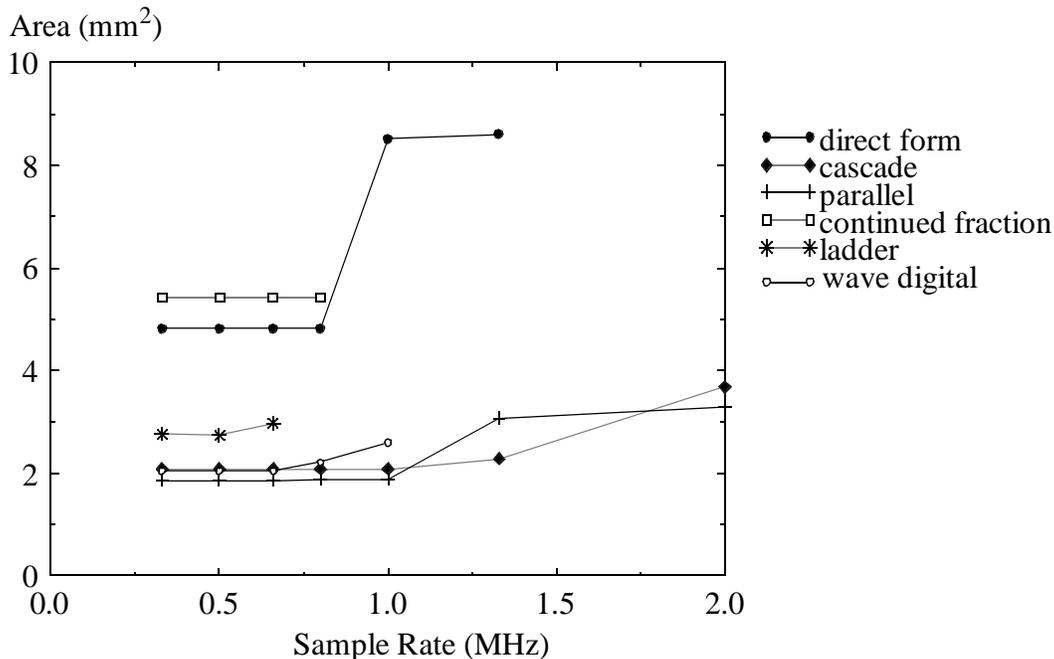


Fig. 31. Area versus sample rate for semi-custom ASIC implementations.

With Architectural Changes, Other Metrics Become Important Differentiators

This picture changes dramatically when implementing the same algorithms on an array of 24-bit processors, each equipped with a general purpose ALU data path (Figure 32). The bitwidths no longer aid in differentiating the designs. Instead, the *total number of*

operations and the *uniformity* become the key prediction parameters. The *total number of operations* includes the input-output operations and register-to-register transfers and are shown in Table 1. The ladder and wave digital forms, with the largest number of operations, require additional processors sooner, as the sample rate is increased. The cascade structure, with the least number of operations, is the cheapest at all sample rates. The continued fraction structure has the second smallest number of operations, but due to its *non-uniformity* and the resulting inability to efficiently utilize the available hardware resources, it requires a number of processors equivalent to the direct-form and greater than both cascade and parallel.

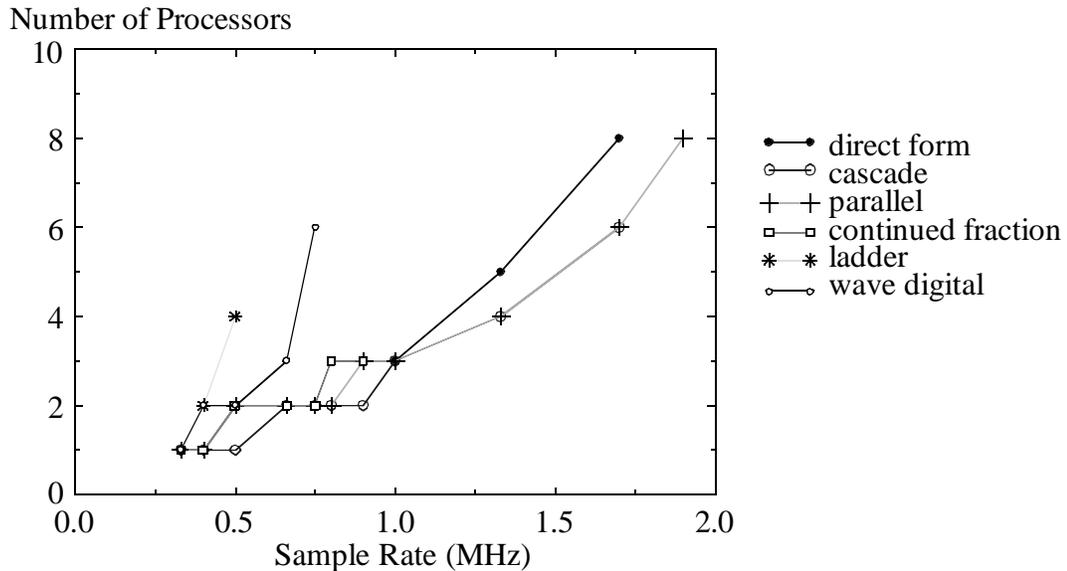


Fig. 32. Number of processors versus sample rate for 24-bit ALU implementations.

With Optimizations, Other Metrics Become Important Differentiators

The picture, sketched above, can change completely with the application of transformations. For example, a common transformation in a custom implementation of a fixed coefficient filter involves replacing the expensive multiplications with additions and

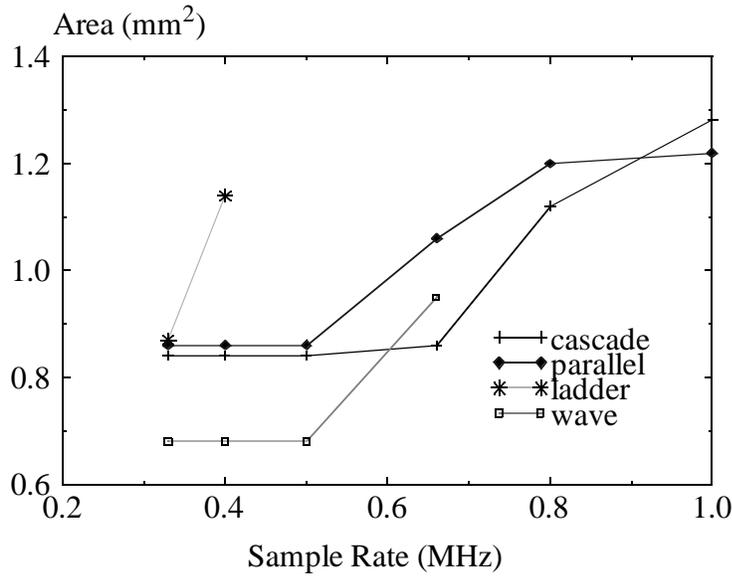


Fig. 33. Area for semi-custom ASIC implementations after constant multiplication expansion into additions and shifts.

shifts. This transformation results in a new scenario altogether. The size metrics, while important, will no longer be the primary differentiators of the designs.

Figure 33 shows the resulting active areas for the four most competitive designs. Notice that the wave digital structure has emerged as the smallest design at low sample rates. Without multipliers, the register area becomes significant, requiring an average of 65% of the active area, in these examples. In order to predict the register cost, a *variable concurrency* metric can be used. Figure 34 shows a comparison of the metric to the actual number of registers required. In this instance, the *maximumheight* metric of the midpoint variable lifetime distribution graph is used. This predictor correlates well with the actual register cost. The wave digital has clearly the least registers, and the lowest *variable concurrency* while the parallel has the highest of both.

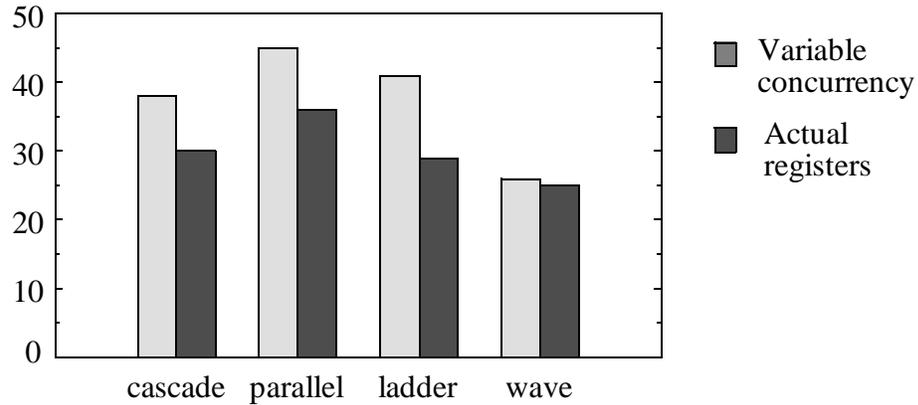


Fig. 34. Variable concurrency metric and number of required registers.

Metrics Can Also Help in Predicting Optimization Effectiveness

Predicting the effect of transformations is also important, as it is a key component of design space exploration. Several metrics can be used to guide this task.

When a high throughput design is desired, pipelining is a natural transformational choice. As some structures are more amenable to pipelining than others, it is useful to be able to foresee the potential impact. A good predictor of the potential improvement in throughput is given by the *ratio between iteration bound and critical path*. Table 2 shows the initial critical paths, the iteration bounds, and this ratio for each of the designs. From these metrics it is clear that pipelining will have the largest impact on the ladder structure, as its operation range is dramatically expanded.

Structure	Initial critical path	Iteration bound	Iteration bound/ critical path
Direct form	20	14	.70
Cascade	17	9	.53

Table 2: Predicting pipelining improvement using timing metrics.

Structure	Initial critical path	Iteration bound	Iteration bound/ critical path
Parallel	21	9	.43
Continued fraction	44	36	.81
Ladder	79	17	.22
Wave digital	50	43	.86

Table 2: Predicting pipelining improvement using timing metrics.

For the reduction of area requirements, retiming is a powerful transformation. Table 3 shows the improvement in execution unit area after its application. Recall that the *uniformity* of the operation distribution graph shows how well operations are distributed over time. In general, the greater the variance in the distribution graphs, the more non-uniform the structure is, and hence the more amenable to improvement through retiming. This correlation is clear in most cases.

Structure	Percentage decrease in EXU area	Variance of distribution graph
Direct form	52	5.45
Cascade	41	2.06
Parallel	33	1.48
Continued fraction	29	1.10
Ladder	0	0.30
Wave digital	42	0.29

Table 3: Predicting retiming improvement using uniformity.

This case study has illustrated several design characteristics that were related to the algorithms and their architecture mappings. Although this section has touched on just a few of the trade-offs in the design of the Avenhaus structures, one can appreciate the amount of exploration required even for such a relatively simple example.

4.2 Algorithm Selection

As was demonstrated in the Avenhaus case study, property metrics can serve as an advance prediction of implementation qualities, and can thus be used in guiding high level decisions. As was also shown, the metrics can be used to characterize and thus distinguish various algorithms. Algorithms with specific types of property metrics may map better onto certain architecture models. For example, a highly concurrent algorithm may map better onto an architecture with multiple parallel datapaths rather than a single sequential one.

4.3 Performance Estimation

This section presents several examples of how property metrics can be used in performance estimation.

4.3.1 Execution Unit Power Estimation: Size

Operation count, for example, can be used in estimating the execution unit power dissipation. An increased operation count not only affects EXU power, but also implies increased register and bus accesses due to the greater number of data transfers. In particular, as shown in [Meh94], the capacitance switched by the execution units is estimated by multiplying (over all types of operations) the number of times the operation is performed per sample period and the average capacitance per access of the operation. The total capacitance is hence given by:

$$C_{exu} = \sum_{i=1}^{numtypes} N_i \times C_i$$

where *numtypes* is the total number of operation types, N_i the number of times the operation of type i is performed per sample period, and C_i is the average capacitance per execution of operation type i . Sample capacitance models are shown in [Cha95].

4.3.2 Execution Unit Area Modeling: Operator and Register Access Concurrency

For estimating execution unit area, concurrency metrics can be used. An empirical study was conducted to detect the trends between actual numbers of units in implementation, and concurrency metrics. For each of 140 examples, the metrics were computed, then the number of units was determined using the Hyper system. The examples include various filters, transforms, sorting, elementary functions, and subsystems, as was described in Section 2.1.1. The specific metric with the highest correlation to the actual number of units is the maximum of operator *maximumheight'* and register access *maximumheight'*. The scatter plot of Figure 35 shows the strong correlation that exists between the two. The linear correlation is 0.92. A clear relation between the property metric and performance metric exists, and thus its use in synthesis or optimization is likely to be significant.

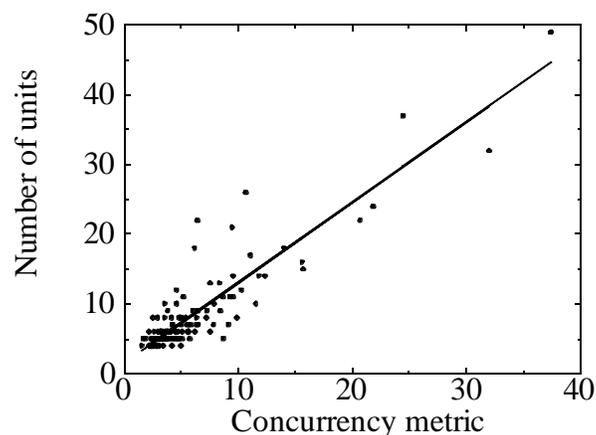
Due to the high correlation between the concurrency metric and the actual number of units, a natural next step involved using the metric for actual estimation. Using linear regression, a linear model based on the metric was built to predict the actual number of units. The model is:

$$\text{predicted number of units} = 1.15 \cdot \text{concurrency metric} + 1.55.$$

The model had an R^2 [Jai91] of 0.87 and an F-ratio [Box78] of 741.8. The R^2 is the ratio of the variability explained by the model over the variability of the data itself. If the model explains all the variability of data, the R^2 would be 1. On the other hand, if it explains

nothing, this value is 0. Clearly, a great deal of the variability of the data can be explained by the model. The F-ratio, another important criterion for evaluating the significance of the model, is the ratio of the mean square of the variation explained by the model over the mean square of the residuals. The F-ratio for the model is very high (just 2 degrees of freedom is used for the models), again indicating the excellent prediction potential of the model. While linear correlation, R^2 , and F-ratio are high, the root-mean squared (RMS) error as compared to the average is poor. The RMS error in predicting the number of units is 2.8 and the mean number of units is 8.6.

More accurate performance models will probably be a function of more complex groups of metrics. Several experiments were conducted to investigate the relationship between the error and several timing metrics, where the error is defined as (predicted number of units - actual number of units)/actual number of units. Timing metrics included the slack metrics, and the available time to critical path ratio. However, no dominant parameters could be determined.



Concurrency metric = $\max(\text{operator } \textit{maximumheight}, \text{register-write } \textit{maximumheight})$

Fig. 35. Scatter plot of number of required execution units versus concurrency metric.

4.3.3 Register Area Modeling: Variable Concurrency

An empirical study was conducted to detect the trends between actual numbers of registers in implementation and various metrics. For a set of examples, the metrics were computed, then the actual number of registers was determined using the Hyper system. The examples include various filters, transforms, sorting, elementary functions, and subsystems, as described in Section 2.1.1. The scatter plots of Figure 36 and Figure 37 clearly indicate a linear relation between the variable concurrency metrics and the number of registers in the final implementation. Linear regression has thus been used to build linear models to predict the number of registers based on the variable concurrency metrics. Table 4 shows the results of the statistical analysis and fit of the models, built using 40 examples. The linear correlation, R^2 , root mean square error, and F-ratio are shown for the models built using the midpoint, distributed, and split-distributed metrics. Results for models built using the Hyper min-bound and max-bound values on registers [Rab91b] are also shown for comparison. The mean number of registers over all examples was 49.6, while the root mean square error for the fitted models was only between 7.2 and 8.2. The

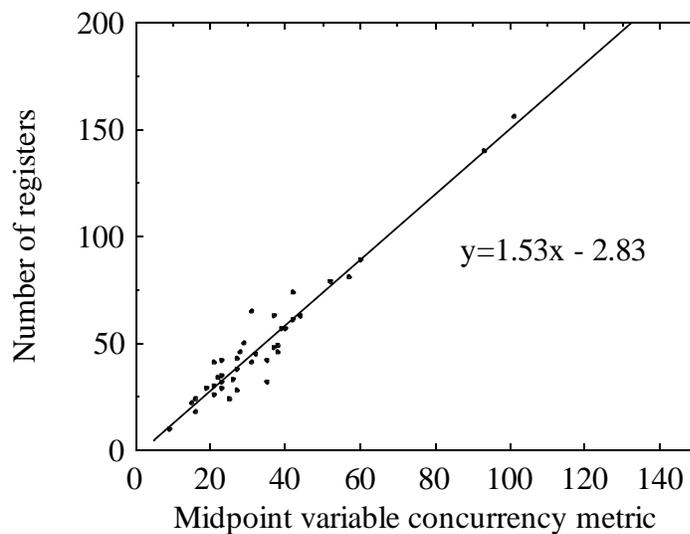


Fig. 36. Scatter plot of number of required registers versus midpoint variable concurrency metric.

number of registers can clearly be predicted by any one of the variable concurrency metrics. All models have R^2 values close to 1, and high F-ratios (just 2 degrees of freedom is used for the models), clearly indicating the excellent prediction abilities of the models.

	Midpoint	Distributed	Split-Distributed	Min-bound	Max-bound
Linear correlation	0.97	0.97	0.97	.92	.81
R^2	0.94	0.92	0.93	.84	.66
Root mean square error	7.2	8.2	8.0	12.4	18.3
F-ratio	598	454	483	178	63.2

Table 4: Summary of register models.

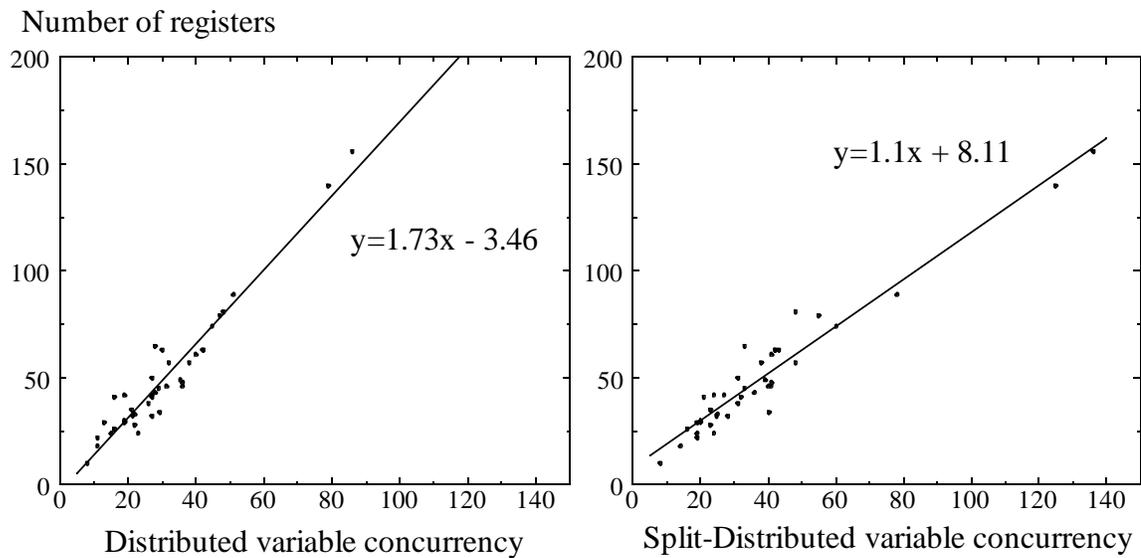


Fig. 37. Scatter plot of number of required registers versus distributed variable concurrency metrics.

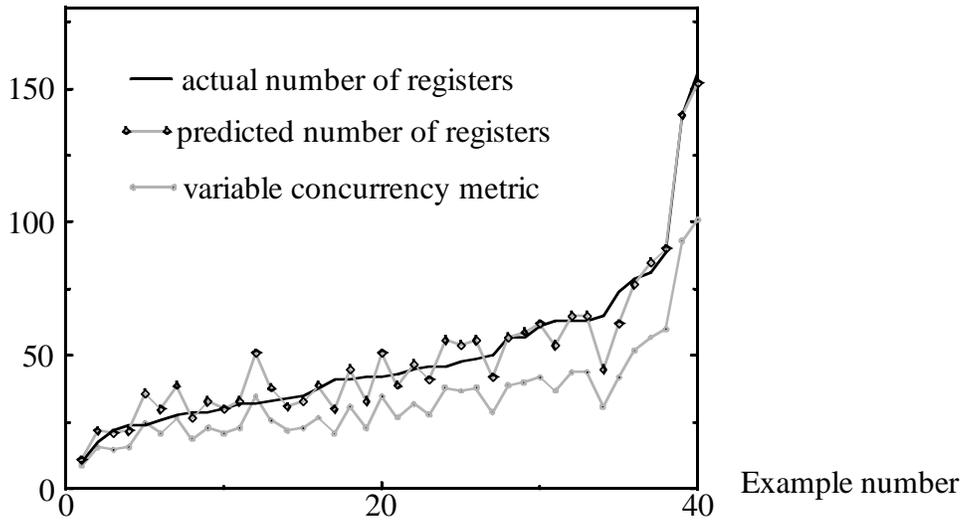


Fig. 38. Variable concurrency metric, model prediction, and required registers.

The plot of Figure 38 shows the actual number of registers and the model's predicted number of registers for forty examples. Also shown is the variable concurrency metric by the midpoint approximation. The strong predictive capability of the model is clear.

The empirical models were validated using the statistical method of analysis using learning and testing examples. This statistical validation technique involves randomly dividing all instances into *learning* and *testing* sets. During model development, only

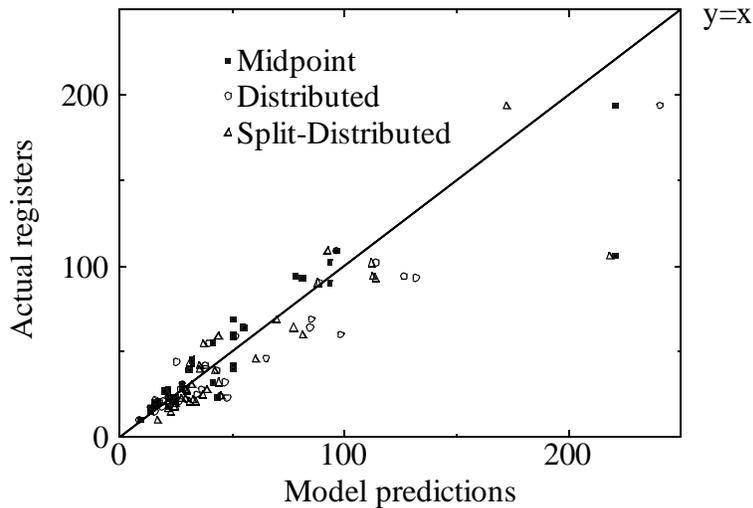


Fig. 39. Model validation using testing set.

examples from the learning set were used. The test set, consisting of forty new examples, was used to evaluate the developed models. If the model is accurate, then the predicted values will be close to the actual values. Figure 39 shows the actual number of registers versus the predicted values for the new examples.

4.3.4 Early Controller Area Estimation

Metrics can also be used in early controller area estimation. The controller can be modeled as a FSM plus control logic or a sequencer with microcoded ROM. The key elements in determining the controller area are the number of states and the width of the output control bus (signals to the registers, multiplexors, tri-state buffers, and execution units). Early estimation of controller size is therefore based on detecting trends in these two parameters. For the architecture model of Appendix A, a majority of the control lines are dedicated to the registers. The variable concurrency metrics can hence be used for correlation to the output bus width. Variable concurrency is related to the number of registers, as shown in Section 4.3.3. For estimation of the number of states, the critical path and regularity metrics can be used. Often times, the number of states is equal to the critical path in clock cycles. For graphs with loops or subroutines, however, regularity of the structure is reflected in the ability to share states. In this case a better prediction of the number of states is given by *critical path / regularity*, where the *critical path* is measured in clock cycles, and the regularity metric, *size/(descriptive complexity)*, defined in Section 3.7.1, is used.

Table 5 shows four examples. The first two examples have loops, and thus the critical path is normalized by the regularity to get a better prediction of the number of states. While the third example is functionally the same as the second, its loops have been unrolled. The synthesis tools are not able to identify the regularity, and thus as with non-

hierarchical graphs, the number of states is equal to the critical path. The last example also has no loops, and thus the number of states is equal to the critical path.

Structure	Loops?	Regularity	Critical path (clock cycles)	Critical path / regularity	Number of states
histogram	Yes	297	22000	74	47
cordic	Yes	14	100	7	5
cordic_unrolled	No	10	77	—	77
8th-order wave digital	No	1.7	25	—	25

Table 5: Examples of the relation between metrics and the number of states.

4.3.5 Relationship Between Regularity and Interconnect Structure

One area that regularity seems to be correlated to is the interconnect structure of the implementation. Namely, the more regular a graph, the less interconnect is expected. High regularity corresponds to simple description of the interconnections between operations in the flowgraphs, which translates into *fewer* types of interconnections and *more highly utilized* interconnections. Consider the two recursive filters of Figure 40. Interconnect

requirements (buses and multiplexors) that result from a hand-optimized assignment and scheduling are shown in Table 6.

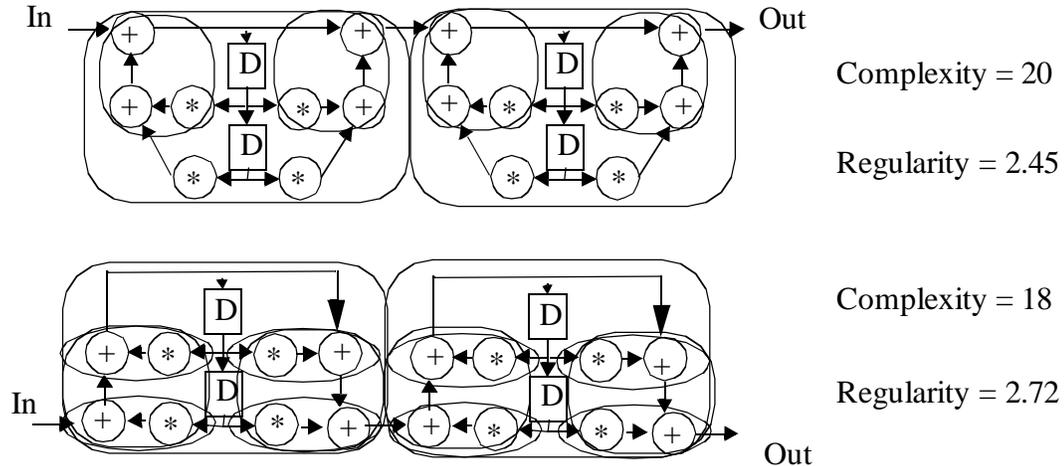


Fig. 40. Fourth-order IIR: a) less regular version b) regular version

In particular, the eight required interconnections for the less regular version are: transfer-> mult_left, mult -> add_left, mult -> add_right, add -> add_left, add -> add_right, add -> mult_left, input -> add_left, and add -> transfer. The 6 required interconnections for the regular version are: add -> transfer, transfer -> mult_left, mult -> add_left, add -> mult_left, input -> add_right, and add -> add_right. Note that a multiplexor is needed when more than one connection is made to an EXU' s input port.

	Less regular version	Regular version
Sample period (clock cycles)	10	10
Required EXU units	1 multiply 1 add 1 input-output, 1 transfer	1 multiply 1 add 1 input-output, 1 transfer
Number of busses	8	6

	Less regular version	Regular version
Sample period (clock cycles)	10	10
Number of muxes	4	2

Table 6: Regularity for interconnect area — hand-analyzed fourth-order IIR.

Automatic synthesis of these 2 structures results in the hardware requirements shown in Table 7. As the Hyper [Rab91a] tools do not exploit regularity, implementations of both versions require the same number of buses and multiplexors.

	Less regular version	Regular version
Required EXU units	1 multiply 1 add 1 input-output, 1 transfer	1 multiply 1 add 1 input-output, 1 transfer
Registers	18	17
Number of busses	8	8
Number of muxes	4	4

Table 7: Regularity for interconnect area — synthesized fourth-order IIR.

4.4 Architectural Synthesis

4.4.1 Assignment for Interconnect Reduction: Exploiting Regularity

As the results of Table 7 demonstrate, regularity of the structure may not result in simpler interconnect structure if the regularity of the structure is not exploited during synthesis. This thesis thus proposes a new assignment approach which explicitly exploits regularity. This work was presented in [Gue93, Gue94], and has since been continued collaboratively in [Rab95] and extended in [Meh96b]. The main ideas are illustrated in Figure 41. Assume that 4 units have been allocated: 2 adders and 2 multipliers. As there are 4

can be reduced, but this will come at the expense of other interconnect structure units such as additional multiplexors and tri-state buffers.

One approach to imposing the exploitation of regularity is through template mapping. Consider again the example of Figure 40. The less regular version can be implemented with the hardware shown in Table 8 column 2, as was previously mentioned. The use of complex units (corresponding to the complex instructions or templates) only increases the area, since the units cannot be used efficiently. The more regular version, however, can be implemented using either primitive or complex units with about the same active area. Simplifying the instruction set from two to one instruction by using the complex multiply_add unit, however, results in a much simpler interconnect structure. In a sense, the template matching explicitly creates a more regular interconnect structure. While the Hyper synthesis tools were not able to detect the regularity for the primitive version, the template selection made explicit the regularity, resulting in interconnect requirements similar to those attained for the hand-optimized version of Table 6.

	Less regular version	Regular version	
	Primitive	Primitive	Complex
Required EXU units	1 multiply 1 add 1 input-output, 1 transfer	1 multiply 1 add 1 input-output, 1 transfer	1 multiply_add 1 input-output, 1 transfer
Registers	18	17	16
Number of busses	8	8	5
Number of muxes	4	4	2

Table 8: Fourth-order IIR: imposing regularity exploitation through template mapping.

Consider again, the eighth-order direct form Avenhaus IIR filter structure (from Section 4.1), which is very regular and can be covered completely with multiply-add templates. The active area of the implementations with primitive units and with complex multiply-add units are approximately the same. However, the simplification of the instruction set from two to one instruction allows for a simpler interconnect structure as shown in Table 9.

	Primitive		Complex	
	.8MHz	1.3MHz	.8MHz	1.3MHz
Required hardware	1 multiply 1 add 1 input-output	2 multiply 2 add 1 input-output	1 multiply_add 1 input-output	2 multiply_add 1 input-output
Number of busses	10	18	8	13
Number of muxes	6	11	4	7

Table 9: Direct-form IIR: imposing regularity exploitation through template mapping.

4.5 Summary

This chapter has shown a number of ways in which the information captured by the design characterization can be used for algorithm selection, performance estimation, architectural synthesis, and for predicting the impact of optimizations. Chapter 5 will demonstrate how design characterization can be integrated into an interactive guidance environment.

Guided Design Exploration

5

Optimization has become a key aspect of the modern high-performance synthesis process. As was described in Section 2.4, due to the overwhelming number of optimization options currently available, design exploration is often conducted in a qualitative, ad-hoc manner.

In this chapter, a methodology for using design guidance to aid in the design exploration process is proposed. Guidance involves providing a combination of quantitative and qualitative feedback to aid the designer in making design decisions. Feedback may include suggesting optimization and synthesis techniques to apply, as well as their predicted impacts on performance. In particular, a methodology and environment for interactive computer-aided guidance for design optimization is presented.

5.1 Related Works

The related work is organized along the lines of design guidance, design planning, and optimization ordering. A design guidance system, the Clio Design Advice System, was proposed by Lopez, Jacome, and Director in 1992 [Lop92]. This system provides design advice consisting of *static* pieces of *qualitative* information about different design options. The advice database operates as the collective memory of the entire community of designers utilizing the framework. In contrast to the Clio system, the system proposed in this the-

sis provides *quantitative* design advice that is *dynamic* in that it is specific to the design at hand.

There have also been several related works in the area of “design planning.” The ADAM Design Planning System [Kna91], for example, is a knowledge-based planner that constructs a sequence of design synthesis tasks in response to a given set of specifications. Once a potential sequence has been developed, its effect is predicted, and then it may be executed. The ADAM system concentrates on the design flow of tools (e.g. module selection, allocation, control generation) to synthesize an RTL design. In this thesis, emphasis lies in optimization flow, which has fewer ordering restrictions, and thus is more difficult. Other research on frameworks, design methodology, and flow management include [Kle94, Har90].

There have also been a number of other approaches to optimization ordering. These, however, only address fixed sets of techniques. Peephole optimization [McK65], for example, is a simple and popular technique for combining transformations where the compiler considers only a limited section of code upon which it applies the available transformations one by one. Using pre-determined static scripts of transformations [e.g., Bra84] is another common approach which works well when the targeted example has similar characteristics to those used in experimentally developing the scripts. In the enumeration-based “generate and test” [Mas87] approach, all combinations of transformations are considered for a particular compilation, and the best one is selected using implicit enumeration search techniques. For linear loop transformations, several research groups specializing in optimizing compilers have used mathematical theory to develop efficient ordering schemes [Wol91]. Generic probabilistic techniques can often provide a good trade-off between design time and solution quality [Fox89]. Bottleneck identification and elimination approaches have been proposed for a limited set of transformations [Iqb93,

Hua96]. Finally, the idea of enabling and disabling transformations, again restricted to a specific set of transformations, has been explored in a number of compilation [Whi90] and behavioral level synthesis efforts [Pot92, Sri95a, Hua94].

5.2 Global Overview

Figure 42 depicts the proposed methodology for guided optimization. This methodology is based on the idea that to fully capitalize on design expertise, a guidance environment should facilitate design rather than simply providing full automation. In this framework, design involves constant and intensive interaction between the system and designer; the system provides design feedback, based on properties, which the designer uses to help make exploration decisions.

Design Flow

The designer starts by entering a design specification, constraints, and goals; for example, a C++ description of a DCT algorithm, the required sample rate, and the goal of power optimization could be specified. The exploration process is shown within the dashed box in Figure 42. The design guidance system analyzes the specified design and generates feedback. This feedback includes not only estimates (e.g., design’s power and throughput), but also suggestions for improving the design. The user interprets this information and directs the exploration by selecting a specific “most promising” direction from among the suggestions or by proposing other alternatives. This decision is communicated back to the design guidance system via a “directive” specifying the action to pursue. Exploration continues, with control alternating between the system and user. After exploration has been completed, the best design(s) can be selected from among all the design alternatives generated.

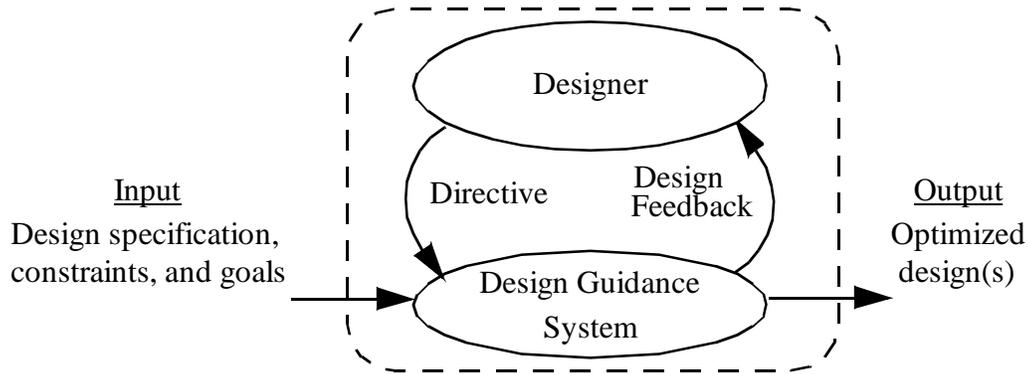


Fig. 42. Guided design space exploration.

Feedback

The critical component in the guided exploration methodology is the design feedback provided by the guidance system. The feedback consists of performance estimates and a ranked set of suggested optimizations. Additionally, for each suggested optimization, the predicted immediate, and possibly longer-term, effects on performance is provided. The estimates are necessary for comparing design alternatives. The design improvement suggestions are used to guide the exploration.

In order to assure that the designer maintains a global view, a graphical display of the exploration trace is provided. The display is a tree representation of all the optimization sequences that have been applied so far (Figure 43). Nodes in the tree (•) represent design versions and edges (→) represent optimizations. Each edge has a single input node, and may have a single output node. The input node represents the design version to be optimized, and the output node the new design version after application of the specified optimization. The root node of the tree represents the original design. The sequence of optimizations applied to generate a particular design version is given by the corresponding edges along the path from root to the node in question.

Design feedback is encapsulated within the exploration trace. Optimizations suggested for further exploration are represented by the leaf edges (edges without output nodes). Adjacent to each leaf edge is its ranking. Links are provided to the performance estimates for all of the design versions, as well as the predicted performance effects for any of the suggested optimizations.



Fig. 43. Abstracted representation of the exploration trace that is displayed for the user.

5.3 Methodology Features and Benefits

The proposed methodology has a number of distinct features and benefits. This section summarizes the key features, emphasizing the main advantages of each:

1. Suggests relevant optimizations, providing reasons for their application as well as predicted effects.

For a particular design, typically only a subset of the known techniques are applicable to its optimization. The designer is usually left to investigate all of the available options. In the proposed methodology, non-applicable optimizations are eliminated from consideration freeing the user of this task. Feedback is given regarding the predicted impact of each of the suggested options. This provides the designer with information by which to more quantitatively analyze the design options.

2. Results in a increased and more formal understanding of optimizations and their impact.

In the methodology, a great deal of optimization knowledge must be encapsulated. This forces the tool developers and users who characterize optimizations to go through the

educational exercise of formalizing and quantifying their impact. Furthermore, this serves as a valuable resource of information from which users of the system can learn.

3. Ranks suggested optimization alternatives.

The system provides a ranking of the proposed design options. The designer can use the ranking to further guide exploration; for example, the designer can at each stage select the optimization with highest ranking. The ranking takes into account performance constraints and optimization potential. The system keeps a global view of the design space, updating rankings as new information is obtained. It can thus suggest backtracking to previous design versions as soon as the current path loses promise.

4. Provides estimates of relevant design performance metrics.

Estimates are necessary to determine the performance of the various design versions. Note that this is distinct from point number one above which refers to predicting the impact that a potential optimization will have on this estimated performance. Behavioral level estimators use the behavioral description and the performance models from a hardware library to provide an estimate without a time consuming run through the synthesis process [e.g., Rab91b, Sha93]. Estimates are also used to provide accurate predictions of the impact of a sequence of optimizations.

5. Available automated optimizations are encapsulated and applied for the designer.

This frees the designer from having to interface with the various optimization routines. As a result, the designer need not worry about the details of the command line calls, parameters, and specific design files.

6. Encourages use of non-automated (manual) techniques which can be applied by re-writing the algorithm specification.

The methodology allows the use of techniques for which an automated routine is not available. The system may propose a non-automated technique which the user can apply manually by modifying the algorithm specification. Using the accompanying predictions, the designer can evaluate the time investment with respect to the potential improvement. The use of non-automated techniques greatly increases optimization power since it expands the library of techniques the designer is likely to apply from just the available ones to include all techniques that have been characterized in the environment. The available techniques are typically limited to include only the most widely used optimizations, due to costs of implementation and maintenance.

7. Provides a global view of the design space.

A global trace of the exploration is provided. This encourages and supports backtracking to any of the previous design versions, thus avoiding narrow searches of the design space.

8. Allows designers to override the system's suggestions.

This allows designer expertise to be integrated into the optimization process. The experienced designer is given the freedom to synergistically combine the information provided by the system with his/her intuition, specific design ideas, and common sense. Instead of selecting the option the system has ranked highest, the designer can select the one (s)he believes is best. Designers can also suggest and apply optimizations not proposed by the system or modify parameters with which an optimization is invoked.

9. Allows users to encapsulate new optimizations.

The system's optimization "knowledge" can continue to grow as designers encapsulate new optimizations. Users can thus benefit from the experience of others.

5.4 Design Guidance System

Figure 44 shows the main components of the design guidance system. Included are techniques for optimization, module selection, estimation, design characterization, evaluation of option effectiveness, and ranking. Further, the system uses a hardware library and a library of optimization characterizations. This section introduces these components, in order of access during guidance generation.

Optimization: The system applies the specified optimization to the specified design version. If the optimization is not available, the system prompts the designer to enter the new manually-optimized design version. (Note: when analyzing the initial design, this step is skipped).

Hardware Library: The hardware library contains performance information for all the available hardware resources. Performance information is given by parameterized models of area, delay, and average capacitance switched.

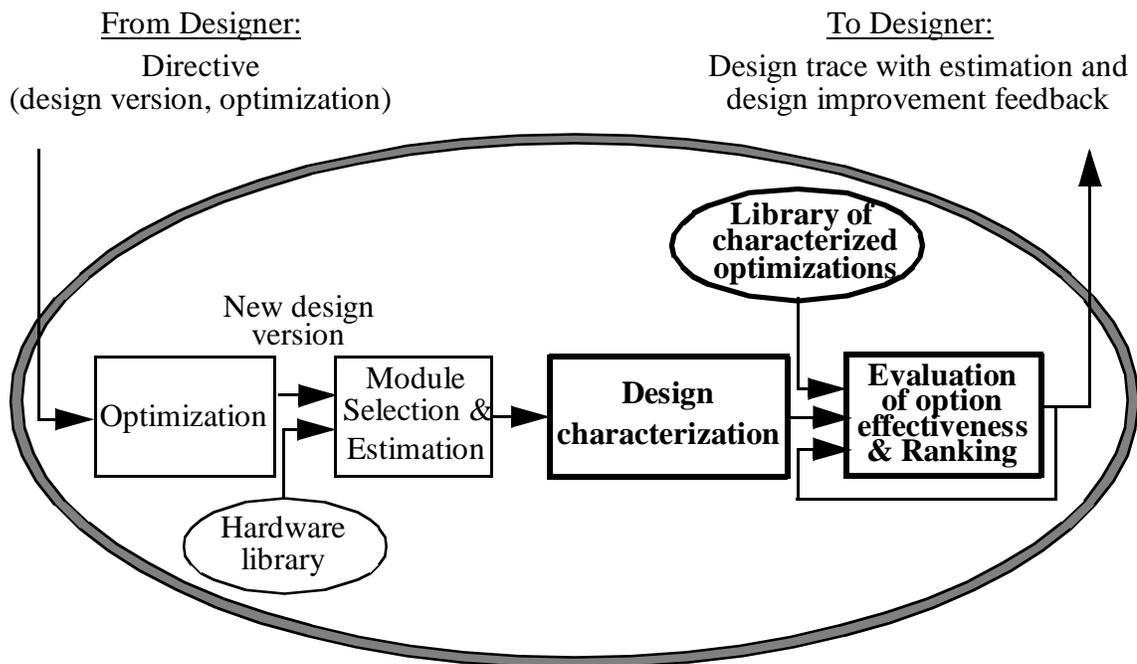


Fig. 44. Components of the design guidance system.

Module Selection and Estimation: During module selection, specific hardware resources are selected for implementation. Performance information for selected resources is annotated onto the design description. The design’s overall performance can then be estimated. Estimation gives immediate feedback regarding the impact of the applied optimization. The estimation results are provided as feedback to the designer. They are also needed in predicting the impact of subsequently proposed optimizations. For example, to predict the power dissipation that will result after voltage scaling (P_{new}), the current power estimate (P_{orig}) is directly used as shown in the following equation¹:

$$P_{new} = (P_{orig} \cdot V_{new}^2) / V_{orig}^2.$$

In the prototype system that has been developed, the behavioral estimators [Rab91b, Meh94, Gue94] are used for predictions of throughput, area, and power. These estimators are detailed in Section 2.3.

The following three parts of the system are the key new components. They are shown in Figure 44 in bold.

Design Characterization: The first component is design characterization, presented in detail in Chapter 3, to extract the essence of a design. The design essence is a characterization of the design using a set of properties relevant for providing design guidance.

Optimization Characterization: In addition to an understanding of the design, it is critical to have an understanding of optimizations that can be applied. The library of optimization characterizations encapsulates this expert knowledge. An overview of optimizations was presented in Section 2.4. An overview of the characterization process is given in Section 5.5 and specific optimization characterizations are presented in Appendix C.

1. The values V_{orig} and V_{new} are the original and scaled voltages, respectively. The prediction assumes that the capacitance switched stays constant.

Evaluation and Ranking: Using both the design and the optimization characterizations, predicted impact of the particular optimizations on the specific design in question is evaluated. Irrelevant options are pruned. The remaining options are ranked based on overall promise in optimizing design goals while meeting constraints. This ranking includes newly suggested options as well as the options that have been previously proposed but not yet applied. The updated design trace is then presented to the designer. The ranking approach used is presented in Section 5.6.

5.5 Optimization Characterization

A crucial step is the identification and characterization of relevant optimizations. Optimizations are pre-characterized and stored in the optimization library. During guidance, the information from the library is combined with the property metrics extracted during design characterization to predict the effectiveness of the optimizations on the specific design.

The first step in characterization involves identifying the optimization degrees of freedom. For example, degrees of freedom for power optimization include the supply voltage, activity factor, physical capacitance, and number of invocations for each resource. For throughput optimization, the degrees of freedom are the critical path in unit clock steps, the clock period, and the operator delays. Optimizations that target the degrees of freedom for the target objective are identified.

Each optimization is quantified by an encapsulated command-line call for its execution, a trigger condition, and as much information as possible about its immediate effect on design metrics, the potential it creates for subsequent optimization, the confidence metrics for all predictions, ordering with respect to other optimizations, and options and parameters to use in its application. Twenty optimizations have been characterized with

focus on their impact on throughput, and 22 optimizations have been characterized for power optimization under throughput constraints (Appendix C). Note that characterized optimization actions include both atomic synthesis techniques (e.g., clock selection, module selection, retiming) and static sequences of techniques (scripts).

Each optimization has an associated trigger. If an option's trigger is not satisfied (or if the optimization has just been applied), the option is eliminated from consideration, and time need not be wasted generating feedback regarding its effects. Design characteristics play a major role in trigger definition. Triggers can be any propositional logic function of the design's properties. Example triggers include "Is the computation linear?" (for optimizations that only work on linear computations), "Does the computation have sample delays?" (for the delay retiming optimization), and "Does the computation have multiplications by constants?" (for constant multiplication expansion into additions and shifts).

For options which are not pruned by their triggers, detailed guidance is generated. This guidance is presented textually to the designer as well as being used in option ranking (*see* Section 5.6). Knowledge regarding each optimization is encapsulated in the form of parameterized rules and equations. The parameters to these are the design's property metrics. Templates for generating textual feedback also contain property-based parameters. While the ranking uses only the quantitative feedback pertaining to immediate and potential effects on the cost functions, the textual feedback can additionally contain qualitative information regarding trends, ordering information, etc.

The most important feedback is the predicted impact on the design metrics. Immediate effects are characterized for each action. Enabling effects are characterized for the optimizations identified as typical enablers/enhancers of subsequent optimizations. Typical enablers include common sub-expression replication, one of the transformations used in Figure 8's example (Section 2.4). This technique replicates sub-expressions resulting in

added computation, but eliminates shared variables which would prevent the application of techniques such as the associative identity. Another technique with enabling qualities is logical partitioning. Logical partitioning is effective for managing computational complexity since it involves breaking the problem into smaller sub-pieces for subsequent optimization. It can enable the use of efficient techniques which, while not applicable to the entire computation, can be applied to individual subparts. Time-loop unfolding is another characteristic enabler. For enablers, predictions are performed regarding not only their immediate effects, but also their 1 or 2-step look-ahead predictions which would result when applied in conjunction with a few common transformations. The use of this information in optimization ordering is addressed in the section on ranking (Section 5.6).

In contrast to enablers, sometimes it is prudent to delay the application of a technique on a particular path. This type of information is not used in ranking, but is textually conveyed to the designer in the generated feedback. For example, it is often better to apply pipelining *after* constant multiplication expansion into additions and shifts (CM). This is because CM may dramatically change the lengths of paths. Clock selection is also often applied after CM due to the large difference in combinatorial delays between initial multiplier operators and resulting shift and add operators. In exploring paths that include CM, the designer can take this ordering guidance into account. Note that CM is an example of an optimization whose application in all practical situations executes as quickly as any reasonable predictor of its effects would. For such optimizations, the guidance environment bypasses prediction and simply applies the technique to determine its immediate effects.

Example: Time-Loop Unfolding

We end this section with an excerpt of the time-loop unfolding (TLU) optimization characterization for power optimization (Figure 45). Details such as bounds checking have

```

Name: Time-loop unfolding (TLU)
Command-line call: Manual
Trigger: "iteration bound < critical path || formal degree == 1 ||
         non-recursive || not all ops. in cycles"
Feedback:
  print ("TLU will reduce the critical path and enable voltage scaling.")
  print ("The following table shows the attainable voltages for various unfolding factors.")

  for (i = 1.. maxUnfoldingFactor) {
    if (non-recursive) {
      speedup = i+1
    } else if (linear) {
      speedup = (i+1) / [log2(NumStates + 1)] + 1      (with max-fast script)
    } else {
      speedup = critical path / iteration bound
    }
    T2 =  $\frac{3.9 \cdot \text{voltage}}{(\text{voltage} - 0.6)^2} \cdot \text{speedup}$        $b = -\left(\frac{3.9}{T_2} + 1.2\right)$ 
    newVoltage =  $0.5 \cdot \left(-b + \sqrt{b^2 - 1.44}\right)$ 
    print ($i, $newVoltage)
  }

```

Fig. 45. Excerpt from the time-loop unfolding characterization.

been omitted for clarity. From the command-line call entry, we observe that within our environment, an automated routine is not available for this technique. Furthermore, we observe a trigger which is a complex function of a number of the design's properties. The feedback shown indicates TLU's potential to enable voltage scaling for power reduction. It shows the generation of a table with the attainable voltages for various unfolding factors. Assume first a design that is non-recursive (an easily identified design property). The feedback given regarding TLU is the attainable speedup for an arbitrary unfolding factor i : $\text{speedup} = \text{old critical path} / \text{new critical path} = (i+1)$ [Mes88]. If the design is both recursive and linear (another property), the feedback conveys that the use of TLU in combination with the maximally-fast script [Pot92] will enable speedups of $(i+1) / [\log_2(\text{NumStates} + 1)] + 1$, where NumStates is the number of internal states to be

computed. For a non-linear recursive design, the feedback indicates that using TLU, the critical path can be reduced to the iteration bound (another easily computable property). Given the predicted speedup, a current voltage, and the empirical voltage-delay curve of [Cha95], we predict the reduced voltage (and although not shown, also reduced power) that can be used while still meeting timing constraints.

5.6 Ranking

In addition to suggesting and predicting the effect of feasible options, another key component of design guidance involves ranking the options. Options are ranked in order of their overall predicted effectiveness in meeting the specified constraint and optimization objectives. Ranking is done between all proposed optimization actions along all partially-explored optimization paths.

A proposed ranking scheme is presented below. There are three basic factors used to evaluate an action a_i 's effectiveness on design version d_j — the resulting performance as compared to existing solutions; its potential to enhance subsequent optimization; and the resulting feasibility in meeting constraints. For the goal of power minimization under a minimum throughput constraint T_c , measures of these three factors are defined below:

ImmediateImprovement_{ij} =

(lowest power so far - predicted power_{ij}) / lowest power so far.

if a_i is not an enabler

EnablingPotential_{ij} = 0.

else

EnablingPotential_{ij} = 1/k, where k is a_i 's rank among actions sorted in order of enabling potential on d_j .

if constraints are met (predicted throughput > T_c)

$$\text{Infeasibility}_{ij} = 0.$$

else

$$\text{Infeasibility}_{ij} = (\text{predicted critical path} / \text{available time}).$$

The immediate improvement function is a measure of how the resulting power after application of a_i to d_j will compare to the lowest power attained so far among all solutions. The one with the lowest predicted power will have the highest immediate improvement function. The enabling potential is 0 if a_i is not an enabler. Otherwise it is a function of the action's predicted enabling potential as compared to other actions. The infeasibility function is 0 if constraints are met, otherwise it is a measure of how far the current achievable throughput is from the minimum throughput constraint.

The basic effectiveness R_{ij} of action a_i on design version d_j can now be formed as a combination of these factors. The actions are ranked in decreasing order of R_{ij} :

$$R_{ij} = \alpha_1 t_{ij} \cdot \text{ImmediateImprovement}_{ij} + \frac{1}{t_{ij}} \text{EnablingPotential}_{ij} - (\alpha_2 t_{ij})^{\alpha_3 t_{ij}} \text{Infeasibility}_{ij}$$

The α_1 , α_2 , and α_3 are empirically derived constants. The value t_{ij} is an indicator of how far the application of action a_i is into the particular optimization trajectory:

$$t_{ij} = (1 + \text{length}(d_j)) / \text{total number of applicable actions} \quad 0 < t_{ij} \leq 1$$

It is defined as the ratio of the number of unique actions that have been applied along a given path to the total number of actions. The functions $1/t_{ij}$, $\alpha_1 t_{ij}$, and $(\alpha_2 t_{ij})^{\alpha_3 t_{ij}}$ are used to weight the sub-components of the ranking function as follows. In the beginning of an exploration trajectory, the actions with high enabling potential are favored. As the exploration progresses, the guidance favors more immediate improvements. Towards the end, ensuring feasibility becomes vitally important. Figure 46 shows how the weighting on the three components of the ranking function vary with t_{ij} (values of 4, 2, and 5 have been

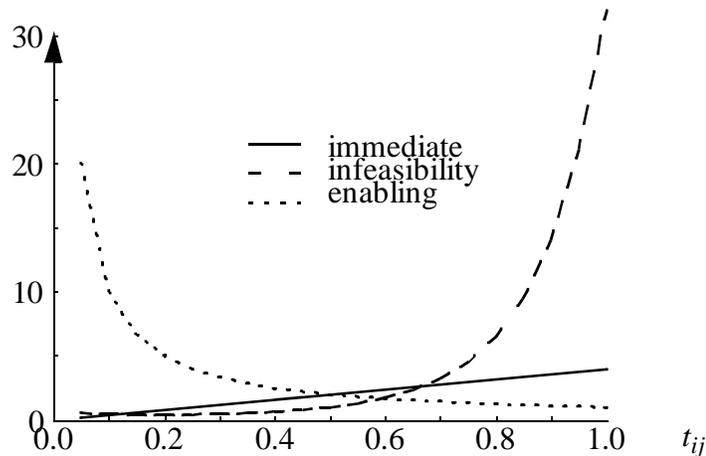


Fig. 46. Plot of the time-varying (t_{ij}) weighting on the immediate improvement, infeasibility, and enabling potential.

chosen for α_1 , α_2 , and α_3 , respectively, and the immediate improvement, enabling potential, and infeasibility are assumed equal). There may be a few special cases under which it is desirable to override an optimization's rank as given by the presented function. These special cases are specified in the optimization characterization. Furthermore, if a quantitative prediction of an optimization's effectiveness is not available, then it is not given a ranking. The designer can evaluate the qualitative feedback to decide whether or not to apply it.

5.7 Interface and Implementation

An environment has been developed to demonstrate the interactive guided exploration approach. The environment's graphical user interface is written using Tcl and Tk [Ous94], chosen since it allows rapid development of Motif-like interfaces. Embedded functions which perform the design analysis, optimization analysis, and ranking are implemented in C. The parameters passed to these functions include the design parameters and constraints, and pointers to the design (in flowgraph form), hardware library, and optimization charac-

terization library. All invocations of existing tools are made through the framework. The Tcl-Tk interface handles all the software system glue logic and management of exploration history.

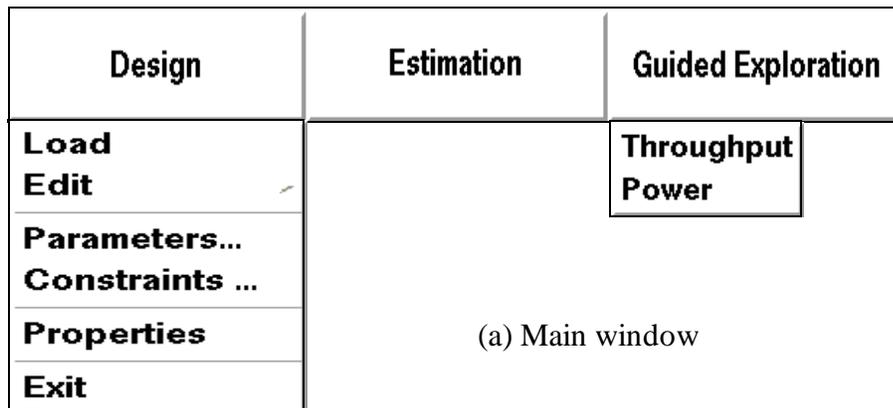
Figure 47a shows the main window of the guidance environment, from which design management, estimation, or guided exploration is invoked. Design management tasks include loading the design, editing the algorithm input description, or exiting the tool. The designer can also set parameters and constraints from the design management menu, or view the design's property metrics. Sample parameter and property windows are pictured in Figures 47b, and 47c, respectively.

An estimate of the design can then be requested. Hyper estimation for area, speed, and power is encapsulated, as well as property-based area estimation. Sample Hyper and property-based estimation windows are shown in Figures 48a and 48b, respectively.

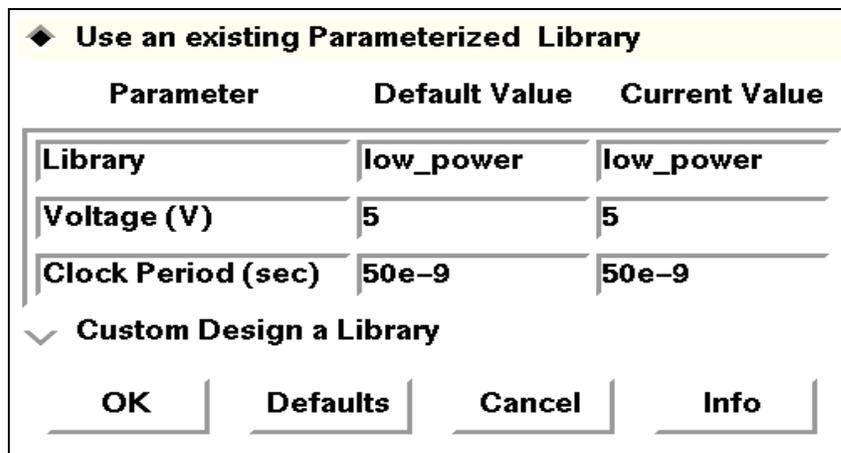
Guided design space exploration is invoked by depressing the guided exploration button, and selecting a desired performance objective. Many of the optimizations described in Appendix C have been encapsulated into the environment. Sample screen shots for guidance will be shown in Section 5.8.

5.8 Design Example

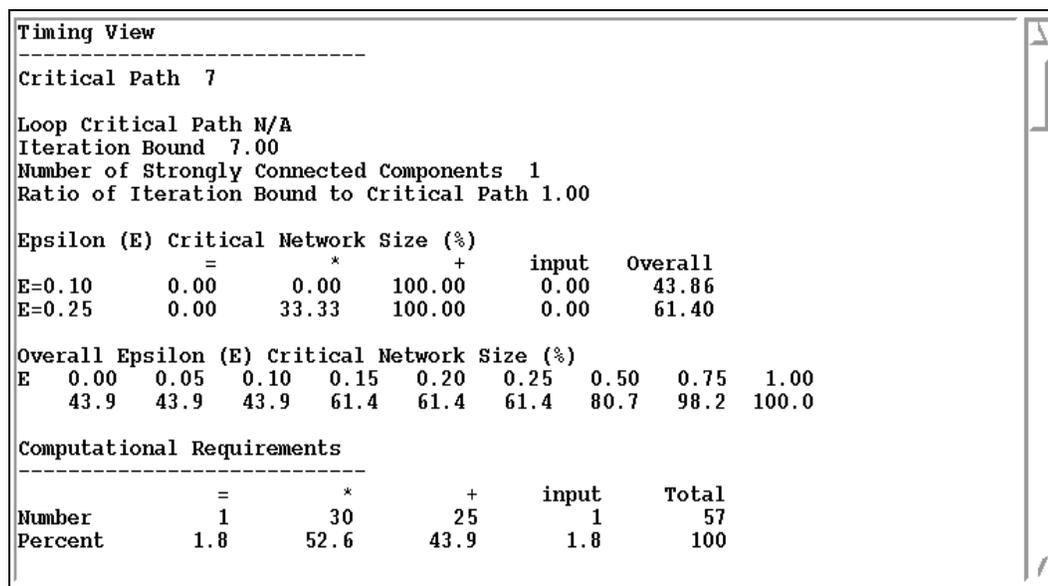
Probably the best way to illustrate the key ideas and the effectiveness of the new guidance methodology is by demonstrating it on a small, but real-life example. The goal is to provide a feel for the system, the types of feedback and guidance that it provides, and the underlying design methodology. This section presents the backbone and highlights of a sample session of an ASIC design of a General Electric (GE), state-space, 5-state, 32-bit linear controller [Cha93]. The Silage code of the initial specification is shown in Figure



(a) Main window

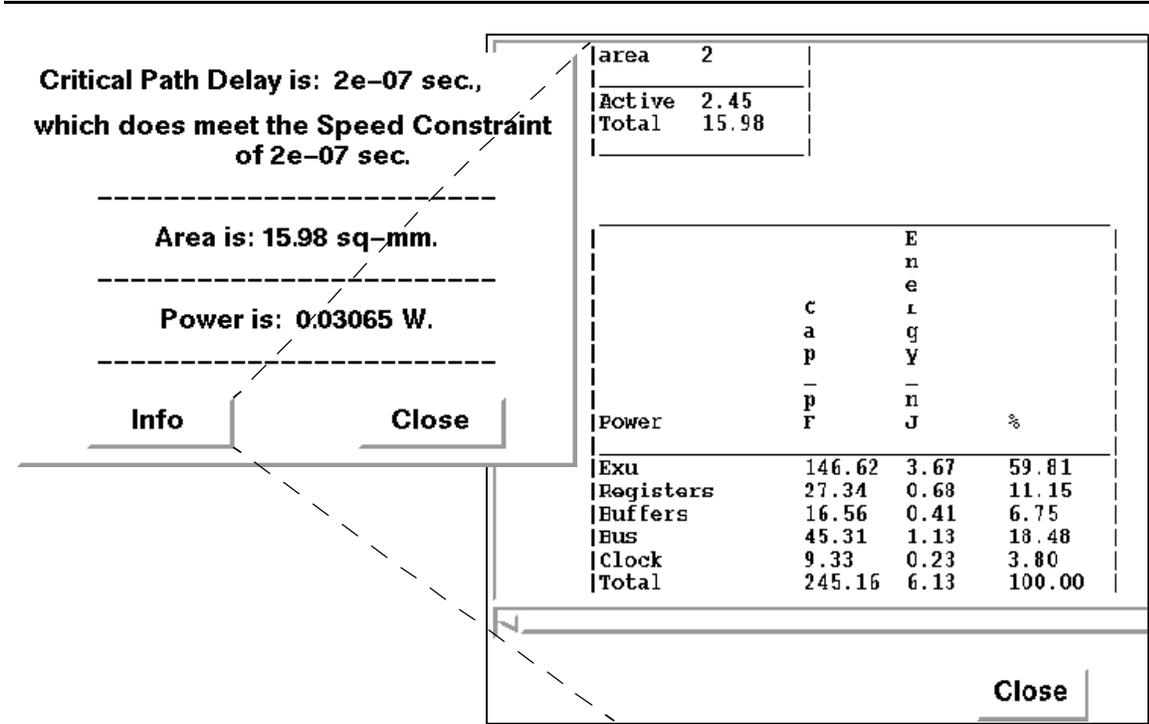


(b) Parameter window

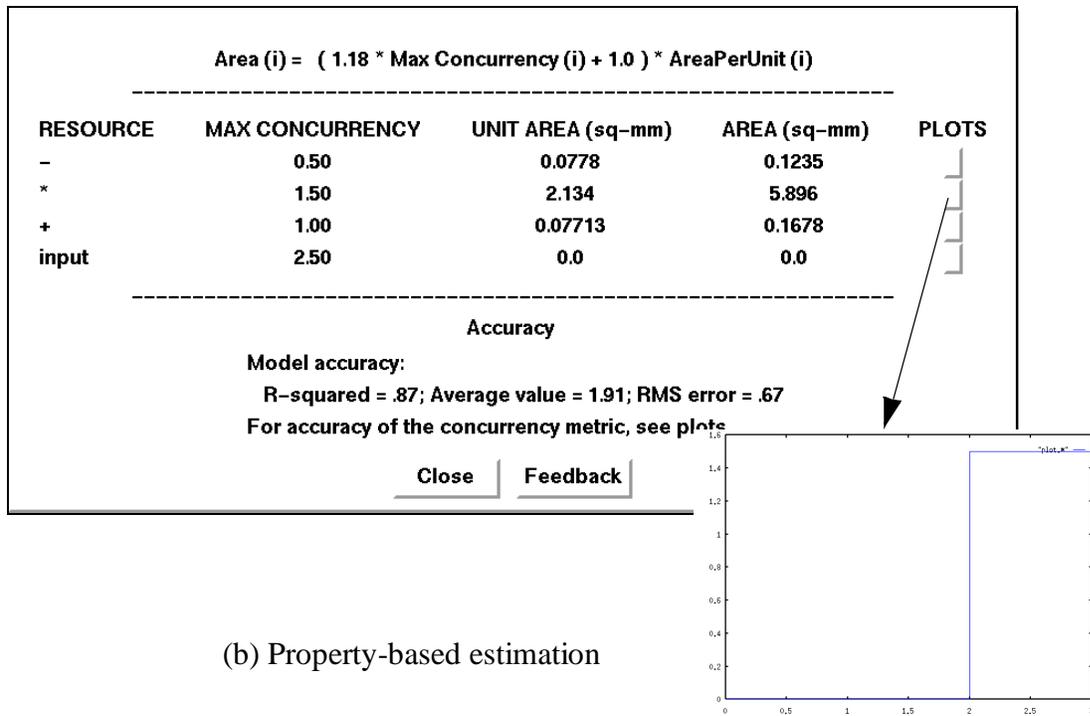


(c) Properties window

Fig. 47. Tcl-Tk based interface to the guidance environment.



(a) Hyper estimation



(b) Property-based estimation

Fig. 48. Tcl-Tk based interface: estimation windows.

```

#define num32 fix<32,10>
/* coefficient definitions omitted */
func main (U1: num32) Y1: num32 =
begin

    S1@@1 = num32(0.0);
    S2@@1 = num32(0.0);
    S3@@1 = num32(0.0);
    S4@@1 = num32(0.0);
    S5@@1 = num32(0.0);
    S1 = num32(A11*S1@1) + num32(A12*S2@1) + num32(A13*S3@1) +
        num32(A14*S4@1)+num32 (A15 * S5@1) + num32(B11*U1);
    S2 = num32 (A21*S1@1) + num32(A22*S2@1) + num32(A23*S3@1) +
        num32(A24*S4@1) + num32 (A25 * S5@1) + num32(B21*U1);
    S3 = num32 (A31*S1@1) + num32(A32*S2@1) + num32(A33*S3@1) +
        num32(A34*S4@1) + num32 (A35 * S5@1) + num32(B31*U1);
    S4 = num32 (A41*S1@1) + num32(A42*S2@1) + num32(A43*S3@1) +
        num32(A44*S4@1) + num32 (A45 * S5@1) + num32(B41*U1);
    S5 = num32 (A51*S1@1) + num32(A52*S2@1) + num32(A53*S3@1) +
        num32(A54*S4@1) + num32 (A55 * S5@1) +num32(B51*U1);
    Y1 = num32(f1*S1) + num32(f2*S2) + num32(f3*S3)
        + num32(f4*S4) + num32(f5*S5) + num32(f1*U1);

end;

```

Fig. 49. Silage code for the GE controller.

49. The design objective is power minimization under a user-specified throughput constraint.

Assume that the GE controller has a required sample rate of 1.2 MHz. If the initial specification is directly scheduled and assigned using the Hyper behavioral level synthesis tools and Berkeley Low-Power library, the resulting power using behavioral-level estimation tools [Meh94] is 441 mW at 5 volts using 1.2 μm technology.

The system prunes a number of optimizations which are not applicable. For example, library selection is not triggered since the low-power library has more effective modules with respect to power than the alternate dpp library [Bro92]. Module selection is not triggered since in the current library, there is just one module of each type. Clock selection is not suggested since it has already been set efficiently by the user. The user-specified description does not have common sub-expressions or for-loops so CSE and for-loop unfolding are not applicable. Finally, direct voltage scaling is not triggered since there is no difference between the critical path and the available time.

The guidance system proposes a number of actions that may result in immediate or subsequent power reduction. Figure 50 shows a snapshot of the design exploration session and the actions that were initially suggested — time-loop unfolding plus the max-fast script, time-loop unfolding, the max-fast script, pipelining, and replacement of constant multiplications with additions and shifts.

For each action, the guidance system provides predicted improvements, reasonings, and the corresponding analysis information. Information regarding pipelining, for example, includes the critical path improvement; in this case, it can reduce the critical path from 12 clock cycles to the iteration bound of 6 clock cycles. A power prediction is also pro-

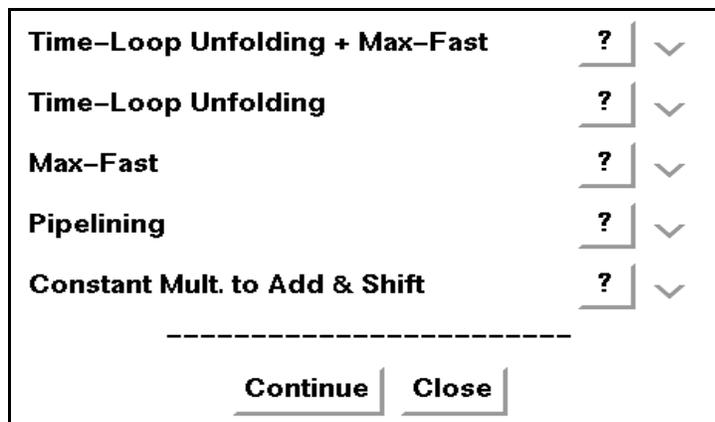


Fig. 50. Initial view of the exploration session.

vided based on an empirical voltage-delay curve [Cha95]; in this case, the critical path reduction has enabled a voltage reduction to 2.8 volts, giving an estimated power reduction of 3.2. Retiming is also applicable, but in this case cannot improve the critical path. For both pipelining and retiming, the new critical path is obtained using a simplified version of the transformation which does not perform the slow initial value recomputation (which is needed if the transformation is actually applied). Constant multiplication expansion into additions and shifts is another potential optimization. There are many constant multiplications: 36 out of 67 total operations. Furthermore, their bitwidth of 32 results in a large difference between multiplier and adder power dissipation, delay, and area.

The maximally fast script is an attractive alternative since it can reduce the critical path to at least 4. The maximally fast static transformation script combines several algebraic and redundancy manipulation transformations for critical path and operation reduction. A closed form equation exists for computing the estimated critical path, and is given in Appendix C.

While each of the mentioned options will result in immediate power reduction, the most promising optimizations are the script consisting of time-loop unfolding (tlu) followed by the maximally-fast script [Pot92], and the time-loop unfolding optimization alone. Time loop unfolding reduces the critical path to the iteration bound of the design, but is favored not so much for its immediate benefit (which is actually the same as pipelining's and inferior to the maximally fast script), but rather for the long-term improvement potential that it provides for significantly greater subsequent optimization. This potential is indicated by the computation's formal degree of 1 and relatively high ratio of states to the number of inputs and outputs. Theoretical analysis indicates that TLU does not enable subsequent optimizations as effectively for computations with higher formal

degrees and computations which have low ratios of number of states to number of inputs and outputs [Sri95a].

The time loop unfolding plus maximally fast script is an exceptionally effective combination. In particular, unfolding plus maximally-fast is selected because the computation is linear, and the knowledge stored in the optimization library indicates that it has the potential to reduce the number of operations, while reducing the critical path as much as needed to explore the potential of voltage scaling. The specific unfolding factor is selected by the designer. The system provides a table of suggested voltage and number of operations for various unfolding factors. The suggested voltage is based on maintaining some slack between the resulting critical path after voltage scaling and the available time (a slack of 10% has been used in this case). Based on this table and qualitative feedback that

Unfolding factor	New critical path	Speedup	Suggested voltage	Number of addition and multiplication operations
0	4	3	2.2	66
1	2	6	1.6	44.5
2	1.3	9	1.4	38
3	1	12	1.4	35.25
4	.8	15	1.3	34
5	.67	18	1.2	33.5
6	.57	21	1.2	33.43
7	.5	24	1.1	33.63

Table 10: GE controller example — the impact on attainable voltage and operation counts for various unfolding factors.

control overhead increases with greater unfolding, the designer selects an unfolding factor of six to balance achieving both speedup for voltage scaling and reduction in capacitance. This step reduces both the effective critical path by a factor of 21 and also reduces the effective number of operations (the number of operations per iteration) by a factor of 2. This in turn enables application of the MCM technique [Pot94b] for conversion of constant multiplications to shared series of shifts and additions, reducing the effective capacitance. The final steps are selected with guidance from the environment so that immediate improvements are maximized. The system, for example, has detected that further speedup will not aid in attaining better voltage scaling. As a result, all optimizations that improve throughput result in marginal power improvements at best. The last few optimizations involve lowering the voltage from 5 to 1 volts and performing clock selection for efficient utilization of the clock period [Cor93]. The optimized design has a power of 2.9 mW. For this example, it turns out that not only is power reduced by a factor of 151, but area is also reduced by a factor of 2.

Figure 51 shows another snapshot of the design exploration session. The left-most justified actions are those that were initially suggested — time-loop unfolding plus the max-fast script, time-loop unfolding, the max-fast script, pipelining, and replacement of constant multiplications with additions and shifts. Application of time-loop unfolding with the max-fast script has already been done through selection of the appropriate radio button and the “Continue” button. The estimated performance of the new version is displayed by selecting the “EST” button. Several optimizations are suggested for application to the new unfolded design version — replacement of constant multiplications with additions and shifts, voltage scaling, and common sub-expression elimination. The specific feedback regarding a suggested action is provided by selecting the corresponding information button (marked with a “?”). The information button for voltage scaling is displayed. Predicted

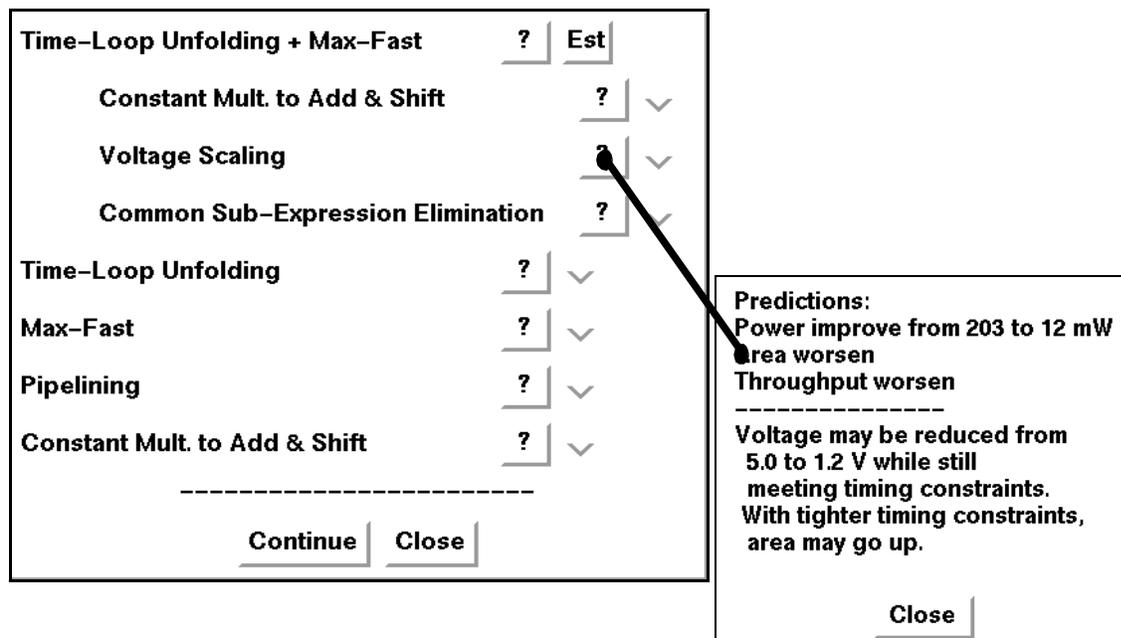


Fig. 51. View of the exploration session after application of time-loop unfolding and max-fast.

effects on power, throughput, and area are shown. Also, the system has predicted and suggested the amount of scaling possible while still meeting timing constraints.

This example has illustrated interactive guided exploration, and the importance of exploiting actions that increase the potential for improvement, as opposed to simply bringing the greatest immediate benefit. This example also emphasizes the importance of coordinating the optimization of multiple sub-goals. For example, the reduction of power using voltage scaling alone, gives at most a factor of 25 power improvement, if voltage is reduced from 5 to 1 volts. In the GE controller example, proper simultaneous consideration of techniques for reduction of effective capacitance along with critical path reduction techniques, provided the other factor of 6 reduction. The greatest gains are clearly realized by simultaneous attention to reduction of all sub-goals.

5.9 Summary

This chapter has presented the methodology for guided design space exploration. A key aspect presented was the bidirectional transfer of information between the system and designer. The system provides automated design analysis and advice regarding design exploration. Designers retain control and the flexibility to integrate their expertise. The chapter also introduced the main components of the guidance system.

Conclusion

6

This chapter summarizes the principle ideas and contributions of this thesis. As with most research efforts, the attempt to address one set of challenging problems has given rise to a whole new set of problems to be solved. This chapter starts by discussing some of these issues, presenting directions for future research. This is followed by the summary of this thesis.

6.1 Directions for Future Research

6.1.1 Improvements to the Guided Optimization Environment

There are a number of extensions that could contribute towards a more powerful guided optimization environment.

Extended optimization knowledge base: Additional optimization techniques targeting not only speed, area, and power but also other metrics such as testability can be studied and their effects characterized. To encourage users to integrate their design knowledge, a mechanism to allow easy entry of optimization characterizations would be useful.

Combined interactive and automated design exploration mechanisms: This thesis has promoted the benefits of an interactive guided exploration system. However, even more powerful is a hybrid system that allows automated exploration in addition to user-driven

interactive exploration. While retaining all the benefits of the interactive system, the hybrid system can additionally perform guided *automated* searches. Since they do not require constant user input, automated searches can be run in the background or overnight, thus exploring a good deal of the design space. However, it is important to note that they are most often limited to automated optimizations.

System reuse: Approaches to capture and reuse knowledge gained through design exploration are important. For example, effective sequences of optimizations may be saved in a library of static optimization scripts. As another example, saved experiences with specific designs can be used as reference for future designs. These experiences could include the various design versions (each with a different speed-cost-power trade-off) generated during exploration.

System learning: Each new design that is run provides new data for refining estimation models, correlations between metrics and performance, triggers, or optimization predictions.

Partitioning: Logical partitioning can be used to divide a computation into smaller, more manageable parts. Each computation subpart can then be optimized in accordance with its property metrics. This enables the use of techniques which, while not applicable to the entire computation, can be applied to individual subparts. A great deal of research remains to be done in exploring various partitioning schemes. The partitioning scheme proposed in [Gue96] for throughput optimization *isolates* subparts in addition to dividing them. As a result, the overall critical path lies within the parts, and independent optimization of subparts is enabled.

6.1.2 Design Characterization in Other Domains

One of the key questions is the portability of the ideas presented in this thesis to new application areas, architecture models, flowgraph representations, and models of computation. A change in any one of these would require a new design characterization where some metrics are unchanged, some are modified, and some are altogether new. It would also require establishing the relations between these metrics and performance. The hope is that with each new domain addressed, the number of new metrics needed will decrease, and at some point, a core superset of metrics will be established.

6.1.3 New Applications of Design Characterization

There is also potential for using design characterization in new applications such as architecture selection or programmable-processor design. For example, metrics could be used to indicate which architecture would be most suitable for a particular algorithm. For programmable-processor design, metrics can be used to identify common features among a set of applications which are to be mapped onto the processor. The hardware can then be designed to best exploit these features.

6.2 Summary

This thesis has presented techniques for improving the high level design and optimization process. As one of its principle contributions, techniques for design characterization using key property metrics of a design were presented. Property metrics most directly related to algorithm-architecture mappings were identified. In particular, metrics were presented for quantification of a design's size, topology, timing, concurrency, uniformity, locality, and regularity. While some of the metrics were defined previously, such as the critical path or the iteration bound, others were presented for the first time in this thesis, such as the concurrency and regularity metrics.

Information such as design characterization, that captures the relationships between algorithms and architectures can be used in a number of different ways. The identification of correlations between property and implementation metrics can already serve as an “advance prediction” to guide high level decisions such as selection of an algorithm, as was demonstrated in the Avenhaus case study. This information can also be used to build statistical models which use the property metrics as input parameters and give more accurate performance estimates of the implementation efficiency. In particular, it was found that metrics of operator and variable concurrency of the computations have strong correlations to the area of execution units and registers, respectively, on semi-custom implementations designed using Hyper. In the case of variable concurrency, it was possible to develop very accurate register area models.

Another important application of the design characteristics is their potential to improve current synthesis tools by providing more accurate and complete objective functions. Property metrics corresponding to the various aspects of the final implementation (execution units, register, interconnect, etc.) can be used to drive synthesis tasks. In particular, it was found that current Hyper tools did not exploit algorithm regularity, and thus resulted in implementations with more complex interconnect structure than expected. Techniques to avoid this through exploitation of regularity during the assignment step were demonstrated.

The final presented application of design characterization was design guidance. The methodology and techniques for providing design guidance are one of the primary contributions of this thesis. Rather than attempting to completely automate the high-level design and optimization process, the use of design guidance was proposed for *facilitation* of the process. A methodology and environment which provides interactive design guidance for optimization was presented. The core mechanisms needed in order to provide design guid-

ance were shown to be not only the design characterization, but also the optimization characterization. For the latter, a database of encapsulated knowledge about the various optimizations is used.

The presented design guidance methodology is based on suggesting and ranking potential optimizations for improving the design. Optimizations that are not applicable to a given scenario are pruned. For this, the use of pre-defined trigger conditions was proposed. For all remaining optimizations, evaluations of the likely effect each will have on the cost metrics is done. A scheme for ranking optimizations was also presented. The ranking takes into consideration the optimization's predicted effect on the cost function, its predicted potential for longer term benefit, and its predicted effect on the constraints.

Using guidance, designers maintain a more global view of the exploration space, can make decisions in a more quantitative and informed manner, and thus can more easily and quickly discover effective trajectories of optimizations. Consequently, higher quality designs and shorter design times can result. With the continued increase in design complexity, and likewise in the number of options the designer is faced with, design guidance methodologies and tools are likely to be of ever increasing interest to the designers of current and future-day systems.

Bibliography

7

- [Aho77] A. Aho and J. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Co., Reading, MA, 1977.
- [All75] F. Allen, "Bibliography on program optimization," Research Report RC-5767, IBM T. J. Watson Research Center, 1975.
- [Ave72] E. Avenhaus, "On the design of digital filters with coefficients of limited word length," *IEEE Transactions on Audio and Electroacoustics*, Vol. 20, pp. 206-212, 1972.
- [Bac94] D. Bacon, S. Graham, O. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, Vol. 26, No. 4, Dec. 1994.
- [Ben93] O. Bentz, *A Hardware Mapper for the Hyper High Level Synthesis System*, Masters Thesis, ERL U.C. Berkeley, 1993.
- [Bha93] S. Bhattacharyya and E. A. Lee, "Scheduling synchronous dataflow graphs for efficient looping", *Journal of VLSI Signal Processing*, Vol. 6, No. 3, pp. 271-288, Dec. 1993.
- [Box78] G. Box, W. Hunter, and S. Hunter, *Statistics for Experimenters: an Introduction to Design, Data Analysis, and Model Building*, 1st ed., John Wiley & Sons, New York, NY, 1978.
- [Bra84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, MA, 1984.
- [Bro92] R. W. Brodersen (ed.), *Anatomy of a Silicon Compiler*, Kluwer Academic Publishers, Boston, MA, 1992.
- [Bur94] T. Burd, *Low-Power CMOS Library Design Methodology*, Masters Thesis, ERL U.C. Berkeley, 1994.
- [Cat94] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man, "Global communication and memory optimizing transformations for low power signal processing systems, *IEEE Workshop on VLSI Signal Processing, VII*, pp. 178-187, 1994.

-
- [Cha93] A. Chatterjee, R. K. Roy, and M. A. d'Abreu, "Greedy hardware optimization for linear digital circuits using number splitting and refactorization", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 1, No. 4, pp. 423-431, 1993.
- [Cha95] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. Brodersen, "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design*, Vol. 14, No. 1, pp. 12-31, 1995.
- [Chu92] C. Chu, *Hardware Mapping and Module Selection in the Hyper Synthesis System*, Ph.D. Thesis, ERL U.C. Berkeley, 1992.
- [Cor93] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabaey, "Instruction set mapping for performance optimization," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 518-521, 1993.
- [Cor94] M. Corazao, "Instruction set mapping for performance and area," Masters Thesis, ERL U.C. Berkeley, 1994.
- [Cor96] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabaey, "Performance optimization using template mapping for datapath intensive high level synthesis" *IEEE Transactions on CAD*, Vol. 15, No. 8, pp. 877-888, August 1996.
- [Cor90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.
- [Cov91] T. Cover and J. Thomas, *Elements of Information Theory*, John Wiley & Sons, New York, NY, 1991.
- [Cro75] R. Crochiere and A. Oppenheim, "Analysis of linear networks," *Proceedings of the IEEE*, Vol. 63, No. 4, pp. 581-595, 1975.
- [Dey92] S. Dey, M. Potkonjak, and S. Rothweiler, "Performance optimization of sequential circuits by eliminating retiming bottlenecks," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 504-509, 1992.
- [Fet90] A. Fettweis, H. Meyr, and L. Thiele, "Algorithm transformations for unlimited parallelism," *IEEE International Symposium on Circuits and Systems*, pp. 1756-1759, 1990.
- [Fox89] G. C. Fox and J. G. Koller, "Code generation by a generalized neural network", *Journal of Parallel and Distributed Computing*, Vol. 7, No. 2, pp. 388-410, 1989.
- [Gaj92] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, MA, 1992.
- [Geu93] W. Geurts, F. Catthoor, and H. De Man, "Quadratic zero-one programming based synthesis of application specific data paths," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 522-525, 1993.
- [Gon94] J. Gong, D. Gajski, and S. Narayan, "Software estimation from executable specifications," *Journal of Computer and Software Engineering*, Vol. 2, No. 3, pp. 239-258, 1994.
-

-
- [Gue93] L. Guerra, "Regularity: Analysis and quantitative exploration in high-level and system-level synthesis," UC Berkeley EE 290T Class Project, Fall 1993.
- [Gue94] L. Guerra, M. Potkonjak, and J. Rabaey, "System-level design guidance using algorithm properties," *IEEE Workshop on VLSI Signal Processing, VII*, pp. 73 - 82, 1994.
- [Gue96] L. Guerra, M. Potkonjak, and J. Rabaey, "Divide-and-conquer techniques for global throughput optimization," *IEEE Workshop on VLSI Signal Processing, IX*, pp. 137-146, 1996.
- [Hal70] K. M. Hall, "An r-dimensional quadratic placement algorithm," *Manag. Sci.*, Vol. 17, pp. 219-229, 1970.
- [Har90] D. S. Harrison, A. R. Newton, R. L. Spickelmier, and T. J. Barnes, "Electronic CAD frameworks," *Proceedings of the IEEE*, Vol.78, No.2, pp. 393-417, 1990.
- [Har89] R. Hartley and A. Casavant, "Tree-height minimization in pipelined architectures," *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 112-115, 1989.
- [Hil85] P. Hilfinger, "A high-level language and silicon compiler for digital signal processing," *Custom Integrated Circuits Conference*, pp. 213-216, 1985.
- [Hoa92] P. Hoang, *Compiling Real-Time Digital Signal Processing Applications onto Multiprocessor Systems*, Ph.D. Thesis, ERL U.C. Berkeley, 1992.
- [Hua94] S. Huang, J. Rabaey, "Maximizing the throughput of high performance DSP applications using behavioral transformations," *European Design Automation Conference*, pp. 25-30, 1994.
- [Hua96] S. Huang and J. Rabaey, "An integrated framework for optimizing transformations," *IEEE Workshop on VLSI Signal Processing, IX*, pp. 263-272, 1996.
- [Hwa84] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.
- [Ito95] K. Ito and K. K. Parhi, "Determining the minimum iteration period of an algorithm," *Journal of VLSI Signal Processing*, Vol.11, No.3, pp. 229-244, Dec. 1995.
- [Iqb93] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker, "Critical path minimization using retiming and algebraic speed-up," *ACM/IEEE 30th Design Automation Conference*, pp. 573-577, 1993.
- [Jam87] L. Jamieson, "Characterizing parallel algorithms," in *The Characteristics of parallel algorithms*, L. Jamieson, D. Gannon, R. Douglass (eds.), MIT Press, Cambridge, MA, 1987.
- [Jai90] R. Jain, "Mosp: Module selection for pipelined designs with multi-cycle operations," *IEEE International Conference on Computer-Aided Design (ICCAD)*, 1990.
- [Jai91] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, New York, NY, 1991.
-

-
- [Jai92] R. Jain, A. Parker, and N. Park, "Predicting system-level area and delay for pipelined and non-pipelined designs," *IEEE Transactions on CAD*, Vol. 11, No. 8, pp. 955-965, Aug. 1992.
- [Jou89] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for super-scalar and superpipelined machines," *3rd International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 272-282, 1989.
- [Kai80] T. Kailath, *Linear Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
- [Kal93] A. Kalavade, E.A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design & Test of Computers*, Vol. 10, No. 3, pp. 16-28, 1993.
- [Keu94] K. Keutzer and P. Vanbekbergen, "The impact of CAD on the design of low power digital circuits," *IEEE Symposium on Low Power Electronics*, pp. 42-45, 1994.
- [Kle94] S. Kleinfeldt, M. Guiney, J. Miller, and M. Barnes, "Design methodology management," *Proceedings of the IEEE*, Vol.82, No.2, pp. 231-250, 1994.
- [Kna91] D. W. Knapp and A. C. Parker, "The ADAM design planning engine," *IEEE Transactions on CAD*, Vol.10, No.7, pp. 829-846, 1991.
- [Knu71] D. Knuth, "An empirical study of FORTRAN programs," *Software Practice & Experience*, Vol. 1, No. 2, pp. 105-133, 1971.
- [Kuc78] D. J. Kuck, *The Structure of Computers and Computations*, John Wiley, New York, NY, 1978.
- [Kum88] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers*, Vol. 37, No. 9, pp. 1088-1098, 1988.
- [Kun76] H. T. Kung, "New algorithms and lower bounds for parallel evaluation of certain rational expressions and recurrences," *Journal of the ACM*, Vol. 23, No. 2, pp. 252-261, 1976.
- [Kun84] S. Y. Kung, "On supercomputing with systolic/wavefront array processors", *Proceedings of the IEEE*, Vol. 72, No. 7, 1984.
- [Kun88] S. Y. Kung, *VLSI Array Processors*, Prentice Hall, Englewood Cliffs, NJ, 1988.
- [Kur89] F. Kurdahi and A. Parker, "Techniques for area estimation of VLSI layouts," *IEEE Transactions on CAD*, Vol. 8, No. 1, pp. 81-92, 1989.
- [Lee87] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, Vol. 36, No. 1, pp. 24-35, 1987.
- [Lee93] J. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," *International Conference on Acoustics, Speech, and Signal Processing*, pp. 429-432, 1993.
- [Lei83] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuits by retiming," *Proceedings of the Third Caltech Conference on VLSI*, pp. 87-116, 1983.
-

-
- [Len90] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Teubner, Chichester, U.K., 1990.
- [Li90] M. Li and P. M. B. Vitanyi, "Kolmogorov Complexity and its Applications," in *Handbook of Theoretical Computer Science, Volume A*, J. V. Leeuwen (ed.), pp. 187-254, MIT Press, Cambridge, MA, 1990.
- [Lid94] D. Lidsky and J. Rabaey, "Low-power design of memory intensive functions case study: vector quantization," *IEEE Workshop on VLSI Signal Processing, VII*, pp. 378-387, 1994.
- [Lip89] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL, Hardware Description and Design*, Kluwer Academic Publishers, Boston, MA, 1989.
- [Lop92] J. C. Lopez, M. F. Jacome, and S. W. Director, "Design assistance for CAD frameworks," *European Design Automation Conference*, pp. 494-499, 1992.
- [Mas87] H. Massalin, "Superoptimizer: A look at the smallest program", *International Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 122-126, 1987.
- [McK65] W. M. McKeeman, "Peephole optimization," *Communications of the ACM*, Vol. 8, No. 7, pp. 443-444, 1965.
- [Mea80] C. Mead and L. Conway, *Introduction to VLSI systems*, Addison-Wesley, Reading, MA, 1980.
- [Meh94] R. Mehra and J. Rabaey, "Behavioral level power estimation and exploration," *International Workshop on Low-Power Design*, pp. 197-202, 1994.
- [Meh96a] R. Mehra, L. Guerra, and J. Rabaey, "Reducing interconnect power using partitioning," to appear in the *Journal of VLSI Signal Processing*, 1996.
- [Meh96b] R. Mehra and J. Rabaey, "Exploiting regularity for low-power design," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 166-172, 1996.
- [Mes88] D. G. Messerschmitt, "Breaking the recursive bottleneck," *Performance Limits in Communication Theory and Practice*, J. K. Skwirzinsky (ed.), Kluwer Academic Publishers, Amsterdam, 1988.
- [Moh88] B. Mohar, "The laplacian spectrum of graphs," *Graph Theory, Combinatorics, and Applications*, Y. Alavi *et al.* (eds.), John Wiley, New York, NY, pp. 871-898, 1988.
- [Mor79] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Communications of the ACM*, Vol. 22, No. 2, pp. 96-103, 1979.
- [Mus95] E. Musoll and J. Cortadella, "High-level synthesis techniques for reducing the activity of functional units," *International Symposium on Low-Power Design*, pp. 99-104, 1995.
- [Nic84] A. Nicolau and J.A. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Transactions on Computers*, Vol. 33, No. 11, pp. 968-976, 1984.
-

-
- [Not91] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-III: architecture-driven high-level synthesis for high throughput DSP applications," *ACM/IEEE 28th Design Automation Conference*, pp. 597-602, 1991.
- [Opp89] A. Oppenheim and R. Schaffer, *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.
- [Pap93] E. Papaefstathiou, D. Kerbyson, and G. Nudd, "A layered approach to the characterization of parallel systems for performance prediction," *Performance Evaluation of Parallel Systems Workshop*, Coventry, UK, pp. 26-34, 1993.
- [Par95] K.K. Parhi, "High-level algorithm and architecture transformations for DSP synthesis," *Journal of VLSI Signal Processing*, Vol. 9, No. 1-2, pp. 121-143, 1995.
- [Pat90] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.
- [Pau89] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on CAD*, Vol. 8, No. 6, pp. 661-679, 1989.
- [Pot89] M. Potkonjak and J. Rabaey, "A scheduling and resource allocation algorithm for hierarchical signal flow graphs," *26th ACM/IEEE Design Automation Conference*, pp. 7-12, 1989.
- [Pot91] M. Potkonjak and J. Rabaey, "Optimizing the resource utilization using transformations," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 88-91, 1991.
- [Pot92] M. Potkonjak and J. Rabaey, "Maximally fast and arbitrarily fast implementation of linear computations," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 304-308, 1992.
- [Pot93] M. Potkonjak and J. Rabaey, "Exploring the DSP algorithm design space using Hyper," *IEEE Workshop on VLSI Signal Processing, VI*, pp. 123-131, 1993.
- [Pot94a] M. Potkonjak and M. Srivastava, "Design of high throughput, low latency and low cost structures for linear systems," *IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 2, pp. 497-500, 1994.
- [Pot94b] M. Potkonjak and J. Rabaey, "Optimizing throughput and resource utilization using pipelining: transformation based approach," *Journal of VLSI Signal Processing*, Vol. 8, No. 2, pp. 117-130, Oct. 1994.
- [Pot94c] M. Potkonjak, M. Srivastava, A. Chandrakasan, "Efficient substitution of multiple constant multiplications by shifts and additions using iterative pairwise matching," *ACM/IEEE Design Automation Conference*, pp. 189-194, 1994.
- [Rab91a] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of data path intensive architectures," *IEEE Design & Test of Computers*, Vol. 8, No. 2, pp. 40-51, June 1991.
- [Rab91b] J. Rabaey and M. Potkonjak, "Complexity estimation for real time application specific circuits," *IEEE European Solid State Circuits Conference*, pp. 201-204, Sept. 1991.
-

-
- [Rab95] J. Rabaey, L. Guerra, and R. Mehra, "Design guidance in the power dimension," *IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 5, pp. 2837-2840, 1995.
- [Rag95] A. Raghunathan and N. Jha, "An iterative improvement algorithm for low power data path synthesis," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 597-602, 1995.
- [Rao92] D. Rao and F. Kurdahi, "Partitioning by Regularity Extraction," *ACM/IEEE 29th Design Automation Conference*, pp. 235-238, 1992.
- [Rao93] D.S. Rao and F.J. Kurdahi, "An Approach to Scheduling and Allocation Using Regularity Extraction," *European Design Automation Conference*, pp. 557-561, 1993.
- [Rob87] M. Roberts, "Optimizing compilers," *Byte Magazine*, October 1987.
- [SIA95] Semiconductor Industry Association, "The national technology roadmap for semiconductors," San Jose, CA, 1995.
- [Seq83] C.H. Sequin, "Managing VLSI Complexity: An Outlook", *Proceedings of the IEEE*, Vol. 71, No. 1, pp. 149-166, 1983.
- [Sha93] A. Sharma and R. Jain, "Estimating architectural resources and performance for high-level synthesis applications," *IEEE Transactions on VLSI Systems*, Vol. 1, No. 2, pp. 175-190, 1993.
- [Sin88] K. J. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 282-285, 1988.
- [Sri95a] M. Srivastava and M. Potkonjak, "Energy efficient implementation of linear systems on programmable processors," *IEEE Workshop on VLSI Signal Processing, VIII*, pp. 147-156, 1995.
- [Sri95b] M. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol.3, No.1, pp. 2-19, March 1995.
- [Tho93] D. Thomas, J. Adams, and H. Schmitt, "A Model and Methodology for Hardware-Software Codesign," *IEEE Design & Test of Computers*, Vol. 10, No. 3, pp. 6-15, 1993.
- [Tja70] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of parallel instructions," *IEEE Transactions on Computers*, Vol. 19, No. 10, pp. 889-895, 1970.
- [Vee84] H. J. M. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *IEEE Journal of Solid-State Circuits*, pp. 468-473, August 1984.
- [Ver94] I. Verbauwhede, C. Scheers, and J. Rabaey, "Specification and support for multi-dimensional DSP in the SILAGE language," *International Conference on Acoustics, Speech and Signal Processing*, pp. 19-22, 1994.
-

-
- [Wal91] R. Walker and R. Camposano (eds.), *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, Boston, MA, 1991.
- [Wan94] M. Wan and E. Yeo, "C++ behavioral input description to the Hyper system," UC Berkeley EE 244 class project report, Fall 1994.
- [Weg91] M. Wegman and F. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages*, Vol. 13, No. 2, pp. 181-210, 1991.
- [Whi90] D. Whitfield and M. Soffa, "An approach to ordering optimizing transformations," *ACM Symposium on Principles and Practice of Parallel Programming*, Vol. 25, No. 3, pp. 137-146, 1990.
- [Wol89] M. J. Wolfe, "Automatic vectorization, data dependence, and optimization for parallel computers," *Parallel Processing for Supercomputing and Artificial Intelligence*, K. Hwang and De Groot (eds.), Ch. 11, McGraw-Hill, New York, NY, 1989.
- [Wol91] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 452-471, Oct. 1991.
- [Wu94] S. Wu, *A Hardware Library Representation for the Hyper Synthesis System*, Masters Thesis, ERL U.C. Berkeley, 1994.
- [Wuy94] S. Wuytack, F. Catthoor, F. Franssen, L. Nachtergaele, and H. De Man, "Global communication and memory optimizing transformations for low power systems," *International Workshop on Low-Power Design*, pp. 203-208, 1994.

APPENDIX A: Architecture Model — Rules and Constraints

The architecture model specifies a set of resource types and the rules and constraints to be used in their composition. An example rule could be: “execution units can connect to any other resource type” or “register files can only connect to execution units.” Example constraints are: “the maximum allowed buses is 5” or “the maximum allowed fanout of a register file is 1.” Thus, two chips with different numbers of busses or registers follow the same architecture model as long as they use similar resources and follow the same rules and constraints.

This appendix presents the rules and constraints of the targeted architecture model. A sample architecture instance is shown in Figure 52.

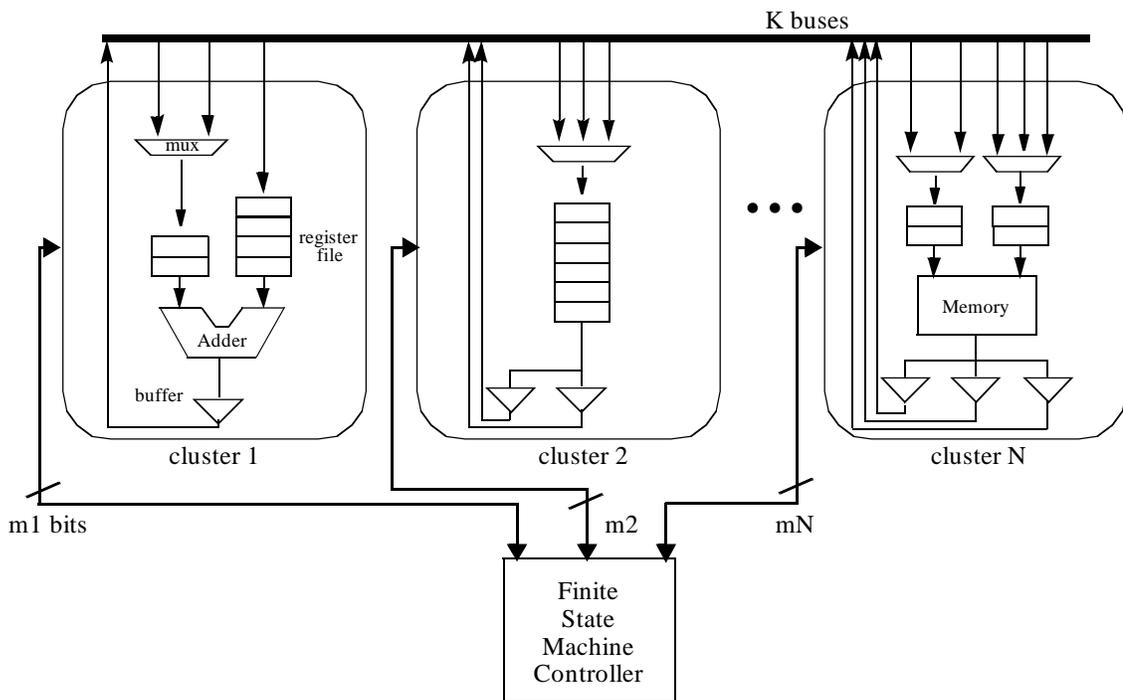


Fig. 52. Sample architecture instance

General

- Resource types are multiplexors, register files, buffers, interconnect, and execution unit blocks. An execution unit block is a single execution unit, a chained set of execution units, or a memory.
- The size of each register file is unbounded.
- Resources are grouped into clusters.
- The number of clusters is unbounded.
- In each cluster:
 - there is at most 1 execution unit block.
 - the number of register files is exactly equal to the number of execution unit block input ports. If there is no execution unit, a single register file is used.
 - the number of multiplexors is less than or equal to the number of register files.
 - the number of buffers is unbounded.
- There is a single finite state machine controller (implying single-threaded control).
- The controller has a single pipeline stage.

Connections

- A multiplexor can connect to exactly one of the register files in its cluster.
- A register file can connect to an input of its cluster's execution unit block. If there is no execution unit, the register file connects to all of the buffers in its cluster.
- An execution unit connects to all of the buffers in its cluster.
- A buffer can connect to any number of multiplexors or register files anywhere.
- The connections made between buffers and other resources can be merged, resulting in a single hardware-shared bus. The same does not hold for connections between any other resource types.
- The controller can connect to any number of multiplexors, register files, execution units, and buffers. It can also connect off-chip.

APPENDIX B: Summary of Property Metrics

The specific property metrics proposed in this thesis are tabulated in Tables 11 and 12. While some metrics are purely algorithmic (Table 11), for others, the more information the designer provides about the architecture, design constraints, design parameters, and hardware library, the more accurate the definitions are (Table 12). The computation of the property metrics in Table 12 use the operation delay information, computed from the design parameters and models in the hardware library.

Property Class	Property Metric
Size	Number of operations of each type (e.g., addition, shift, sample delay, constant multiplication, variable multiplication)*
	Number of operator types
	Number of data transfers of each type (e.g. shift-to-shift, shift-to-multiply)
	Number of data transfer types
	Number of inputs and outputs
	Number of constants
	Number of array write and read accesses
	Number of variables
	Number of for-loops
	Bitwidths for each operator type

Table 11: Algorithmic properties.

Property Class	Property Metric
Topology	Longest path
	Average node fanout
	Ratio of the number of operations to the longest path
	Number of strongly connected components (SCCs)
	Percentage of operations in feedback cycles
Spatial Locality	Number of isolated components
	Number of bridges
	Connectivity ratio of the number of internal data transfers to the total possible data transfers*
	Second smallest eigenvector placement: number of distances greater than $\mu + a \cdot \sigma$, for $a = 1, 2, \text{ and } 3$
Regularity	Percentage operation coverage for common templates (e.g., the set of patterns containing all chained pairs of associative operators)
	size/(descriptive complexity)
Other	Computation classification as linear, non-linear but feedback linear*, non-linear but not feedback linear*
	Formal degree*

Table 11: Algorithmic properties.

*Also computed for the ϵ -critical network, and for each for-loop and SCC.

Property Class	Property Metric
Timing	Critical path [*]
	Latency
	Ratio of the sample period to the critical path
	Percentage and types of operations on the ϵ -critical network; $\epsilon = 0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.5, 0.75, 1.0$
	Slack average, quartiles, and variance over all operations and for each operator type
	Iteration bound [*]
	Ratio of iteration bound to critical path
	Maximum operator delay
	Maximum operator delay on the ϵ -critical network
Concurrency	<i>Maximumheight</i> and <i>maximumheight'</i> of operation distribution graph
	<i>Maximumheight</i> and <i>maximumheight'</i> of register write distribution graph
	<i>Maximumheight</i> and <i>maximumheight'</i> of data transfer distribution graph
	<i>Maximumheight</i> of midpoint, distributed, and split-distributed variable lifetime distribution graphs
Uniformity	Variance of the distribution graphs
Temporal Locality	Average variable lifetimes using the midpoint lifetime approximation

Table 12: Algorithmic properties dependent on user-defined design parameters, hardware library, and performance constraints.

^{*}Also computed for each for-loop and SCC.

APPENDIX C: Library of Optimization Characterizations

The library of optimization characterizations contains the pre-characterized optimizations. During guidance, these characterizations are combined with the property metrics extracted during design characterization to predict the optimization effectiveness on the specific design. In library development, an understanding of the optimization degrees of freedom is important to assure that relevant optimizations are characterized. An optimization's effects are a function of the design's property metrics, and thus an understanding of the property metrics is crucial.

A pseudo-code format is used to describe each of the entries in the library of optimization characterizations. There are four fields in each entry: name, command-line call, trigger, and feedback. The first field specifies the name of the optimization technique, followed by references when appropriate. The command-line call specifies whether an automated routine is available in the environment. Automated routines from the Hyper optimization library have been encapsulated. For each, the appropriate command-line call is given. The trigger expression is used for pruning optimizations from consideration. If it evaluates to true, the optimization is considered; if false, the optimization is pruned. In trigger expressions, logical *AND* is represented by “&” and logical *OR* is represented by “||.” The feedback field specifies the feedback to be presented to the designer and to be used in ranking.

Goal: Throughput Improvement (no restriction on area and power)

Name: **Library Selection**

Command-line call: Manual

Trigger: “an alternate library contains a faster module for at least one operation on the ϵ -critical network”

Feedback:

NewDesign = ModuleSelectionForSpeed(Design, NewLibrary) [Chu92].

NewCriticalPath = GetCriticalPath(NewDesign).

Name: **Module Selection** [Chu92]

Command-line call: Manual *OR* ModuleSelect -a -L “Library” -G “speed” Design

Trigger: “the current library contains a faster module for at least one operation on the ϵ -critical network”

Feedback:

NewDesign = ModuleSelectionForSpeed(Design, Library) [Chu92].

or

NewDesign = ManualModuleSelection(Design, Library).

NewCriticalPath = GetCriticalPath(NewDesign).

Name: **Clock Selection** [Cor96]

Command-line call: Manual *OR* ModuleSelect -a -C Design

Trigger: “clock period is greater than the largest operator delay || ClockCycleUtilization < α_1 ”

ClockCycleUtilization = average clock cycle utilization over all operations on the ϵ -critical network. Clock cycle utilization for an operation is defined as its delay divided by its delay rounded up to the nearest multiple of the clock period. $\text{GetCriticalPathAssumingNoRegistersBetweenOperations} / \text{GetCriticalPath}$.

α_1 is an empirically derived threshold between 0 and 1, but typically close to 1.

Feedback:

if (clock period is longer than the largest operator delay) {

Ranking override: Set rank of this optimization to “1.”

While its immediate effect may be minimal, the large clock period may result in a skewed picture of the design. Thus this optimization is suggested for immediate application.

}

NewCriticalPath \geq GetCriticalPathAssumingNoRegistersBetweenOperations.

With each transformation or parameter change, the clock cycle utilization along the critical path is likely to change. It’s usually prudent to apply clock selection after significant changes in the computation’s operation delays or in the clock

cycle utilization. In general, if clock selection hasn't been applied for a number of optimization steps, then it is prudent to apply it. It is possible to augment the ranking to take this into account.

Area is likely to go down, with the reduction in the critical path to sample period ratio. This is especially true the closer the ratio initially was to 1.

Name: **Voltage Selection** [Cha95]

Command-line call: Manual

Trigger: "current voltage is less than the maximum voltage as determined by technology constraints"

Feedback:

NewVoltage = MaxVoltage *OR* user-specified Voltage.

NewCriticalPath = EstimateCriticalPathUsingEmpiricalVoltageDelayCurve(CriticalPath, Voltage, MaxVoltage) [Cha95].

Power approximately increases quadratically with voltage.

Area is likely to go down, with the reduction in the critical path to sample period ratio. This is especially true the closer the ratio initially was to 1.

Name: **Template Matching + Clock Selection** [Cor96]

Command-line call: ModuleSelect -a -C -T TemplateLibrary -G "speed" Design

Trigger: "percentage ϵ -critical network coverage by common pre-defined templates $> \alpha_2$ "

α_2 is an empirically derived threshold between 0 and 1, but typically closer to 0.

Feedback:

NewCriticalPath = (1-FractionCovered)•CriticalPath + FractionCovered•CriticalPath•EstimatedReductionFactor.

FractionCovered is the percentage template coverage on the ϵ -critical network.

EstimatedReductionFactor = average over all potential templates of:

(TemplateDelay/AveOperDelayAmongOpersInTemplate)/NumOpersInTemplate.

Area tends to increase. The resulting computation has fewer primitive operations, but usually a larger number of hardware functional types. As a result, resource utilization tends to drop.

Table presenting patterns and percentage coverage on ϵ -critical network is given.

Name: **Pipelining-for-Speed** [Lei83, Pot94b]

Command-line call: RetimeCP -a -p Design

Trigger: "critical path is greater than max(\lceil iteration bound \rceil , maximum operator delay)"

Feedback:

$(\text{NewCriticalPath}, \text{NewDesign}) = \text{PipeliningRoutineWithoutInitialValueRecalculation}(\text{Design}).$

$\text{NewLatency} = \text{GetLatency}(\text{NewDesign}).$

Observations:

The value $\max(\lceil \text{iteration bound} \rceil, \text{maximum operator delay})$ is a lower bound on the new critical path.

When applying both constant multiplication expansion and pipelining, it is typically better to apply pipelining *after* constant multiplication expansion.

Name: **Retiming-for-Speed** [Lei83]

Command-line call: `RetimeCP -a Design`

Trigger: “the computation has sample delays & critical path is greater than $\max(\lceil \text{iteration bound} \rceil, \text{maximum operator delay})$ ”

Feedback:

$(\text{NewCriticalPath}, \text{NewDesign}) = \text{RetimingRoutineWithoutInitialValueRecalculation}(\text{Design}).$

Observations:

Typically, retiming does not increase the latency.

As it changes the distribution of the operations significantly, the computation uniformity may also change.

The value $\max(\lceil \text{iteration bound} \rceil, \text{maximum operator delay})$ is a lower bound on the new critical path.

Name: **Time-Loop Unfolding (TLU)**

Command-line call: `Manual`

Trigger: “number of operations $< \alpha_3$ ”

This threshold, α_3 , specifies the size of the computation beyond which TLU overhead becomes too expensive to handle practically.

Feedback:

if (computation is non-recursive)

$\text{NewCriticalPath} = \text{CriticalPath} / (i+1).$

else

$\text{NewCriticalPath} \geq \text{iteration bound}.$

NewCriticalPath depends on the unfolding factor selected. It is at best equal to the iteration bound. For non-recursive computations, the iteration bound of zero can be approached to an arbitrary closeness. For all other computations, the iteration bound can always be reached, with a high enough level of unfolding.

Tables showing the critical path and iteration bound of each SCC are presented. Tables presenting the speedup for various values of the unfolding factor are presented. These tables are useful in selection of the unfolding factor.

TLU is an exceptionally effective optimization enabler. Its predicted enabling potential is inversely related to the formal degree of the computation.

Name: **Common Sub-Expression Elimination (CSE)** [Mor79]

Command-line call: CSE -a Design

Trigger: “there is at least one common sub-expression”

Note: this is determined easily if the computation has no conditionals, otherwise it is an undecidable problem [Weg91].

Feedback:

No immediate effect on throughput.

Can potentially enable more effective subsequent optimizations, such as template matching. However, its potential for enhancing effectiveness is limited, for throughput.

Removes redundant operations, thus area of execution units may be reduced.

CSE is often included towards the end of optimization sequences.

Name: **Common Sub-Expression Replication (CSR)**

Command-line call: Manual

Trigger: “there is at least one operation in a cycle that has fanout to at least two other operations” (if the operation is not in a cycle, then it can be pipelined)

Feedback:

No immediate effect on throughput.

CSR is an effective optimization enabler. It is especially effective in enabling algebraic transformations and retiming.

Enabling potential is in general inversely related to the computation’s formal degree. It is positively related to the number of operations with fanout that lie on the ϵ -critical network. Furthermore, the closer the critical path is to $\lceil \log_2(\text{NumStates} + 1) \rceil + \text{multDuration}$, the less further improvement is expected (*NumStates* is the number of sample delays in the computation), since this gives an indication of what can be achieved using algebraic optimizations.

Never end a sequence of optimizations with CSR.

Name: **Constant Multiplication Replacement by Additions and Shifts** [Rab91a or Pot94c] + **Module Selection + Clock Selection**

Command-line call: Multiplications -a Design; ModuleSelect -a -C -L "Library" -G "speed" Design

Trigger: "there is at least one constant multiplication"

Feedback:

Apply the script since constant multiplication replacement is fast (linear in the size of the number of constant multiplications).

NewCriticalPath = GetCriticalPath(NewDesign).

Observations:

The resulting critical path in clock cycles often increases. However this depends greatly on the coefficient values and the multiplication operand bitwidths. Simple coefficients are replaced with just a few additions and shifts (e.g., multiplication by 2 can be replaced with a single shift operation). The greater the bitwidths, the greater the difference in delay of multiplication and addition/shift (multiplication delay grows quadratically with the bitwidth while addition or shift delays grow linearly). After replacing constant multiplications, it is possible that a better clock cycle utilization can be obtained.

Area is likely to go down, since expensive multiplications of size A_m are replaced by shifts of size A_s and additions of size A_a . However this depends on several factors such as the bitwidths, the characteristics of variable multiplications, and the register overhead. Area of multipliers grows quadratically with the bitwidth while adder or shift areas grow linearly. The number of multipliers required may not reduce if there are variable multiplications remaining in the computation. The following metrics give an indication of expected multiplication resource needs: the concurrency of multiplications is $maximumheight_{mult.all}$ and the concurrency of just variable multiplications is $maximumheight_{mult.var}$. Another area consideration is that with the added operation count, register needs may increase. Finally, if hardware sharing of shifts is not done, then they can be eliminated.

Name: **Maximally-Fast Script** [Pot92]

Command-line call: Manual

Trigger: "computation is recursive, and either linear or feedback linear"

Feedback:

$CriticalPathUsingMaxFast \leq \lceil \log_2(NumStates + 1) \rceil + multDuration .$

In the case where multiplications are expanded into shifts and additions, an analogous equation exists.

if ($CriticalPathUsingMaxFast < CriticalPath$)

$\text{NewCriticalPath} = \text{CriticalPathUsingMaxFast}.$

Bounds on the resulting number of additions and multiplications are given by the closed-form equations presented in [Sri95a]. These affect both area and power.

The computation uniformity, and thus area will increase significantly.

Name: **Time-Loop Unfolding + Maximally-Fast Script** [Pot92]

Command-line call: Manual

Trigger: “computation is recursive, and either linear or feedback linear”

Feedback:

$\text{NewCriticalPath} \leq (\lceil \log_2(\text{NumStates} + 1) \rceil + \text{multDuration}) / (i+1),$

where i is the unfolding factor.

In the case where multiplications are expanded into shifts and additions, an analogous equation exists.

Resulting number of additions and multiplications are given by the closed-form equations presented in [Sri95a]. These affect both area and power.

Generate tables presenting the speedup for various unfolding factors. Generate tables presenting the number of multiplication and addition operations and the predicted voltage scaling achievable, for various unfolding factors. This information is useful in indicating area and power impact.

Name: **Associativity-for-Iteration Bound + Pipelining** [Hua94]

Command-line call: Algebraic -a -rRD -g -i Design; RetimeCP -ap Design

Trigger: “for at least one non-trivial SCC, on its ϵ -critical network, Trigger_a is TRUE;
 Trigger_a : “there are at least 2 associative operations in sequence”

Feedback:

Associativity can reduce the iteration bound by moving operations out of the cycles. All operations outside of the cycles can then be pipelined.

$\text{NewIterationBound} = \text{IterationBound} \cdot (1 - \alpha_4 \cdot \text{RatioCoverage})$ where RatioCoverage = percentage coverage of SCCs by associative pairs of operators, and α_4 is an empirically derived constant between 0 and 1.

$\text{NewCriticalPath} = \max(\lceil \text{newiteration bound} \rceil, \text{maximum operator delay}).$

Name: **Associativity-for-Critical Path** (with CSR and distributivity enablers) [Har89, Hua94]

Command-line call: Algebraic -a -g -c Design

Trigger: “there is at least one associative pair of operations on the ϵ -critical network”

Feedback:

Transforms from chain to tree structures.

$$\text{NewCriticalPath} \geq \lceil \log_2(\text{number of operations on the critical path} + 1) \rceil \cdot \text{AverageOperatorDelayRoundedUpToMultipleOfClockPeriod}.$$

As the optimization has execution time proportional to the number of operations in the computation, it is also possible to directly apply it.

Name: **For-Loop Unfolding**

Command-line call: Manual

Trigger: “there is at least one for-loop”

Feedback:

NewCriticalPath depends directly on the unfolding factor selected.

The effective critical path per iteration of the loop body can be reduced to the for-loop iteration bound.

With unfolding, the loop overhead (bound checking and update of the bound index) is reduced proportionally to the unfolding factor. Array index calculation may become more complex with unfolding. Further, an additional loop prologue may be required.

For-loop unfolding can enable subsequent optimizations since it exposes parallelism. Its predicted enabling potential is inversely related to the loop formal degree.

A distinct difference between for-loop unfolding and time-loop unfolding is that for loops can only be unfolded a finite number of times.

Name: **Eliminate Hardware Sharing**

Command-line call: Manual

Trigger: “TRUE”

Feedback:

$$\text{NewCriticalPath} = \text{GetCriticalPathAssumingNoRegisters BetweenOperationsAndSkippingShifts}(\text{Design}).$$

(Without hardware sharing, the computation’s intermediate variables need not be latched and thus the critical path is not quantized into clock cycles.)

Shifts can be hardwired. Control overhead is eliminated.

This value is an upper bound, since it does not take into account chaining effects and false paths.

Implementation area is proportional to the size of the computation.

Name: **Partition into SCCs using Pipelining** [Gue96]

Command-line call: Manual

Trigger: “the computation has more than one non-trivial SCC”

Feedback:

NewCriticalPath = longest critical path among all non-trivial SCCs.

SCC partitioning is a highly effective divide-and-conquer based optimization enabler. Not only is each part more manageable in size than the overall computation, the approach enables independent throughput optimization of each part, in accordance with its properties. Optimizations that were not applicable to the computation as a whole can often be applied to its parts. Its effectiveness depends on the number of SCCs and the formal degree of each.

Name: **Partition into Cones** [Dey92, Gue96]

Command-line call: Manual

Trigger: “the computation is recursive & the computation is neither linear nor feedback linear & there are at least two sample delays”

Feedback:

No immediate effect on throughput.

Replication results in increased computation size.

Cone partitioning is an effective optimization enabler.

Name: **Latency-Throughput script, Single Input, Single Output Case** [Pot94a, Sri95b]

Command-line call: Manual

Trigger: “computation is linear & computation has a single input and single output”

Feedback:

NewCriticalPath = multiplication delay + addition delay.

Goal: Power Minimization Under Throughput Constraint

In this work, optimization characterization for power has focused on optimization impact on power reduction through voltage scaling and through reduction of effective capacitance (due to the number and type of operations). These factors were chosen due to their high impact on the overall power. Information about an optimization's impact on the effective capacitance of other components such as interconnect, clock, and registers is also provided in a number of cases. An alternative to throughput improvement for voltage scaling is throughput improvement to enable increased scheduling slack and thus better computation uniformity, resource utilization, active area, and consequently interconnect capacitance. If the voltage is fixed or has already been lower to the minimum value, then this alternative can be used.

Given a specific speedup of the critical path as compared to the available time, an estimate of a new voltage and of the resulting power can be made. The estimate is based on the empirical voltage-delay curve presented by Chandrakasan *et al.* [Cha95]. The estimate also assumes that the delay of all modules scale with voltage at the same rate. The estimate takes into account the minimum voltage as specified by technology constraints. A rule of thumb integrated into the estimate is that scaling to the maximum extent possible is not always the most prudent choice. With maximum voltage scaling, timing constraints will be tight with the critical path being very close or equal to the available time. The suggested voltage and thus power estimation is based on maintaining a $k\%$ slack between critical path and available time (we use an experimentally derived value of k equal to 10), to allow scheduling slack and thus better resource utilization, area, and thus interconnect capacitance. Throughout this appendix, the routine “(NewPower, SuggestedNewVoltage) = EstimatePower&SuggestVoltage(Speedup, Voltage)” is used. The routine takes two

inputs: the speedup and the current voltage. The routine returns a suggested new voltage and a corresponding power estimate.

Name: **Library Selection**

Command-line call: Manual

Trigger: “an alternate library contains a more power efficient module for at least one operation (lower capacitance, C) || a faster module for at least one operation on the ϵ -critical network (lower delay, T) || a better global TC product than the current library”

For the global TC product, T is the weighted average delay of all operations on the ϵ -critical network, and C is the weighted average capacitance of all operations.

Feedback:

NewDesign = ModuleSelectionForSpeed(Design, NewLibrary) [Chu92].

or

NewDesign = ManualModuleSelection(Design, NewLibrary).

NewCriticalPath = GetCriticalPath(NewDesign).

NewPower = GetPower(NewDesign).

Data for evaluating the various libraries is presented. For each library, a table containing the following information is created: for each operation type, percentage of operations of that type in the computation, percentage of operations of that type on the ϵ -critical network, delay of each module that can implement that operation, capacitance of modules, TC product of modules.

Name: **Module Selection**

Command-line call: Manual *OR* ModuleSelect -a -L “Library” -G “speed” Design

Trigger: “the current library contains a more power efficient module for at least one operation (lower capacitance, C) || a faster module for at least one operation on the ϵ -critical network (lower delay, T) || a module with a better TC product for at least one operation”

Feedback:

NewDesign = ModuleSelectionForSpeed(Design, Library) [Chu92].

or

NewDesign = ManualModuleSelection(Design, Library).

NewCriticalPath = GetCriticalPath(NewDesign).

$\text{NewPower} = \text{GetPower}(\text{NewDesign}).$

Data for evaluating the various modules is presented. A table containing the following information is created: for each operation type, percentage of operations of that type in the computation, percentage of operations of that type on the ϵ -critical network, delay of each module that can implement that operation, capacitance of modules, TC product of modules.

Name: **Clock Selection** [Cor96]

Command-line call: Manual *OR* ModuleSelect -a -C Design

Trigger: “clock period is greater than the largest operator delay || $\text{ClockCycleUtilization} < \alpha_5$ ”

$\text{ClockCycleUtilization}$ = average clock cycle utilization over all operations on the ϵ -critical network. Clock cycle utilization for an operation is defined as its delay divided by its delay rounded up to the nearest multiple of the clock period. $\text{GetCriticalPathAssumingNoRegistersBetweenOperations} / \text{GetCriticalPath}$.

α_5 is an empirically derived threshold between 0 and 1, but typically close to 1.

Feedback:

Often applied with voltage selection.

if (clock period is longer than the largest operator delay) {

 Ranking override: Set rank of this optimization to “1.”

 While its immediate effect may be minimal, the large clock period may result in a skewed picture of the design. Thus this optimization is suggested for immediate application.

}

$\text{NewCriticalPath} \geq \text{GetCriticalPathAssumingNoRegistersBetweenOperations}.$

With each transformation or parameter change, the clock cycle utilization along the critical path is likely to change. It’s usually prudent to apply clock selection after significant changes in the computation’s operation delays or in the clock cycle utilization. In general, if clock selection hasn’t been applied for a number of optimization steps, then it is prudent to apply it. It is possible to augment the ranking to take this into account.

Capacitance: the smaller the clock period, the greater the power dissipated in clocking. Thus larger clock periods are favored.

Name: **Voltage Selection** [Cha95]

Command-line call: Manual

Trigger: “current voltage is greater than the minimum voltage as determined by technology constraints & the critical path is shorter than the available time”

Feedback:

$(\text{NewPower}, \text{SuggestedNewVoltage}) = \text{EstimatePower\&SuggestVoltage}(\text{Speedup}, \text{Voltage}).$

Name: **Template Matching + Clock Selection** [Cor94, Cor96]

Command-line call: `ModuleSelect -a -C -T TemplateLibrary -G "speed" Design` *OR*

`ModuleSelect -a -C -T TemplateLibrary -G "area" Design`

Trigger: "percentage computation coverage by common pre-defined templates $> \alpha_6$ "

α_6 is an empirically derived threshold between 0 and 1, but typically closer to 0.

Feedback:

In this optimization, two modules may be chained, or a combined unit such as a multiply-accumulate unit may be used.

$\text{NewCriticalPath} = (1 - \text{FractionCovered}) \cdot \text{CriticalPath} + \text{FractionCovered} \cdot \text{CriticalPath} \cdot \text{EstimatedReductionFactor}.$

FractionCovered is the percentage template coverage on the ϵ -critical network.

EstimatedReductionFactor = average over all potential templates of:

$(\text{TemplateDelay} / \text{AveOperDelayAmongOpersInTemplate}) / \text{NumOpersInTemplate}.$

Capacitance: register accesses reduces due to chaining. Perform a greedy coverage using the 2 most prevalent patterns. Then the new weighted average capacitance of operation accesses can be computed.

Table presenting patterns and percentage coverage on computation as well as on ϵ -critical network is given.

Name: **Pipelining-for-Speed** [Lei83, Pot94b]

Command-line call: `RetimeCP -a -p Design`

Trigger: "critical path is greater than $\max(\lceil \text{iteration bound} \rceil, \text{maximum operator delay})$ "

Feedback:

$(\text{NewCriticalPath}, \text{NewDesign}) = \text{PipeliningRoutineWithoutInitialValueRecalculation}(\text{Design}).$

$\text{NewLatency} = \text{GetLatency}(\text{NewDesign}).$

$\text{NewPower} = \text{GetPower}(\text{NewDesign}).$

Observations:

The value $\max(\lceil \text{iteration bound} \rceil, \text{maximum operator delay})$ is a lower bound on the new critical path.

Capacitance: It is likely that capacitance will decrease since the added scheduling freedom that pipelining brings often results in smaller areas, and thus smaller interconnect capacitance.

When applying both constant multiplication expansion and pipelining, it is typically better to apply pipelining *after* constant multiplication expansion.

Name: **Retiming-for-Speed** [Lei83]

Command-line call: RetimeCP -a Design

Trigger: “the computation has sample delays & critical path is greater than $\max(\lceil \text{iteration bound} \rceil, \text{maximum operator delay})$ ”

Feedback:

$(\text{NewCriticalPath}, \text{NewDesign}) = \text{RetimingRoutineWithoutInitial-ValueRecalculation}(\text{Design}).$

$\text{NewPower} = \text{GetPower}(\text{NewDesign}).$

Observations:

Typically, retiming does not increase the latency.

The value $\max(\lceil \text{iteration bound} \rceil, \text{maximum operator delay})$ is a lower bound on the new critical path.

Name: **Time-Loop Unfolding (TLU)**

Command-line call: Manual

Trigger: “number of operations $< \alpha_7$ ”

This threshold, α_7 , specifies the size of the computation beyond which TLU overhead becomes too expensive to handle practically.

Feedback:

if (computation is non-recursive)

$\text{NewCriticalPath} = \text{CriticalPath} / (i+1).$

else

$\text{NewCriticalPath} \geq \text{iteration bound}.$

NewCriticalPath depends on the unfolding factor selected. It is at best equal to the iteration bound. For non-recursive computations, the iteration bound of zero can be approached to an arbitrary closeness. For all other computations, the iteration bound can always be reached, with a high enough level of unfolding.

Tables showing the critical path, iteration bound, and formal degree of each SCC are presented (linear SCCs are preferred since they can more effectively be

optimized). Tables presenting the speedup for various values of the unfolding factor are presented. These tables are useful in selection of the unfolding factor.

TLU is an exceptionally effective optimization enabler. Its predicted enabling potential is inversely related to the formal degree of the computation.

Capacitance: while for throughput, TLU can be done freely, for power this is not the case. Typically, no more than a factor of 28 speedup [Cha95] is beneficial. Further, unfolding often incurs an overhead in control logic and thus control power. Due to the importance of operation count, TLU is favored when the computation is linear. In non-linear computations, subsequent optimization often results in exponentially increasing number of operations (with greater unfolding factors).

Name: **Common Sub-Expression Elimination (CSE)** [Mor79]

Command-line call: CSE -a Design

Trigger: “there is at least one common sub-expression”

Note: this is determined easily if the computation has no conditionals, otherwise it is an undecidable problem [Weg91].

Feedback:

No immediate effect on throughput.

Can potentially enable more effective subsequent optimizations, such as template matching. However, its potential for enhancing effectiveness is limited, for throughput.

Capacitance: Removes redundant operations thus power of execution units may be reduced. It is difficult to determine the amount of reduction that is possible.

CSE is often included towards the end of optimization sequences.

Name: **Common Sub-Expression Replication (CSR)**

Command-line call: Manual

Trigger: “there is at least one operation in a cycle that has fanout to at least two other operations” (if the operation is not in the cycle, then it can be pipelined)

Feedback:

No immediate effect on throughput.

CSR is an effective optimization enabler for throughput. It is especially effective in enabling algebraic transformations and retiming.

Enabling potential is in general inversely related to the computation’s formal degree. It is positively related to the number of operations with fanout that lie on the ϵ -critical network. Furthermore, the closer the critical path is to $\lceil \log_2(\text{NumStates} + 1) \rceil + \text{multDuration}$, the less further improvement is expected (*NumStates* is the number of sample delays in the computation), since this gives an indication of what can be achieved using algebraic optimizations.

Never end a sequence of optimizations with CSR.

Capacitance: This optimization is in general not effective reducing power. It can enable subsequent throughput improvements, however it increases the operation count. If the percentage increase in operations is low, then it may be worthwhile investigating.

Name: **Constant Multiplication Replacement by Additions and Shifts** [Rab91a or Pot94c] + **Module Selection** + **Clock Selection**

Command-line call: Multiplications -a Design; ModuleSelect -a -C -L "Library" -G "speed" Design

Trigger: "there is at least one constant multiplication"

Feedback:

Apply the script since constant multiplication replacement is fast (linear in the size of the number of constant multiplications).

NewCriticalPath = GetCriticalPath(NewDesign).

NewPower = GetPower(NewDesign).

Observations:

The resulting critical path in clock cycles often increases. However this depends greatly on the coefficient values and the multiplication operand bitwidths. Simple coefficients are replaced with just a few additions and shifts (e.g., multiplication by 2 can be replaced with a single shift operation). The greater the bitwidths, the greater the difference in delay of multiplication and addition/shift (multiplication delay grows quadratically with the bitwidth while addition or shift delays grow linearly). After replacing constant multiplications, it is possible that a better clock cycle utilization can be obtained.

Capacitance: expensive multiplications are replaced by lower power shifts and additions. The percentage of constant multiplications and the bitwidths are important factors in determining this impact. Operation count typically goes up. As a result, register accesses and thus also clocking power will increase. Finally, if hardware sharing of shifts is not done, then they can be eliminated, to further reduce operation count.

Name: **Maximally-Fast Script** [Pot92]

Command-line call: Manual

Trigger: "computation is recursive, and either linear or feedback linear"

Feedback:

$$\text{CriticalPathUsingMaxFast} \leq \lceil \log_2(\text{NumStates} + 1) \rceil + \text{multDuration} .$$

In the case where multiplications are expanded into shifts and additions, an analogous equation exists.

if (CriticalPathUsingMaxFast < CriticalPath)

 NewCriticalPath = CriticalPathUsingMaxFast.

Capacitance: Bounds on the resulting number of additions and multiplications are given by the closed-form equations presented in [Sri95a]. The computation uniformity, and thus area will increase significantly. This will impact interconnect capacitance.

Name: **Time-Loop Unfolding + Maximally-Fast Script** [Pot92]

Command-line call: Manual

Trigger: “computation is recursive, and either linear or feedback linear”

Feedback:

$$\text{NewCriticalPath} \leq (\lceil \log_2(\text{NumStates} + 1) \rceil + \text{multDuration}) / (i+1),$$

where i is the unfolding factor.

In the case where multiplications are expanded into shifts and additions, an analogous equation exists.

Resulting number of additions and multiplications are given by the closed-form equations presented in [Sri95a]. These affect both area and power.

Generate tables presenting the speedup for various unfolding factors.

Capacitance: Generate tables presenting the number of multiplication and addition operations and the predicted voltage scaling achievable, for various unfolding factors. Favor this optimization when the computation is linear rather than feedback linear, and when it has a lot of states relative to the number of inputs and outputs.

Name: **Associativity-for-Iteration Bound + Pipelining** [Hua94]

Command-line call: Algebraic -a -rRD -g -i Design; RetimeCP -ap Design

Trigger: “for at least one non-trivial SCC, on its ϵ -critical network, Trigger_a is TRUE;

Trigger_a : “there are at least 2 associative operations in sequence”

Feedback:

Associativity can reduce the iteration bound by moving operations out of the cycles. All operations outside of the cycles can then be pipelined.

$$\text{NewIterationBound} = \text{IterationBound} \cdot (1 - \alpha_g \cdot \text{RatioCoverage}),$$
 where RatioCoverage = percentage coverage of SCCs by associative pairs of operators, and α_g is an empirically derived constant between 0 and 1.

$$\text{NewCriticalPath} = \max(\lceil \text{newiteration bound} \rceil, \text{maximum operator delay}).$$

Capacitance: operation count remains unchanged.

Name: **Associativity-for-Critical Path** (with CSR and distributivity enablers) [Har89, Hua94]

Command-line call: Algebraic -a -g -c Design

Trigger: “there is at least one associative pair of operations on the ε -critical network”

Feedback:

Transforms from chain to tree structures.

$\text{NewCriticalPath} \geq \lceil \log_2 (\text{number of operations on the critical path} + 1) \rceil \cdot \text{AverageOperatorDelayRoundedUpToMultipleOfClockPeriod.}$

As the optimization has execution time proportional to the number of operations in the computation, it is also possible to directly apply it.

Capacitance: operation count remains unchanged.

Name: **For-Loop Unfolding**

Command-line call: Manual

Trigger: “there is at least one for-loop”

Feedback:

NewCriticalPath depends directly on the unfolding factor selected.

The effective critical path per iteration of the loop body can be reduced to the for-loop iteration bound.

Capacitance: With unfolding, the loop overhead (bound checking and update of the bound index) is reduced proportionally to the unfolding factor. Array index calculation may become more complex with unfolding. Further, an additional loop prologue may be required.

For-loop unfolding can enable subsequent optimizations since it exposes parallelism. Its predicted enabling potential is inversely related to the loop formal degree.

A distinct difference between for-loop unfolding and time-loop unfolding is that for loops can only be unfolded a finite number of times.

Name: **Eliminate Hardware Sharing**

Command-line call: Manual

Trigger: “TRUE”

Feedback:

$\text{NewCriticalPath} = \text{GetCriticalPathAssumingNoRegisters} \text{ BetweenOperationsAndSkippingShifts}(\text{Design}).$

(Without hardware sharing, the computation’s intermediate variables need not be latched and thus the critical path is not quantized into clock cycles.)

This value is an upper bound, since it does not take into account chaining effects and false paths.

Power may be reduced due to lower control overhead, localized communications, and hardwired shifts. See Wu's [Wu94] comparison of time-shared and non-timed shared implementations of a quadrature mirror filter. Implementation area is proportional to the size of the computation. Area can increase dramatically which might result in large interconnect capacitance. However, if communication can be kept local, this may not be the case.

Name: **Partition into SCCs using Pipelining** [Gue96]

Command-line call: Manual

Trigger: "the computation has more than one non-trivial SCC"

Feedback:

NewCriticalPath = longest critical path among all non-trivial SCCs.

Capacitance: operation count is unchanged.

SCC partitioning is a highly effective divide-and-conquer based optimization enabler. Not only is each part more manageable in size than the overall computation, the approach enables independent throughput optimization of each part, in accordance with its properties. Optimizations that were not applicable to the computation as a whole can often be applied to its parts. Its effectiveness depends on the number of SCCs and the formal degree of each.

Name: **Partition into Cones** [Dey92, Gue96]

Command-line call: Manual

Trigger: "the computation is recursive & the computation is neither linear nor feedback linear & there are at least two sample delays"

Feedback:

No immediate effect on throughput.

Replication results in increased computation size. The exact change in operation count can be found since it involves finding the transitive fanin, which is linear in the size of the cone.

Cone partitioning is an effective optimization enabler for throughput.

Capacitance: Since replication increases operation count, this optimization should be used selectively, and only if the gains in throughput outweigh the increased operation count. For example, it is suggested to only replicate cones which have some operations on the ϵ -critical network.

Name: **Locality-Based Partitioning** [Meh96a]

Command-line call: Manual

Trigger: “locality metric of the computation is $> \alpha_9$ ”

α_9 is an empirically derived constant.

Feedback:

Throughput remains unchanged.

Capacitance: Exploration of locality reduces accesses to longer global buses, reduces the number of used multiplexors, average bus length, and interconnect capacitance (in contrast to the non-partitioned Hyper implementations).

Name: **Latency-Throughput script, Single Input, Single Output Case** [Pot94a, Sri95b]

Command-line call: Manual

Trigger: “computation is linear & computation has a single input and single output”

Feedback:

NewCriticalPath = multiplication delay + addition delay.

Capacitance: The number of operations decreases.

Name: **Regularity-Based Assignment**

Command-line call: Manual

Trigger: “regularity metric of the computation is $> \alpha_{10}$ ”

α_{10} is an empirically derived constant between 0 and 1.

Feedback:

Throughput remains unchanged.

Capacitance: Exploration of regularity reduces accesses to longer global buses, reduces the number of used multiplexors, average bus length, controller logic, and interconnect capacitance at possible expense of additional execution units (in contrast to the Hyper scheduler [Rab91a]).