ALGORITHMS FOR HIGH LEVEL SYNTHESIS: RESOURCE UTILIZATION BASED APPROACH

by

Miodrag M. Potkonjak

Memorandum No. UCB/ERL M92/10

28 January 1992

ALGORITHMS FOR HIGH LEVEL SYNTHESIS: RESOURCE UTILIZATION BASED APPROACH

by

Miodrag M. Potkonjak

Memorandum No. UCB/ERL M92/10

28 January 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ALGORITHMS FOR HIGH LEVEL SYNTHESIS: RESOURCE UTILIZATION BASED APPROACH

by

Miodrag M. Potkonjak

Memorandum No. UCB/ERL M92/10

28 January 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ALGORITHMS FOR HIGH LEVEL SYNTHESIS: RESOURCE UTILIZATION BASED APPROACH

by

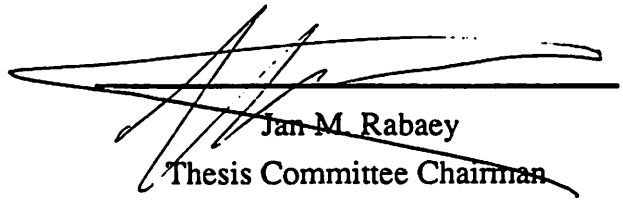
Miodrag Potkonjak

ABSTRACT

This thesis presents a new high level synthesis system, HYPER. HYPER uses a single, global quality measure, called the resource utilization measure, to drive the design space exploration process. This unique approach effectively merges the allocation, assignment, scheduling, transformations, and estimation with hierarchy handling in an unified manner.

Three HYPER parts are discussed: estimations, hierarchical graph allocation, assignment and scheduling and transformations. New transformation environment has been developed. Several new transformations (including retiming and associativity, software pipelining and software retiming, commutativity, and very fast implementation of recursive programs), based on a novel learning while searching and anti-voter rejectionless techniques, are introduced.

The effectiveness of the proposed algorithms is demonstrated in multiple ways: using standard benchmarks examples, with the aid of statistical analysis and through a comparison with estimated minimal bounds. Sharp minimal bounds based on a discrete relaxation technique are used.



Jan M. Rabaey
Thesis Committee Chairman

ACKNOWLEDGMENTS

Ever since I started my Ph.D. studies I regularly went through many thesis. Regardless of the vast variety of topics, the quality of results, the sizes and styles of presentation, applied and developed techniques, I was, by far, most puzzled by acknowledgment sections.

Many of them where long, and there were so many acknowledgments to so many persons, that it seems the only plausible conclusion is that the author actually did nothing, but collected work and ideas of many people. Or, that actually the author did a little bit something, but it was not of the greatest quality, and therefore should be backed with scores of "big names".

I am still going through various thesis and I am still most puzzled by acknowledgments. But what now puzzles me is no longer the size of acknowledgments. I have now realized that PhD research work is very difficult to be pursued in isolation and that a close interaction with other researches is almost prerequisite for a high quality work. What troubles me now is why majority of acknowledgments are so impersonal, almost as rewritten from some template, with appropriate insertion of names.

Anyhow, now is time that I write my Ph.D. acknowledgment. After reading it, maybe somebody will get some of just described impressions. Hopefully, the rest of the thesis will dismiss the doubts about the quality of technical contributions. And the rest of the acknowledgments maybe will not provide the most flattering viewpoint on my personality, but it is the accurate one.

First of all, I would like to thank my research adviser Prof. Jan Rabaey. He was supporting me, when I was writing "messy" code, when I was far behind any reasonable schedule, when I was not working because of midterms, final exams, prelims, because I was tired, "depressed", busy playing computer games (chess), or even because of my creative reasons such as "just being lazy". He was supportive even when I was making progress (at least I was thinking so) and bragged too much, too long. He did not just share his research vision, new ideas, and technical knowledge (UC Berkeley spoiled me so much, that I took this for granted), but also, whenever it

was necessary he debugged my code, wrote missing parts, translated from "Serbo-Croatian English" and "German English" to literate English, and made numerous counter-examples for my "new excellent" concepts and algorithms. He even survived some of my "ultimate" solutions for non-solvable problems. What to say now? Thank you, Jan. It has been a pleasure working with you.

If Prof. Bob Brodersen did not gave me a researcher assistantship before I arrived at Berkeley, most likely I would not attend this wonderful school. He also invited me to BJgroup (research group headed by Profs. Bob Brodersen and Jan Rabaey). Professors Bob Brayton and Leo Breiman served on my qualifying exam committee, as well as being the thesis readers. and taught me some of the most exciting topics in CAD and statistics. Professor David Messershmit served on my qualifying exam committee. Everybody who knows how busy Berkeley professors are, will understand why I appreciate their "volunteering". I would like to thank all of them and many other Berkeley professors who taught me fascinating topics in fascinating manner.

BJgroup provided a friendly and inspiring environment for study, research and even fun. I am grateful to all members of the group. I like to give special "thank you" to two special persons. Phu Hoang, who not only wrote many programs which where necessary for the success of my research, but has also been all the time a true friend. In the last couple of years, Anantha Chandrakasan was my favorite "conversation partner". He was the one of the first ones to use the HYPER tools and provided the most valuable feedback and some nice layouts which I included here. Also, discussions with him provided new interesting directions for high level synthesis and algorithm applications. Thank you, Phu. Thank you, Anantha.

My sister and parents, although thousand of miles away from Berkeley, always provided the most valuable moral support and cheering.

Last, but by no means least, I would like to thank Nada Aleksic. She helped me with almost all aspects of a graduate student life, including preparing posters, reports, papers (and in particular figures for papers), and this thesis. (Actually for the sake of truth, this sentence should be rephrased to "I helped her to prepare many of my posters, reports,..."). She also helped me to handle all the stress of graduate studies. And most importantly, she was the person who most and unlimitedly believed in me regardless of occasional temporary local minima. Thank you, Nada.

CONTENTS

1

INTRODUCTION

1.1 INTRODUCTION

During last three decades the number of devices in an integrated circuit has increased dramatically. The minimum feature size shrunk 11 percent per year, chip area increased 19 percent per year, and packing efficiency improved 2 times per decade [Bak90]. As the result, today multi-million transistor integrated circuits are widely available. It is imminent that further design complexity improvement will be achieved in future.

Although, the speed of the uniprocessor has grown, it is also approaching its limits. If not for technical and fundamental physics limit issues, then definitely for commercial ones. With the advent of multichip modules and further differentiation of memory hierarchies, exploiting concurrency becomes simultaneously both a very important as well as a difficult task.

Today there is a consensus that computer aided design tools are unavoidable in the design of high performance high complexity computing devices. The obvious, but necessary, requirement for the application of a new CAD tool, is that it performs as well or better than a human designer. The primary vehicle to achieve such CAD tools are new concepts and new algorithms.

The process of the design of integrated circuits from architecture primitive specifications (execution units, memory units, interconnects and controller structure) is relatively well understood. High quality tools are readily available [Shu91, Gaj87]. However, the algorithm and architecture design from application descriptions still remains a difficult task. This problem is particularly acute, when there is a need for a high level of parallelism.

The parallelism exploration for fixed hardware configurations is already a very difficult problem. Even for well understood problems, resource utilization is rarely higher than 5 to 10% [Don91]. However for custom hardware, new degrees of freedom in the selection and allocation of hardware make the problem even many times more difficult.

Therefore, we can conclude that the path from the high level specification of an application specific design to the implementation in architectural primitives is long and complex. It involves a multitude of interrelated problems, including the selection of clock frequency and hardware module set, partitioning, transformations, resource allocation, assignment and scheduling and hardware mapping. All those problems are computationally very complex, especially when posed with all constraints and all aspects imposed by real-life problems.

In the last decade, an impressive effort and significant progress has been made in the area of high level synthesis [McF91, Cam91]. There are a number of systems which provide a more or less convenient way for the automatic synthesis of relatively small designs. The application of optimizations is however often limited and local. The major features which distinguish HYPER from other high level synthesis systems are the organized application of transformations, an

extensive use of sharp estimation techniques, and a global search optimization mechanism organized through a system manager.

The research presented in this thesis got its impetus from the requirements for extensive optimization as required in HYPER. The thesis has two goals: to present the algorithms which have been integrated in the HYPER high level synthesis system, as well as to discuss some of those algorithms as generic optimization techniques.

1.2 OVERVIEW OF THE THESIS

The thesis is organized in the following way. Chapter 2 describes the high level synthesis system HYPER and its software and design flow architecture. Also, in order to provide complete HYPER picture, parts contributed by other HYPER researchers, namely the Silage Translator, Behavioral Simulation, Module Selection and Hardware Mapping, are briefly described.

The major technical contributions of this thesis: estimations, hierarchical graph allocation, assignment and scheduling as well as several transformations are presented in Chapters 3, 4 and 5. Chapter 6 provides a different, more global viewpoint, describing a novel probabilistic rejectionless anti-voter algorithm. A summary of all those contributions will be presented in the rest of this introduction. Finally in Chapter 7, we will draw some conclusions and outline possibilities for further research.

1.2.1 Implementation Complexity Prediction of Signal Processing ASICs

In the design space exploration phase of the behavioral synthesis process (as conducted in the HYPER system), two types of information can be used to steer the design solution space search: feedback and prediction. While feedback information is obviously more precise, it is often computationally expensive, since it requires full cycle through the design process. Therefore, prediction tools are essential. Many other attractive applications for the use of prediction

can be defined, such as the design of high level synthesis algorithms (e.g. scheduling, assignment, transformations), objective function definition, and benchmark design.

The complexity of a signal processing algorithm is usually expressed by the number of operations executed during the algorithm. Very often, a significant emphasis is placed on the number of multiplications. However, such measures are poor predictors of the ASIC implementation cost of the algorithm. For example, important parameters such as the memory and interconnect requirements as well as the critical path of the algorithm are neglected.

In the Chapter 3, the following implementation complexity prediction problem is addressed: Given an arbitrary real time signal processing algorithm represented by its control data flow graph, determine sharp minimum and maximum bounds on its implementation cost in terms of the required number of execution units, interconnect and memory.

Two crucial properties determine the success of prediction tools: the accuracy and the low computational cost. For example, a computationally efficient minimum bound on the required number of executional elements can be obtained by observing that, given a number of resources of class R_i (N_{Ri}), at most $N_{Ri} \times t_{max} / L_{Ri}$ operations can be performed on those resources (with L_{Ri} the length of a single operation). The required number of operations (O_{Ri}) can be easily derived from the control data flow graph, resulting in the following lower bound on N_{Ri}:

$$N_{Ri} \geq \frac{O_{Ri} \times L_{Ri}}{t_{max}}$$

Much more sophisticated and sharper lower bounds can be derived using approach presented in Chapter 3. In HYPER, we have established techniques to accurately predict both max- and min-bounds on the different hardware resources, based on a so called **discrete relaxation** technique. The relaxation approach turns an NP-complete problem into a polynomial one by

relaxing some of the constraints. Then the optimum solution for the polynomial complexity problem can be used as a bound for the initial NP-complete problem.

For instance, the number of adders needed to implement a given algorithm can be accurately estimated by scheduling only the additions and temporarily ignoring precedence relationships between the operations. This problem can be optimally solved in $O(N^2)$ time. The Chapter 3 presents a spectrum of possibilities, for relaxing the scheduling and assignment problem into a variety of problems of polynomial complexity.

1.2.2 Allocation, Assignment and Scheduling Algorithms

Scheduling is defined as the task, which decides in which control step a given operation will happen. Assignment determines on which particular execution unit a given operation will be realized, from which register it will request data and where it will send the result using which connection. Resource allocation is closely related to the above tasks: it reserves the amount of hardware (in terms of execution units, memory registers and interconnect) necessary for the realization. It might also set the allotted time, available for the execution of the algorithm. Obviously, those three tasks are interdependent.

Almost all scheduling and assignment problems, even when posed in a highly restricted form, are at least NP-complete. Although high level synthesis is a relatively young area there are numerous approaches towards solving those problems. Those approaches can be categorized in several groups: manual and brute force approaches, various heuristics which use as soon as possible (ASAP) and as late as possible (ALAP) scheduling to obtain a global picture of the solution space, integer programming, various probabilistic approaches (e.g. simulated annealing and neural nets) and continuous relaxation techniques (e.g. linear programming and gradient methods).

There exist more than 70 published approaches, but some aspects of the problem have not been adequately addressed. First of all, in VLSI technology it is essential to simultaneously address all three components of the cost function (being the number of execution units, memory registers and interconnect). Very few scheduling and assignment algorithms are doing this. Furthermore, it is necessary to consider during the scheduling not only the structure of the control data flow graph but also the available hardware and its properties. It is obvious that scheduling for two different technologies which, for example, have different hardware cost for the same functional unit, can be drastically different. Thirdly, it is important that, once a hardware unit has been selected, the utilization of that device should be maximized during the assignment and scheduling phases.

A major conceptual difference relative to published scheduling and assignment algorithms is that we first do assignment and then scheduling. We succeeded to characterize a possible assignment with a simple quality measure, which predicts the changes to find a successful schedule for this assignment. Once an assignment is accepted, scheduling is performed using the resource utilization as a priority function. We are always trying to schedule first those operations which relax constraints on critical resources (execution unit, interconnect, or register). A critical resource is a resource which is in large demand and short supply. (Due to precedence and timing constraints some resources can not be used during some control cycles)

The algorithms have been tested on a wide variety of examples, and performed better or at least as good as other algorithms using only a fraction of the time required by those algorithms. A detailed description of the techniques can be found in the Chapter 4.

1.2.3 Behavioral Transformations for the Synthesis

To solve a given DSP computational problem, one can use a large number of algorithms. Often any one of these algorithms can lead to several implementations, each with vastly different

execution times, hardware requirements, power constraints, testability and other parameters of interest. The selection of the algorithm best suited for the optimization of those objectives is a crucially important task in the design process of high performance DSP ASICs. An equally important task is to ensure that the potentials of a given algorithm are maximized. This is achieved through the application of optimizing transformations. To maximize effectiveness it is crucial that transformations will be globally optimized.

Transformations are changes in control data flow graph structure which improve the final implementation, without altering the input-output relationships. Most of the behavioral transformations have been introduced in the field of software compilers. They include constant arithmetic, common subexpression elimination and value numbering. The most important among them are the loop transformations, such as loop retiming, software pipelining, loop jamming, partial and complete loop unrolling and strength reduction. Those are especially suitable for real-time systems, in which a program always contains an infinite loop over time and concurrency can be exploited more efficiently. Although a majority of those transformations are well known from the software compiler literature, an attempt to apply them to high level synthesis poses specific challenges. Furthermore, two additional degrees of freedom can be observed: hardware parallelism and hardware definition. Chapter 5 describes the application of several transformations in a high level synthesis environment: retiming, pipelining, software retiming and pipelining, commutativity and fast implementation of recursive programs. They are briefly introduced bellow.

1.2.3.1 Retiming for synthesis

Retiming is a conceptually simple and powerful transformation which has been successfully applied in several areas of design synthesis and automation. The goal of the retiming transformation is to move delays (which can be either clocked delays in a circuit or algorithmic delays in a behavioral flow graph, depending upon application) such that a certain objective

function is optimized. Until recently, the objective function has been exclusively the critical path or the number of delays in a graph or a circuit. However, the potential of retiming is significantly higher. We have developed a new formulation, this time targeted towards behavioral synthesis: given a signal flow graph, retime it in a such a way that the resulting signal flow graph will have a minimum hardware cost, while still satisfying all timing constraints. Since, the implementation with minimum cost has the most efficient resource utilization we call this transformation: retiming for *efficient resource utilization.*

While the traditional retiming optimization problem is of polynomial complexity, we proved that retiming for efficient resource utilization is an NP-complete problem. In order to maximize the effectiveness of retiming, we combined it with associativity. We also developed a probabilistic algorithm for efficient resource utilization algorithm [Pot89a, Pot91a]. The algorithm has been tested on a number of examples. Those examples show the distinctive advantage in high level synthesis of retiming for scheduling over retiming for minimal critical path. Another major advantage of the proposed algorithm is that other transformations, such as associativity, pipelining and software pipelining, can be easily combined with the retiming operation. We continue the investigation for farther improvements in algorithm performance and additional generalization of the retiming operation.

1.2.3.2 Pipelining

Pipelining is probably the most often used technique for throughput improvement in ASIC design. Despite its popularity, pipelining is also the transformation which is most often incorrectly applied. For example, a number of authors are ignoring constraints imposed by recursion in computation. Also, pipelining almost always has a side effect, namely an increased demand for registers. Since the cost of registers is of the same order of magnitude as for example an adder (1/3 to 1/2 of the area cost), savings in execution units can be wasted due to memory overhead.

Pipelining can be defined as a retiming where additional delays are available on all input or output edges. Once pipelining is formulated in a such manner the perception often emerges, that pipelining is hardly different then retiming with an increased number of delays. However, the increased number of delays, and their initial location on input (or output) edges impose important demands on optimization algorithm. There are two major questions: to what extent to pipeline an algorithm, and what is the ultimate pipelining potential. Those two questions are discussed in the pipelining section. Again, numerous empirical results are presented.

Retiming and pipelining are based on exploration of parallelism in a loop over time. Of course, they can be applied also when the source of parallelism is a program control loop. In this case pipelining is most often called software pipelining [Lam88, Lam89]. Although the retiming and pipelining algorithms can also handle software retiming and pipelining, in the later case it is important to take into account the transformation overhead, especially when the number of loop iteration is small.

1.2.3.3 Commutativity

Commutativity is probably the most widely known and the simplest control data flow graph transformation. However, unfortunately, commutativity is only conceptually a simple transformation. In Chapter 5, it is proven that even in a very restricted case when there is only one type of operation in control data flow graph which possesses the commutativity property, the optimal application of commutativity for the maximization of resource utilization is an NP-complete optimization problem.

It is obvious that the application of commutativity can result in savings in required number of registers and interconnect. In Chapter 5, it is also shown that indirectly it can also improve the resource utilization of the execution units. Although commutativity does not result in as a spectacular design improvement, as produced by some other transformations, its usefulness is ampli-

fied by two facts: (1) it has a broad range of applications (it is applicable not only on additions and multiplications, but also on *and*, *or*, *max* and *min* operations), and (2) its application is to the greatest extent orthogonal to the effects of other transformation, so that it can be applied as the last transformation before allocation, assignment and scheduling without a negative influence on the effectiveness of other transformations.

Commutativity is especially interesting as an optimization problem. It is straightforward to enumerate the size of the solution space: when we have n commutative operations in the control data flow graph, there exist 2^n different possible solutions. Also, the topology of the solution space is exceptionally well structured: it is *an* n-dimensional cube. Furthermore, it is easy to show when all operations are commutative that the solution space is symmetric. All those properties make the commutativity for resource utilization an excellent testbed for the study of combinatorial optimization algorithms. The commutativity section in Chapter 5 is concluded by a more elaborate study of commutativity from an optimization viewpoint.

1.2.3.4 Fast implementation of arbitrary recursive programs

Signal processing applications often tend to push the throughput requirements to the edge of what can be possibly achieved for a given algorithm. Reaching the throughput requirements (while still keeping the hardware cost in reign) is then the most important optimization goal.

When the algorithm at hand does not display any recursion (feedback), this goal can be achieved by pipelining the algorithm to the extent needed. One should, however, be aware that over-pipelining should be avoided, as (as mentioned higher) registers have a non-ignorable hardware cost. The optimal positioning of the pipeline registers can be easily determined using the pipelining for resource utilization transformations, which determines the position of the registers in a such a way that the estimated hardware cost is minimized, while still meeting the throughput constraints.

Most of the signal processing algorithms however have internal recursion. Examples of such programs include both relatively simple cases, such as infinite response and adaptive filters, and more complex ones, such as systems solving non-linear equations and adaptive compression algorithms. Graphs involving recursions display an upper bound on the computation rate, called the pipeline stage bound. This pipeline stage bound is given by $T_{pipe} = \max\left(\frac{T_l}{ND_l}\right)$. The maximum is taken over all loops l, T_l is the sum of the computations times of all nodes in loop l, and ND_l is the number of delay elements in loop l [Mes88].

Several researchers addressed some special program instances (e.g. IIR filters) and achieved a significant progress in reducing the pipeline stage bound [Mes88, Par89a, Par89b, Lin91, Fet90]. Our goal is to find an approach that will automatically transform arbitrary recursive programs into a form where the pipeline stage bound is reduced to a minimum. This can be achieved in a dual way: reducing T_l by applying algebraic transformations (associativity, commutativity and distributivity) and increasing ND_l by moving delays (retiming). The latter task is non-straightforward and requires partial unrolling of the outer loop.

It is interesting to note that when only retiming or only algebraic transformations are used, the problem can be explicitly solved. Leiserson and Saxe developed a polynomial algorithm for retiming for critical path [Lei83]. Valiant [Val83] and Miller [Mil88] described efficient algorithms for the reduction of expression-tree height when only algebraic transformations are used. Although they primarily discuss the algebraic transformations, for the case where the computational structure is a ring, the results are valid for many instances of fields as well. We have proven however that, when both types of transformations are combined, the problem becomes NP-complete (even for a very restricted formulation of the problem, where the flow graph only contains additions and delays).

A simple and efficient algorithm for fast implementation of recursive programs is discussed in Chapter 5. The effectiveness of this approach is illustrated on several examples.

1.2.4 Probabilistic Rejectionless Anti-Voter Algorithm

While all other chapters of this thesis are mainly involved with high level synthesis problems, and the HYPER system, Chapter 6 has a different scope and a more general view.

Several new algorithms are proposed and used for solving high level synthesis tasks in HYPER, including learning while searching and partial relaxation. It seems that all of them are rather general optimization techniques. While during development of the HYPER system, the major accent was on the development of fast, high performance custom algorithms for specific tasks, this Chapter is devoted to the in depth exploration of the novel algorithms.

We discuss one of them, the Probabilistic Rejectionless Anti-Voter (PRAV) algorithm in much more detail. While in HYPER it is used in the commutativity and in the assignment subroutine, we discuss its application to generic NP-complete problems in this Chapter. For one problem from this class, graph partitioning, we present empirical results and a comparison with other general purpose hill-climbing approaches, such as simulated annealing and Kernighan-Lin iterative improvement technique.

1.2.5 Conclusion and Future Research

In this Chapter we will draw some conclusions and briefly discuss the relationship of presented techniques with other research areas. Some directions for further research will be outlined. This includes ASIC background memory and I/O design and optimization, design for low power, benchmark design and algorithms architecture and algorithms selection and algorithm design.

2

HYPER - HIGH LEVE SYNTHESIS SYSTEM FOR NUMERICALLY INTENSIVE APPLICATIONS

2.1 INTRODUCTION

Due to the excessive computational requirements of many tasks in Digital Signal Processing (DSP) applications (such as sonar, radar, speech, image, video processing and numerical algebra and analysis based computations), special purpose hardware is often needed for their implementation. In order to reduce the time to market, substantial efforts have been devoted in the last decade to automatically generate this type of implementations from higher level specifications. Silicon compilation, which automatically map an architectural description into silicon (or any other hardware platform) are now commercially available and are widely used in industry [Shu91, Gaj87]. However, the most time consuming part of the synthesis process is located in the architectural synthesis process, which maps the behavioral description of the algorithm into a suitable architecture.

This area of computer aided design process is often called high level synthesis or behavioral synthesis [McF90, Cam91]. A wide class of behavioral synthesis tools has been reported in literature [Cam91]. Typical elements of those environments are hardware allocation (determination of the amount of hardware needed), hardware assignment (binding operators to specific hardware instances) and operation scheduling (specification of the instruction cycle in which an operator will be executed).

Numerically intensive signal processing implementations (such as mobile telephone, personal communications networks, speech and image recognition, HDTV) are usually bound by a number of real time constraints, which impose the requirement that the algorithm is executed within a fixed time period (often called the sample or frame rate). Due to these requirements, the implementation of a signal processing algorithm has to be close to optimum. This not only requires excellent schedulers and hardware allocation programs, but also needs an elaborate set of optimizing transformations to ensure that the algorithm description itself is close to optimum.

2.2 THE HYPER SYNTHESIS ENVIRONMENT

The HYPER, developed at the University of California, Berkeley, is a high level synthesis systems for numerically intensive applications. Its goal is not only to provide an automated path from high level language description to silicon implementation, but also to provide efficient tools for the design space exploration and optimization mechanisms which are most often beyond the scope and power of integrated circuits designers.

The synthesis process requires the execution of many translations, operations and/or transformations. Figure 1 displays the elements of the HYPER system. The real-time application is described in Silage, a signal-flow-graph language [Hil91]. HYPER parses and compiles this description into an intermediate control data flow graph database (CDFG). The CDFG represents

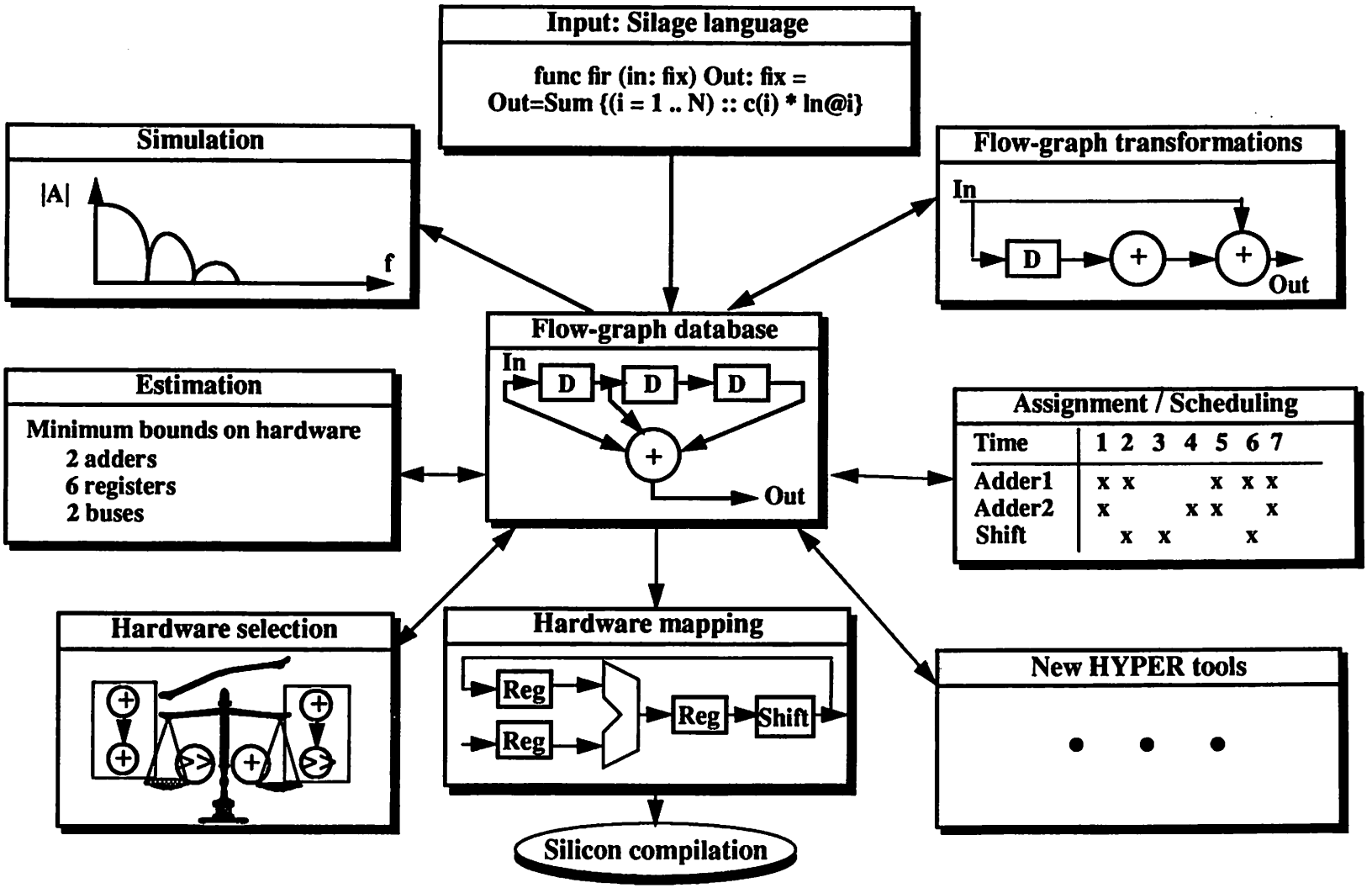


FIGURE 2.1. The HYPER Software Architecture

the algorithm as a dataflow graph, extended with some macro control flow statements such as loops and if-then-else structures [Hoa92].

This graph serves as a central repository on which all synthesis operations, such as complexity estimations, flow-graph transformations, and hardware allocation and scheduling are applied. The results of those synthesis operations are back-annotated onto the CDFG database. As a result, HYPER has a very modular software organization, and new tools are easily integrated into it.

At each point of synthesis process, HYPER can generate a simulation model of the control data flow graph, so that correctness of the executed synthesis operations can be verified. At the same time, this simulation enables the checking of the synthesis operation influence on performance parameters, such as the signal-to-noise ratio and the numerical stability, as a function of the required word length.

Most of the high level synthesis tasks are NP-complete or even harder from a computational complexity viewpoint. Furthermore, they are interdependent and ordering of those synthesis operations has a profound effect on the final solution. For example, the quality of the final solution is very sensitive to the ordering of the CDFG transformations.

In order to handle this high complexity, HYPER implements the overall synthesis procedure as a search process. Starting from an initial solution, obtained using min bound estimations, HYPER proposes new solutions by executing a number of basic moves, such as adding or removing of hardware resources, changing the time allocation for different subgraphs in the algorithm, or by applying an optimizing graph transformation.

The assignment module checks the feasibility of the proposed solution and determines the cost of a proposed solution, by binding instances of CDFG nodes to instances of architecture elements. The synthesis manager manages the overall search and synthesis process and decides

either interactively or automatically what move to perform next. In the automatic mode, it bases this decision on the results of the estimation process and the feedback information from the scheduling or transformation modules on bottlenecks and problem areas.

The synthesis manager is the single most important feature of HYPER system and distinguishes it from a multitude of proposed synthesis systems [Cam91]. Throughout the exploration of the design space, HYPER uses a single global quality measure, called resource utilization. This unique approach merges synthesis operations such as transformation and allocation as well as the handling of hierarchy in a consistent fashion.

Once HYPER arrives at an acceptable solution, it stops the search and maps the solution onto a hardware architecture. After that using silicon compilers the silicon can be generated. In HYPER, the Lager IV silicon assembler is used. [Shu91]

Figure 2 displays a common action scenario. The user interface and the information feedback provided by HYPER tools are described during the description of corresponding tools. To demonstrate the capabilities of the HYPER tools we will use the example shown in Figures 3 and 4. Figure 3 shows the Silage description of 7th order IIR filter. Figure 4 shows the layouts of some implementation generated by HYPER and Lager IV, for various throughput requirements. Run times for all HYPER tools on those examples is around 1 minute per example. High level synthesis tools, including compilation process and application of several transformations is less than 10 seconds on Sun 4/100. Layout generation takes around 1 hour.

The estimation, transformation, allocation, assignment and scheduling modules are described in a detail in the following chapters. Here, we will briefly outline other HYPER parts: behavioral specification, module selection, and hardware mapping. The primary author of the first part is Phu Hoang, and the latter two Chi-Min Chu. Those parts are described in a detail elsewhere [Chu91, Hoa92].

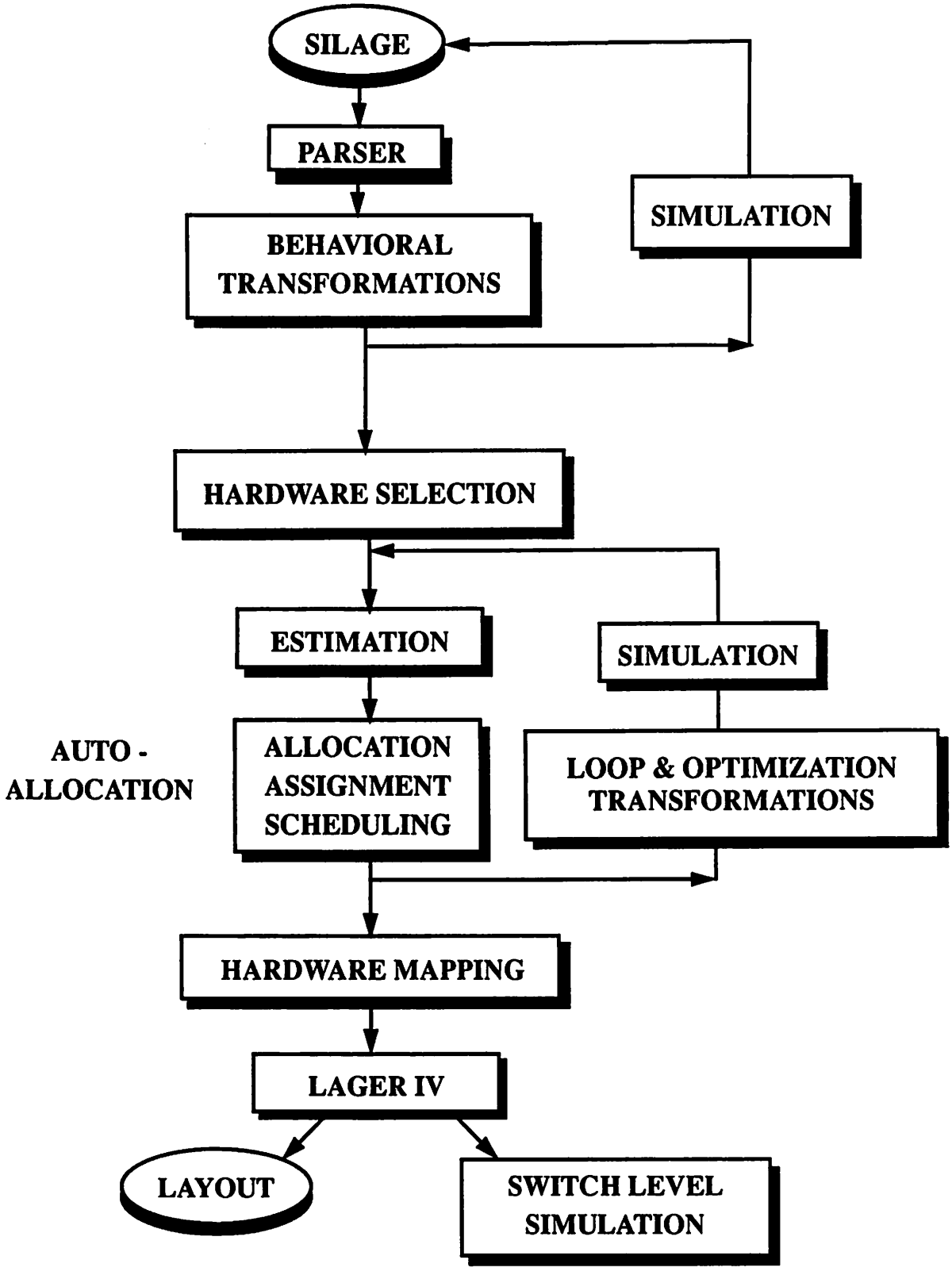


FIGURE 2.1. Scenario of a typical HYPER session

```
#define num16 fix<32,8>
#define Coef0 0.001953125
#define Coef1_1 -1.3125
#define Coef1_2 0.625
#define Coef1_3 1
#define Coef1_4 1
#define Coef2_1 -1.25
#define Coef2_2 0.75
#define Coef2_3 0.0625
#define Coef2_4 1
#define Coef3_1 -1.125
#define Coef3_2 0.921875
#define Coef3_3 -0.25
#define Coef3_4 1
#define Coef4_1 -0.71875
#define Coef4_2 1

func main (In : num16) Out : num16 =
begin
  In1 = num16(In*Coef0);
  In2 = biquad(In1, Coef1_1, Coef1_2, Coef1_3, Coef1_4);
  In3 = biquad(In2, Coef2_1, Coef2_2, Coef2_3, Coef2_4);
  In4 = biquad(In3, Coef3_1, Coef3_2, Coef3_3, Coef3_4);
  Out = firstorder(In4, Coef4_1, Coef4_2);
end;


func biquad(in, a1, a2, b1, b2 : num16) : num16 =
begin
  state@@1 = 0.0;
  state@@2 = 0.0;
  state = in - (num16(a1*state@1) + num16(a2*state@2));
  return = state + num16(b1*state@1) + num16(b2*state@2);
end;

func firstorder(in, a1, b1: num16) : num16 =
begin
  state@@1 = 0.0;
  state = in - num16(a1*state@1);
  return = state + num16(b1*state@1);
end;
```

IGURE 2.2. An example of the Silage description: 7th order IIR Filter

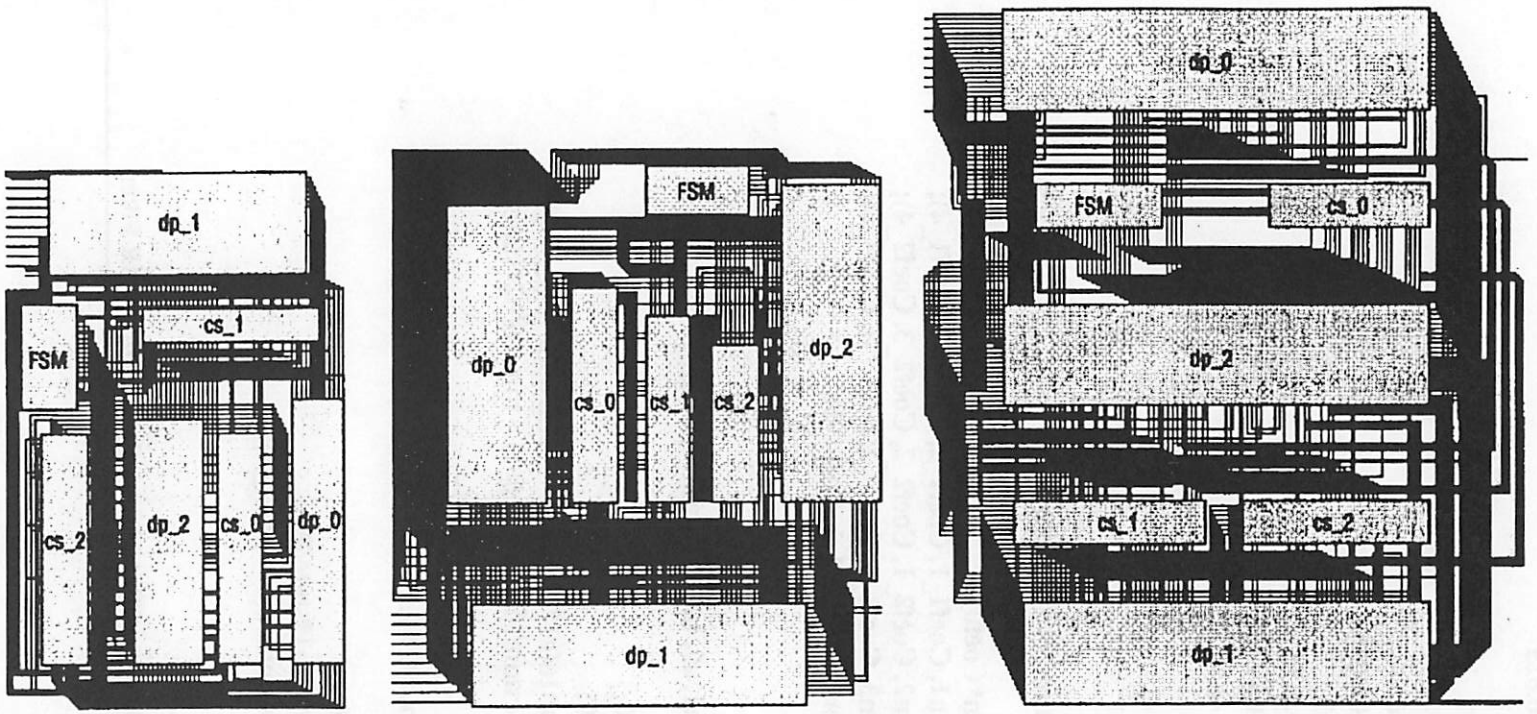


FIGURE 2.3. Examples of layouts generated using HYPER: 7th order IIR Filter for three different throughputs.

2.3 BEHAVIORAL SPECIFICATION

A proper algorithm representation is crucial to the performance of any synthesis environment. The representation should allow for efficient synthesis, regardless of whether the description is dataflow oriented, control flow oriented, or a combination of those two extremes. Information on the algorithm's data flow suitably exposes all the available parallelism in the algorithm. The available parallelism has the most profound effect on area/performance tradeoffs. However, in order to design a fast and area efficient control unit, insight in the overall control flow is also necessary. For those reasons, the basic computational model in f HYPER is a mixed control data flow graph (CDFG).

The CDFG represents the algorithm essentially as a flow graph, with nodes, data edges, and control edges. The nodes represent data operations, while the data edges represent data precedence between those nodes. In addition, control edges can be introduced to enforce extra precedence rules between nodes. They provide an efficient mechanism for the explicit representation of superimposed timing constrains on nodes. For example, we can say that the execution time of operation A has to trail the execution of operation B by at least N cycles.

Aside from standard arithmetic and Boolean operations, the CDFG allows for a number of macro control-flow operations such as loops and if-then-else's. By using these control statements, we can handle a hierarchical graph whose subgraphs represent the bodies of loops or conditionals. The subgraphs contracts into a single node at the next hierarchy level. This hierarchical representation is both compact and descriptive. It also stores the control data flow graph efficiently. Finally, it provides a way for clean definition of the algorithm's macro control flow, which results in more efficient control structures.

HYPER stores the flow graph in the Oct database [Har86, Cas91]. The Oct database is extended with a versioning mechanism to track the subsequent phases of design process.

HYPER also provides a mechanism for the interactive graphical presentation at every point during synthesis, using a schematics placement and routing tool.

Silage, the HYPER input verification language, is a signal-flow language developed at the University of California, Berkeley, especially for the specification of digital signal processing algorithms. HYPER uses as input an application algorithm presented in Silage [Hil85]. Silage code is translated in the Silage-To-Flow translator into CDFG with essentially the same hierarchical structure. During this translation, several standard, architecture-independent transformations, such as dead-code elimination and manifest expression calculation are applied.

Figure 3 shows the Silage description of a seventh-order biquadratic IIR filter. The filter is composed of three cascaded biquads and one first-order section.

Simulating the algorithm is an important part of synthesis. Simulation verifies the functionality of the algorithm and the correctness of the applied transformations. It also provides a mechanism to verify numerical stability and - word length trade-offs. The simulation generator translates the CDFG description into executable C code. It generates two simulation modes: (i) using floating-point data types, and (ii) using model with fixed point entities. The floating-point mode offers quasi-infinite precision, while the fixed point mode uses the exact data type defined in Silage and therefore allows for the modeling of truncation and rounding effects. In this way, we can accurately model the noise and distortion behavior of the system at the every phase of the design.

2.4 MODULE SELECTION

There are many ways to order high level synthesis tasks during synthesis. HYPER does module selection very early in the synthesis process. This decision is based mainly on the idea to leverage on high quality estimation routines.

The hardware selection addresses three subtasks:

(i)selection of the clock period, if the user does not provide this information;

(ii) selection of proper hardware modules from the hardware database;

(iii) combination of primitive hardware modules into more complex combinational elements, in order to minimize a number of variables which have to be stored in registers and to increase the resource utilization.

The goal during hardware selection is to select those hardware primitives, which will facilitate the resource utilization optimization during consequent design steps. HYPER tries to select those modules which are simultaneously area efficient and have strong chances for good resource utilization. It is important to optimize the resource utilization inside each control step, which can be achieved by addressing of subtasks (i) and (iii) appropriately. Resource utilization over the total available time is predicted using estimation modules and optimized by addressing the last two subtasks. During the creation of the complex modules, their timing characteristics are predicted using a ripple model [Chu91]. This model characterizes a functional block using three parameters: a ripple direction, a ripple delay, and a one-bit delay. This model enables a good compromise between the accuracy of the estimation and the required computational effort.

A detailed description of the hardware selection module can be found in [Chu91].

2.5 HARDWARE MAPPING

Hardware mapping is the step in the design synthesis which provides the connection between high level synthesis and structural design. In the HYPER system the input to the hardware mapping is the allocated, assigned and scheduled flow graph, called the decorated flow graph. The output is a structural description of the architecture in a structural description language, called SDL [Shu91]. During the mapping, the decorated flow graph is translated into three structural graphs: the datapath-structure, the controller state-machine graph, and the inter-

face graph. After that, dedicated tools translate each of those subgraphs into corresponding structural views.

During the hardware mapping, several optimization steps are executed. A detailed description of the hardware mapping tool and the optimization steps can be found in [Chu89, Chu91].

Currently the hardware mapping targets a macrocell library. There is an ongoing effort to provide mapping to standard cells and sea-of-gates [Rab91a], as well as to PADDI devices [Che90], which are specialized, field programmable devices for data-path prototyping.

2.6 CONCLUSION

In this Chapter we briefly described the high level synthesis system HYPER. Also, references to more detailed description are given.

3

ESTIMATING IMPLEMENTATION BOUNDS FOR REAL TIME APPLICATION SPECIFIC CIRCUITS

3.1 INTRODUCTION

At numerous times in the design process of a real time application, important decisions have to be made, which might affect the quality of the final solution in a dramatic way. Unfortunately, most of these decisions are currently made on an ad hoc base, since evaluating the effect of a decision requires a complete run through the design process and is therefore extremely time consuming. The advent of high level synthesis helps to alleviate this problem, as it allows for a much faster traversal of the design cycle. However, the majority of the high level synthesis tasks, such as optimizing transformations, allocation, assignment and scheduling have been proven to be at least NP-complete. To solve the problems in polynomial time, numerous heuristic as well as probabilistic solutions have been proposed. As a result, it is hard for a designer to qualify the solution, obtained with a particular synthesis environment. Furthermore, even though faster than

the manual approach, running through the complete synthesis cycle still takes a substantial amount of computation time and is hence not effective for traversing the global design space.

The problem could be alleviated considerably, if estimations of the implementation complexity of an application could be made fast and accurately. Currently, the complexity of an algorithm is usually expressed by the number of operations, required during the execution of an algorithm. Very often, a significant emphasis is placed on the number of multiplications. However, such measures are poor predictors of the ASIC implementation cost of the algorithm. For example, important parameters as the distribution of the operators over the algorithm, the critical path and the cost of memory and interconnect are neglected.

This chapter presents a set of techniques to accurately predict the computational requirements of an algorithm, given the algorithmic flow graph and the maximum execution time. A technique called *discrete relaxation* is introduced to establish sharp minimum and maximum bounds on all hardware resources. The computed bounds can be used for a myriad of purposes in the design synthesis process.

- The derived minimum and maximum bounds delimit the search space, thus speeding up the design synthesis search process.
- The minimum bounds can serve as an initial solution for the above search process. We have experienced that this solution is often very close to the final solution.
- Most synthesis tasks are optimization tasks, attempting to minimize a cost function, which is very often the implementation area (or complexity). The accuracy of this cost function will directly influence the quality of the synthesis process. Accurate estimations can therefore help to boost synthesis performance.
- Estimations can help to establish the relative or absolute quality of a proposed solution. This information is extremely useful to direct the overall synthesis search process. For instance, in an iterative transformation environment, it is important to know

how much a proposed transformation will influence the implementation cost, hence establishing a relative ordering of the candidate moves. It is also useful to know the maximum improvement to be expected from a transformation.

- Estimation of the optimal solution can help to determine the absolute qualify of a synthesis algorithm (such as a scheduler or a transformation). Since most algorithms are at least NP-complete, benchmarking (with all associated traps) is the dominant approach at present.
- Finally and most importantly, estimations can play a crucial role in the algorithm and architecture selection and partitioning processes, topics which are by and large unexplored territory at present.

After a brief discussion of the previous work in this area and a description of the global estimation framework, the proposed techniques for the estimation of maximum and minimum bounds will be discussed in detail. The effectiveness as well as the application domain of the estimation techniques will be demonstrated with a number of examples.

3.1.1 Previous Work

While the idea of estimations is relatively new in the high level synthesis world, the concept of complexity prediction has been around for quite some time in the areas of compiler design, performance modeling, operational research, computer science, VLSI and digital signal processing.

Until relatively recently, there was little published work in the software compiler literature regarding estimations and their use. This is easily explained by the fact that the designers of software compilers are as much concerned about compilation speed as about execution speed. Therefore, the most complex scheduling algorithms used are of a quadratic complexity, which is as fast as any non-trivial estimation technique can achieve. With the introduction of vector computers [Kuc72, Ban79] and particularly after the introduction of very long instruction words [Nic84], super-scalar and super-pipeline architectures [Jou89, Smi89], numerous studies have

been published on the available parallelism in both general purpose and numerically intensive computations. The scope of those papers is quite different form the one addressed here: their main goal is to demonstrate that a particular class of algorithms has sufficient parallelism for a given architecture. This is measured by explicit scheduling of the algorithms on the target architecture.

Performance modeling, and benchmarking particularly, got a lot of attention recently [Hei84, Jai91]. The major concern here is to predict the average performance of a general purpose machine for a particular group of users. The most often used techniques are simple statistical models, build either manually or with the help of statistical packages, which use as input parameters the run-time results of a set of benchmark programs, typical for a particular area such as linear algebra or databases.

The complexity theory, as an area in theoretical computer science, studies the implementation complexity of an algorithm, assuming that all operations have the same cost. The major concern here is the asymptotic behavior of an algorithm, when the problem instance size goes to infinity [Pap82, Com90]. This is of very limited interest in the complexity prediction of ASIC implementations.

The work in the area of operational research and theoretical computer science that most resembles the problem addressed here, is in the framework of approximation algorithm development [Hoc87]. The goal is to develop algorithms for a particular NP-difficult scheduling problem, which guarantee that the solution will be within ε percent of the unknown optimal solution. Although those algorithms have a polynomial complexity, their complexity order is most often high. The goal is once again different: explicit solution generation versus prediction.

The implementation complexity is very important in the area of digital signal processing and therefore discussed extensively [Bla85]. The pre-dominant measure used to express the

complexity on an algorithm is the number of operations, often with stress on the number of multiplications. The regularity of an algorithm has also become an issue recently.

In the VLSI arena, the theoretical computer science has made some significant progress in study of lower bounds on area and time of elementary circuits such as adders and multipliers [Ull84]. Three fundamentally different techniques are used, providing bounds on A (area), A x T (area-time) and A x T^2.

It seems that the use of prediction has predated CAD in the circuit design area. In the early 1960's, E. Rent performed an unpublished study (at IBM) on the structure of computer logic design. His formulation, now known as Rent's rule, was followed by many other studies, not only on the block-to-pin ratio, but also on other physical implementation parameters [Han88]. More recent studies achieve excellent correlation between predicted and actual physical design characteristics [Ped89, Sas85, Kur89].

The role of estimation has not received much attention in the architectural and high- level synthesis area. Early estimation efforts include a model, developed by Davio et all [Dav83]. One of their assumptions was that all nodes (called universal logic elements) have identical delay, area and functional characteristics, which greatly reduces the prediction complexity, but severely limits the application range.

Significant and important work has been performed at the University of Southern California. Kurdahi [Kur87] presented a technique to predict the number of required registers using a variant of the Dinic max-flow/min-cut algorithm, which was later refined by Mlinar [Mli91. Jain [Jai88] used absolute min-bounds (discussed later in this chapter) to drive the module selection process. Most recently, Kucukcakar [Kuc91] reported the successful usage of prediction tools in the partitioning during behavioral synthesis as well as several new estimation approaches [Kuc90].

Several approaches on the border between high level and lower level design have been reported. The Chippe expert design system [Bre90] analyses interconnect in terms of performance, area, and power using a worst-case waterfilling model. The ELF system had a mechanism for the prediction of the wiring area, given an RTL level description [Gyr84]. Several authors, including [McF90b], studied the area-delay performance trade-off, yielding a model which could be used in a prediction tool. Finally, Powell and Chau proposed an approximation technique for the early estimation of power consumption [Pow90].

The techniques presented in this chapter differ in both the employed techniques as well as the application areas. The idea of relaxation is introduced and applied extensively for the estimation of computational units, memory and interconnect. Attention is paid to hierarchy in the representation, an area which was largely unexplored until now. Furthermore, techniques to derive the best possible estimations for max-bounds are presented. The obtained results are analyzed and verified to a great extent.

3.1.2 Global Framework

Before starting the discussion of the estimation algorithms, a number of assumptions and definitions have to be put forward. First of all, we assume that the algorithm under study is represented as a control data flow graph G (N, E, C), where the nodes N represent the flow graph operations, and the edges E and C are respectively the data and control dependencies between the operations. The control dependencies are used to express relations between operations, which are not imposed by the data precedence relations. The control dependencies are particularly useful for the expression of timing constraints, as well as for the implementation of side effects caused by memory assignments (both for background and foreground memories) [Rab91a].

We also assume that the graph G is a hierarchical graph: each vertex N of G can be an instance of a subgraph G'(N', E', C'). An example of such a hierarchical flow graph is shown in Figure 1. The representation allows for the simple introduction of loops and block-conditionals in the flow graph. It is assumed that for each data dependent loop, either the maximum or average number of iterations is known. Which one is chosen depends upon the application: some signal processing applications (for instance audio processing) require that a fixed throughput rate is sustained, hence enforcing worse case design. Others, such as speech recognition, only require an average rate. The value of those non -deterministic parameters can be estimated through algorithm profiling, based on simulation or sometimes using amortization techniques [Kam91].

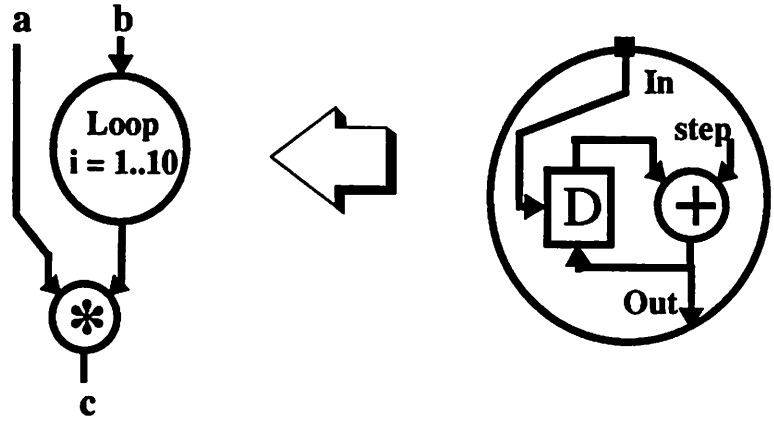


FIGURE 3.1. Example of Hierarchical Flow Graph. This graph is equivalent to the following computation: c = a * (b + 10 * step). D represents a delay operation with initial value In.

In order to make the estimations useful and accurate, a well defined underlying hardware model is essential. The following restrictions are placed on the implementation:

- All edges E represent variables, which will be stored in registers.
- All leaf nodes N are purely combinational. The hardware unit of choice r (such as adder, multiplier or ALU) and its execution time t_{dr} (in number of clock cycles) is known a priori.

It is always possible to transform a non-complying flow graph into the format described above by contracting nodes, connected by temporary edges (without storage), into a single, complex combinational node.

Without loss of generality, the chapter will adopt the hardware model H, shown in Figure 2a: it is assumed that all registers are clustered in register files, connected to the inputs of the execution units. The techniques described below are however easily adapted to other models, such as the "register file - interconnect - exu - interconnect" model, shown in Figure 2b (called H* in the rest of the text). While the hardware model will not affect the execution unit bounds, it

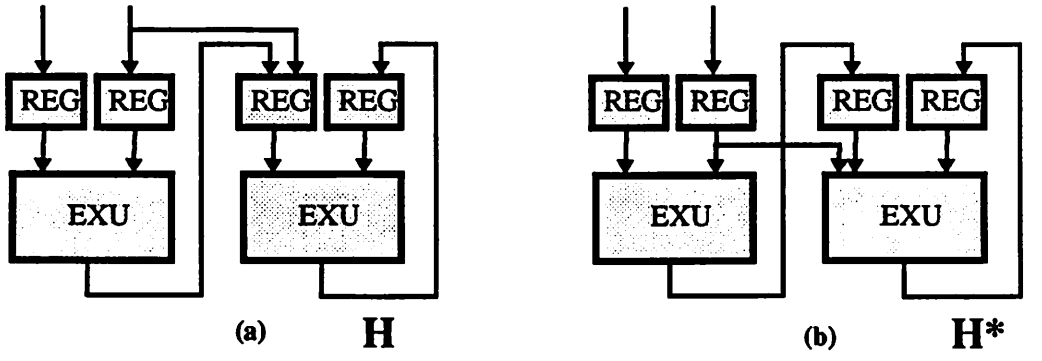


FIGURE 3.2. Hardware Models: (a) INTERCONNECT-REGFILE-EXU, (b) INTERCONNECT-REGFILE-INTERCONNECT-EXU

will have a severe effect on interconnect and register bounds.

The estimation process for real time applications can be defined within this frame- work:

Given a hierarchical flow graph G(N,E,C), an underlying hardware model H and a maximum execution time t_{max}, determine the minimal and the maximum bounds on the required hardware resources (execution units, registers and interconnect) such that the graph G can be executed within t_{max}.

For sake of completeness, it should be mentioned that the techniques presented below are easily adapted to address the dual problem (given the hardware resources, estimate the bounds on the execution time). The rest of the chapter will now proceed as follows: Section 3.2 will describe techniques to estimate the max bounds on the resources, while a selection of

approaches to estimate min bounds will be analyzed in Section 3.3. The chapter will be concluded with a study of the applications of predictions and future work. A number of examples will be used throughout the chapter to demonstrate the effectiveness of the proposed techniques.

3.2 ESTIMATING THE MAXIMUM BOUNDS

It might be argued that maximum bounds on hardware resources are hardly interesting, since the synthesis process is in essence a minimization process. However, knowing those upper bounds helps to delineate the search space for the hardware allocation and transformation processes. Furthermore, it is in general advantageous to have the maximum bounds on the resources as high as possible, since they are a measure of the concurrency available in a particular instance of the flow graph (this is demonstrated further in the chapter as well as in [Pot91a]). Therefore, results of the max bound estimation can act as a driver for the concurrency improving transformations.

3.2.1 Max Bounds on Execution Units

For the sake of clarity, we will first assume that the graph G does not contain any hierarchy. This constraint will be relaxed further in the chapter. The estimation process starts with a topological ordering and leveling of the graph with respect to the input nodes and the output edges. As a result, the earliest (t^i_{asap}) and latest (t^i_{alap}) execution times are obtained for each node N_i. The time slot available for the execution of N_i is called the *slack time* ($t^i_{slack} = t^i_{alap} - t^i_{asap}$). The length of the critical path is also determined during this process.

Based on this information, a set of parallelism plots $par_r(t)$ (with r = 1..R) can be constructed:

$$par_r(t) = \sum all\ nodes\ i\ executed\ on\ resource\ r\ \ (\text{with}\ \ (t^i_{asap} \le t \le t^i_{alap})) \quad \text{(EQ 1)}$$

Such as parallelism plot displays nothing else than the potentially available concurrency over time. Hence, the max bound on resource r equals:

$$max_r = \underset{t=1}{\overset{t_{max}}{MAX}} \; (par_r(t)) \qquad \text{(EQ 2)}$$

(EQ 2) is however overly pessimistic, since it totally ignores the precedence relationships, which might prevent nodes from being executed simultaneously, even though they have overlapping slack times. We will call this bound the *absolute max bound.* The simple example of Figure 3 will clarify the issue. Assume that all operations take 1 control step and that t_{max} equals 4 clock cycles. The earliest and latest execution times of each node are shown between brackets. par_+ (t) is plotted in Figure 3b. This suggests that the maximal parallelism for adders equals 3. A close inspection of the plot of Figure 3b clearly proves that this is not achievable, due to the precedence relations between the operations.

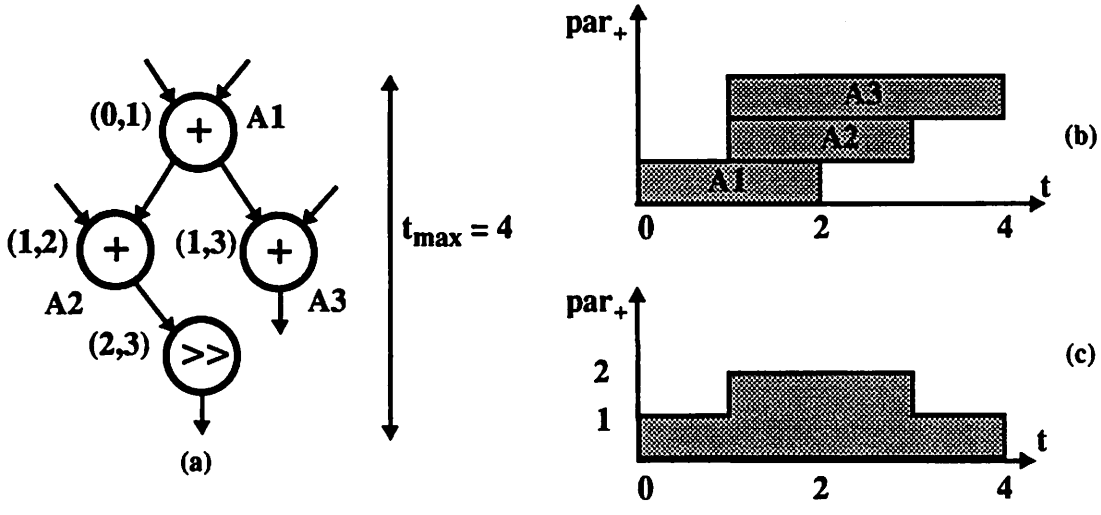


FIGURE 3.3. Analysis Of Max Bound: Example Flowgraph (a), Parallelism Graphs before (a) and after (b) precedence collapsing. It is assumed that each operation takes one control step.

A more precise max bound can be obtained by eliminating nodes with potential concurrency, but with precedence relationships between them (as demonstrated in Figure 3c for the simple example). This task can be defined as a maximum independent set problem:

Given: For control step *t* and resource *r*, a set of nodes N_i with the following properties: N_i is executed on r and $t^i_{asap} \le t \le t^i_{alap}$, a set of precedence relations (E,C) between N_i.

Problem: Determine the maximal potential concurrency at time *t* for resource *r*.

Solution: Construct a directed graph F(N, R), with the N_i as nodes. An edge R_{ij} is provided between two nodes N_i and N_j when there exists a precedence relation ($\in$ (E,C)) between the two nodes. The maximal concurrency equals the *maximum independent set* of F.

Proof: Two nodes N_i and N_j can be executed simultaneously at time *t* when no precedence relationship exists between them, or, in other words, when no edge exists between them in the graph F. This is exactly the definition of an independent set: two nodes of a graph F form an independent set, when F contains no edge between those two nodes. The maximum possible concurrency then obviously equals the maximum set of nodes without precedence relations, or equivalently, the maximum independent set.

The maximum independent set problem is known to be NP-complete. Fortunately, the graph F exhibits a property, which turns the problem into a polynomial one. It is known that the maximum independent set problem for a class of graphs, called *comparability graphs* [Gol81], can be solved in polynomial time ($O(N^3)$) using a minimum-flow algorithm. A comparability graph F(N,R) is defined as a graph with the following property: if $R_{ij} \in R$ and $R_{jk} \in R$, then also $R_{ik} \in R$. This is clearly valid for the graph F, defined in the max bound problem: when a precedence relation is present between nodes N_i and N_j as well as between nodes N_j and N_k, then node N_i has also to precede N_k. Efficient algorithms to solve the maximum independent set problem for comparability graphs have been published in [Gol80]. It should also be noticed also that the

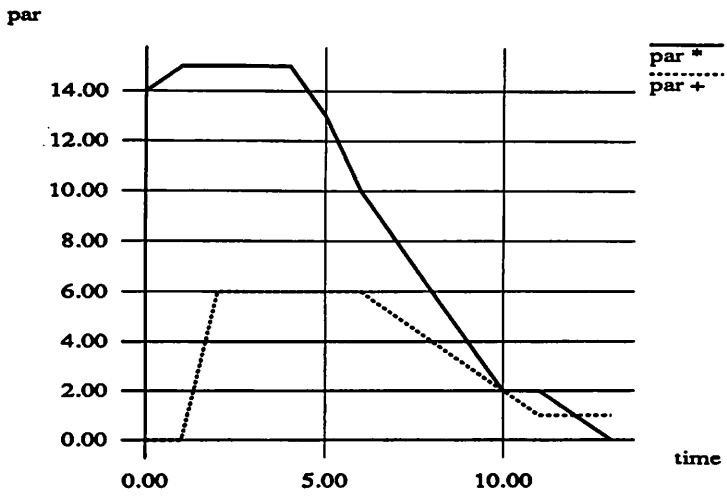


FIGURE 3.5. Parallelism Plots for 7th Order IIR Filter (t_{max} = 14, t_* = 2, t_+ = 1)

sizes of the graphs F are small, since they only consider the nodes, alive at time *t* and executing on resource *r*. The independent set problem has to be executed for every time *t* and for every resource (thus t_{max} x R times) to obtain the improved parallelism graphs.

The max bound algorithm has been applied on the example of a seventh order biquadratic IIR filter, shown in Figure 4. The results are plotted in Figure 5 for both adder/subtracters and multipliers[1] (for t_{max} = 14). From the parallelism plots, it can be deduced that the max bounds on

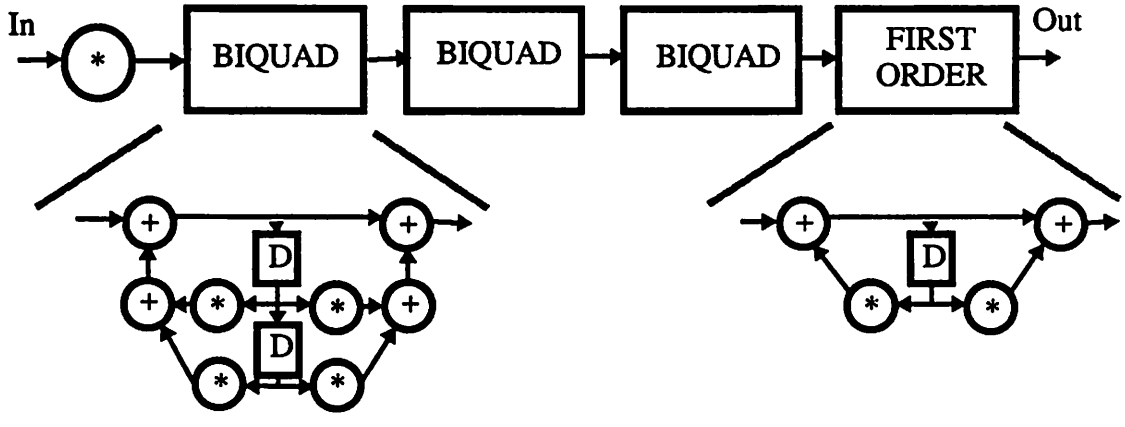


FIGURE 3.4. Seventh Order Biquadratic IIR Filter: Signal Flow Graph

1. In a real implementation of such a filter, multiplications are replaced by add/shifts. We have opted here for using parallel multipliers (with a duration of 2 clock-cycles) to simplify the example.

multipliers and adders respectively equal 15 and 6. More important than those bounds (which are of little practical value besides the delineation of the search space) is the structure of the parallelism graphs: almost all parallelism is available in the initial clock cycles. This will definitely result in a poor implementation with low resource utilization towards the end of the algorithm. This demonstrates that the distribution of the plots can serve as a measure to drive resource utilization improving transformations, such as retiming and pipelining [Pot91a].

The above techniques can be easily extended to cover hierarchical graphs as well. The main problem in dealing with hierarchy is that the available time is only known for the uppermost level and not for the sub-graphs. Fortunately, there is little or no dependency between the available time and the max-bound (since the dependencies remain identical). We therefore opted for the following approach. For each sub-graph, the max-bounds of the resources are estimated using the critical path as the available time. The max-bounds for the hierarchical graph are then obtained by taking the max over all sub-graphs for all resources.

3.2.2 Max Bounds on Connectivity and Registers

Similar techniques can be used to estimate max-bounds on connectivity. The algorithms are executed on the connectivity graph G_c (N_c, E_c). Every node N_c in G_c corresponds to an edge $E(N_i, N_j)$ in G and represents a hardware connection between resource r_i and r_j. For our hardware model H, the slack time of N_c is equivalent to the slack time of the source node N_i, since the output bus is directly connected to the source and should therefore be reserved for the duration of the computation. An edge E_c is defined in G_c when there exists a precedence between two interconnections. These precedences can be derived directly from the computation graph G. An example of how to derive the connectivity graph from the computation graph is shown in Figure 6. It should be stressed that the construction of G_c strongly depends upon the hardware model H used. Furthermore, one should be aware that, although there exist a strong correlation between

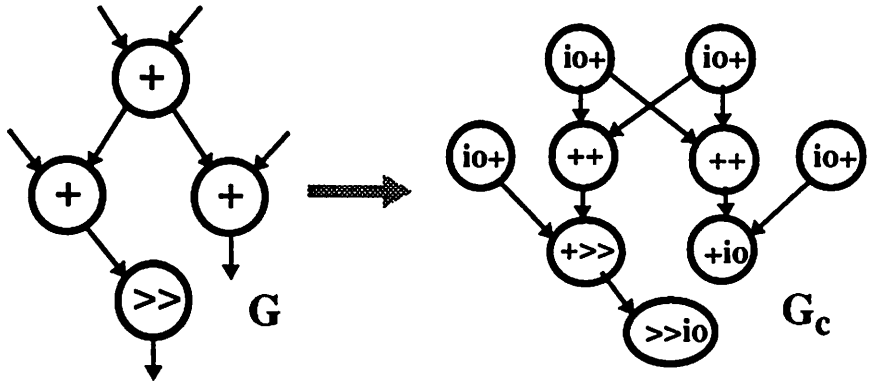


FIGURE 3.6. Derivation of Connectivity Graph from Computation Graph. The nodes in G_c are represented by the source and destination resources (e.g io→+).

the number of busses in the architecture and the interconnect area in the physical implementation, no precise cost can be attributed to a bus. This is in contrast with execution units and registers, which have a very precise implementation cost. Finally, it should be noticed from Figure 6 that the interconnect estimation also allows us to estimate the number of input-output ports needed. This is of crucial importance in real time applications, which are often input-output (and hence IO-pin) hungry (for example, see [Sto90]). This measure can for instance be used when considering chip partitioning.

Establishing a sharp max-bound on registers is non-trivial and is strongly dependent upon the selected hardware model. In the hardware model H, the register count is closely correlated to the EXU assignment: in the worse case, every operation can be assigned to a different EXU and hence every variable has its own register. Therefore, the absolute max-bound on the number of registers used is identical to the number of edges in the graph, each of them multiplied with its fan-out. Parallelism plots can also be derived for registers, using exactly the same techniques as described higher. The graph G_r used is actually identical to the interconnect graph G_c in case of the H hardware model. Each node now corresponds to a variable. The slack time of the node is set to the maximum life-time of the variable, which equals $t_{al}^{dest} - t_{as}^{source}$, with *source* the source node of the variable and *dest* the destination node. Once again, these parallelism plots are

only indicative for the demand on registers over time and have no meaning from a max-bound point of view.

In the hardware model H*, the broadcasting factor is equal to 1, potentially saving some registers at the expense of extra interconnect. Since the register assignment is decoupled form the EXU-assignment, more accurate bounds can be predicted using the techniques described higher.

3.3 ESTIMATING THE MINIMUM BOUNDS

From a design point of view, far more interesting information is represented by the minimum bounds: accurate lower bounds allow us to estimate the absolute minimal area, needed for the implementation of a given computational graph. Lower bounds can also serve as an initial seed for allocation and design space search processes, normally resulting in faster convergence. Finally, a good lower bound allows us also to judge the quality of solutions produced by heuristic or statistical tools, tackling NP-complete problems, such as schedulers or module selectors.

Unfortunately, no exact lower bounds for the design synthesis problem have been established and deriving those might prove to be an NP-complete on itself[2]. In order to be useful, the estimation process should be very efficient (the complexity of the estimation routines should not exceed $O(N^2)$). This precludes the utilization of complex estimation algorithms. Instead, we opted for a technique called **discrete relaxation**, which turns the estimation problem into a tractable one by relaxing some of the constraints imposed by the original problem. As will be demonstrated below, this results in extremely efficient estimation, while yet delivering very sharp bounds. One set of constraints which can be relaxed on are the precedence relations. Most of our attention will be devoted to this class of relaxations. Similar to the chapter on maximum bounds, we will first concentrate on the estimation for execution units and extend the approach later on to

2. The authors are not aware of any of any proof of this statement.

registers and interconnect. At the end of the chapter, we will discuss some other relaxation approaches.

3.3.1 Min Bounds on Execution Units -Leaf Graphs

As stated in section 1.1.1, the majority of the complexity estimation approaches in the high level synthesis arena are based on the *absolute min-bound.* This min-bound is in essence equivalent to the traditional approach of a designer, when he measures the complexity of an algorithm by counting the number of multiplies (as the parallel multiplier is the most expensive execution unit) and dividing them by the available time. Such a measure is a poor predictor of the ASIC implementation cost of an algorithm. It assumes that the flow graph contains enough parallelism to support 100% utilization of the resource. Furthermore, important elements contributing to the implementation cost, such as input-output, interconnect, fore- and background memory are ignored. This approach therefore generally results in a poor estimation. Within our framework, the absolute min-bound for an execution unit r (r = 1..R) is defined in (EQ 3).

$$min_r^{abs} = \frac{\eta_r \times t_{dr}}{t_{max}} \qquad \text{(EQ 3)}$$

with η_r the number of nodes to be executed on resource r and t_{dr} the number of clock-cycles it takes to execute one operation on r.

We have evaluated the performance of the absolute min-bound using a class of 50 examples (all of them without hierarchy). The list of the examples include the 7th-order IIR filter, the standard 5th order Wave Digital Filter, an 11th order FIR filter, a 19th order CORDIC rotation and an 8-point Discrete Cosine Transform. For all those examples, we constructed different alternatives structures (and hence different graph properties) by applying transformations such as pipelining and retiming. Furthermore, we considered multiple ratios of t_{max}/critical path for each example. This ratio (which will be called the *stress ratio* from now on) influences the per-

formance of the estimation algorithms in an important way. We have compared the results of the estimation with the results obtained after going through the complete synthesis process[3]. The results are plotted in Figure 7. The cost factor used in the evaluations is equivalent to the total area of the execution units. The maximum error observed between actual and estimated cost equals 386.3%, while the average and median errors respectively equal 72.1 and 86.6%.

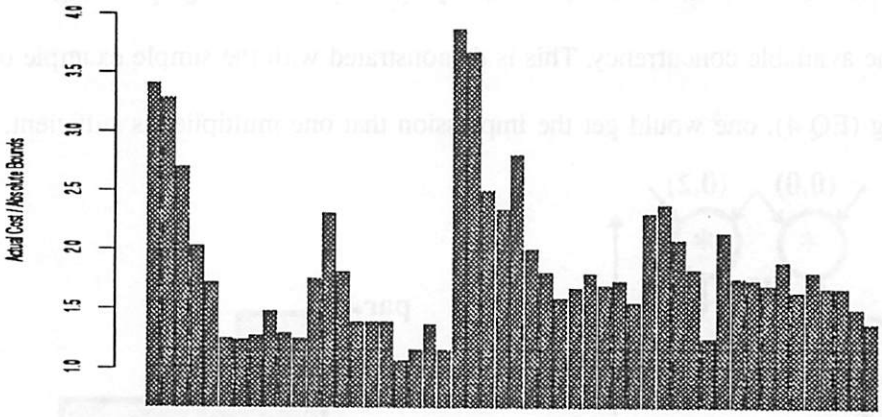


FIGURE 3.7. Ratio between Actual Cost and Absolute Min Bound (for 50 examples).

An improvement on the absolute lower bound can be found by observing from the parallelism graphs, obtained in the previous section, that for some clock cycles not enough parallelism is available to sustain 100% utilization. This results in a more precise lower bound:

$$min_r^{adj} = \frac{\eta_r \times t_{dr} + Unused\ Time}{t_{max}} \qquad \text{(EQ 4)}$$

Unused Time is actually a function of min_r^{adj} as the resource utilization is clearly dependent upon the number of resource available. (EQ 4) has therefore to be solved iteratively:

1. Derive the Parallelism graph for r, using the techniques discussed in the previous section. Set the initial value of MIN to the absolute min-bound (EQ 3).

3. To make the comparison between the different algorithms fair, we have used identical scheduling and allocation routines for all examples.

2. Compute Unused Time given MIN.
3. Recompute MIN using (EQ 4).
4. If MIN changed with respect to the previous iteration, go to 2, else stop.

Even though this approach presents a significant improvement over the absolute bound, it still might be significantly off. This discrepancy is mainly caused by the fact that nodes with a large slack (or a lot of freedom to move) are falsifying the parallelism graphs by giving a too rosy view on the available concurrency. This is demonstrated with the simple example of FIGURE 3.8. Using (EQ 4), one would get the impression that one multiplier is sufficient, as the

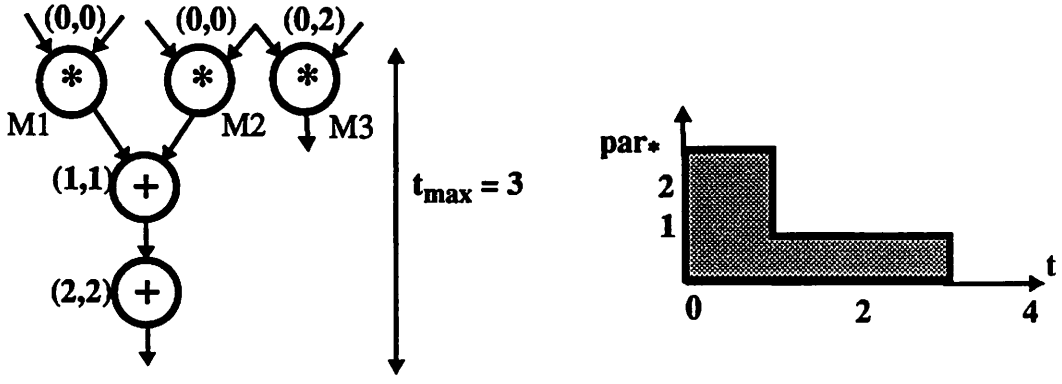


FIGURE 3.8. Nodes with Large Slack Can Give the Impression of Plentiful Concurrency (all operations take one clock-cycle).

Unused Time on the Parallelism Graph for 1 multiplier is 0. A close inspection of the flow graph reveals that 2 multipliers are needed: both multiplications M1 and M2 have to be executed in the first time slot. The discrepancy between estimations and actual result is caused by the large slack of node M3, which gives the impression that at least one multiplication can be executed in every clock cycle.

A more advanced approach is therefore necessary. Our proposed approach is based on the principle of relaxation: while the general scheduling problem is NP-complete, some simplified scheduling problems have been proven to be of polynomial complexity. By removing some constraints of the original problem, we can turn the estimation problem into a polynomial one. In other words, we trade-off accuracy versus speed. In this section, we will relax on the precedence

constraints. For the lower bound estimation problem for EXU r, let us temporarily consider the precedences only indirectly (through the ASAP and ALAP times of the nodes) and ignore the direct formulation. This translates the estimation problem into the following format:

Relaxed Estimation Problem: Given a computational problem, consisting of η_r identical tasks with integer ASAP and ALAP times and a known duration t_{dr}. Determine the minimum number of resources *r* needed to complete the task within the available time t_{max}.

This problem cannot be solved directly, but can be defined as an iterative version of its dual formulation:

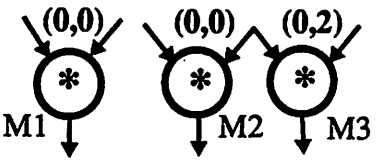
Dual Relaxed Estimation Problem: Given a computational problem, consisting of η_r identical tasks with integer ASAP and ALAP times and a known duration t_{dr} and the number of available resources *r*. Determine the minimum execution time t_{min}.

Given a solution for the dual problem, the original relaxed iteration problem can then be solved with the following simple iteration:

Relaxed Estimation Problem (iterative version):

1. *Set min_r to the absolute lower bound min $^{abs}_{r}$.*
2. *Given min_r, determine the minimum execution time t_{min} (dual relaxed estimation problem).*
3. *If $t_{min} > t_{max}$ or if no solution, go to 2, else min_r is the relaxed min-bound.*

This procedure is illustrated with the aid of the simple example of Figure 8. The absolute min-bound on the number of multiplications for this example is equal to 1. Consider know the relaxed problem shown in FIGURE 3.9. It is obvious that no solution can be found for the scheduling of the problem on 1 multiplier. When 2 multipliers are available, the problem can be solved in 2 clock-cycles ($< t_{max} = 3$). Hence the min $^{rel}_{*} = 2$.



1 multiplier

	mult 1
0	M1
1	M3

M2 ?

2 multipliers

	mult 1	mult 2
0	M1	M2
1	M3	-

FIGURE 3.9. Determination of Relaxed Min-Bound for the example of FIGURE 3.8.

The dual estimation problem as defined above is well known in the scheduling literature. From [Sim81], the following analysis can be derived.

(1) When the duration t_{dr} of all tasks (or operations) is equal to 1 clock-cycle, then the minimum execution time can be determined exactly using a *slack driven list scheduling* (also known as the earliest deadline scheduling algorithm). The complexity of this algorithm is O(N log N), with N the number of tasks.

(2) When the duration of the tasks is larger than 1, the problem becomes more complex. It can be transformed into a scheduling problem with unit task duration time, but with the ASAP and ALAP times set to real numbers by dividing all times by t_{dr}. Finding the exact solution for this problem requires back-tracking during the list-scheduling and is known as the *earliest deadline with barriers scheduling* algorithm [Sim81]. The complexity of this algorithm is $O(N^3 \log N)$. Since this exceeds our goal of using only algorithms with maximally quadratic complexity, some further relaxation is needed. This can be achieved by turning the ASAP and ALAP times into integer numbers: the ASAP times are rounded to the nearest lower integer number, while the ALAP time are rounded to the next higher integer. The resulting problem can once again be solved with the earliest deadline scheduling problem. It is easily seen that the integer relaxation can only lower t_{min}, hence preserving the min-bound nature of the obtained solution.

The performance of the relaxed estimation routines is analyzed using the same benchmark set as was used for the absolute lower bound. The results are plotted in Figure 10. The maximum

error has been reduced to 67%, while the average and median errors respectively equal 13.7 and 7%. The largest errors occur when the stress ratio approaches 1. This is easily explained by the fact that at that time the relaxation on the precedence relations introduces an over-simplification.

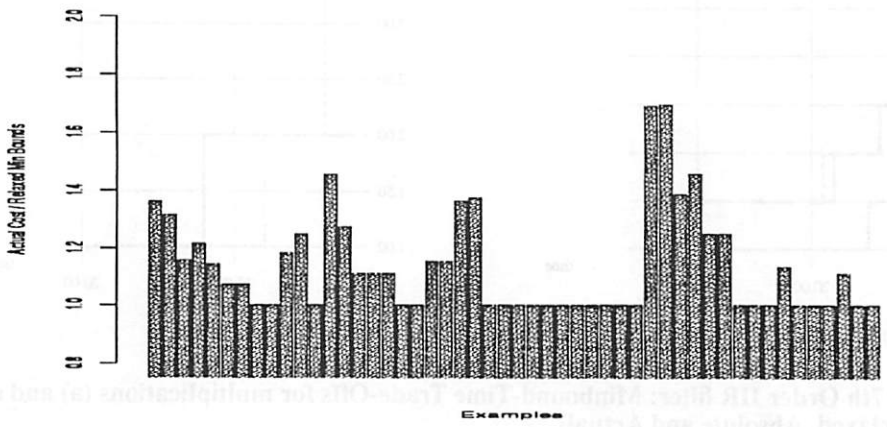


FIGURE 3.10. Ratio between Actual Cost and Relaxed Min-Bound (for 50 examples)

The proposed iterative algorithm for solving the min-bound problem can also be used to generate the minbound-time plots for each execution unit, which determine for each possible time t_{max} the minimal number of units needed. These plots are extremely useful when studying the area-time trade-off's for a particular algorithm (this will be discussed in more detail in the application section). They will also be used extensively when performing estimations for hierarchical graphs.

Generating the Minbound-Time Graph for resource *r:*

1. Set min_r *to 1.*
2. Solve t_{min} *using the dual relaxed estimation algorithm.*
3. When $t_{min} > t_{critical\ path}$, *go to 2, else stop.*

The minbound-time graphs for multipliers and adders for the 7th-order IIR filter are plotted in Figure 11. To demonstrate the excellent performance of the relaxed estimation algorithms, the area-time plot obtained using the absolute min-bound estimation technique as well as the

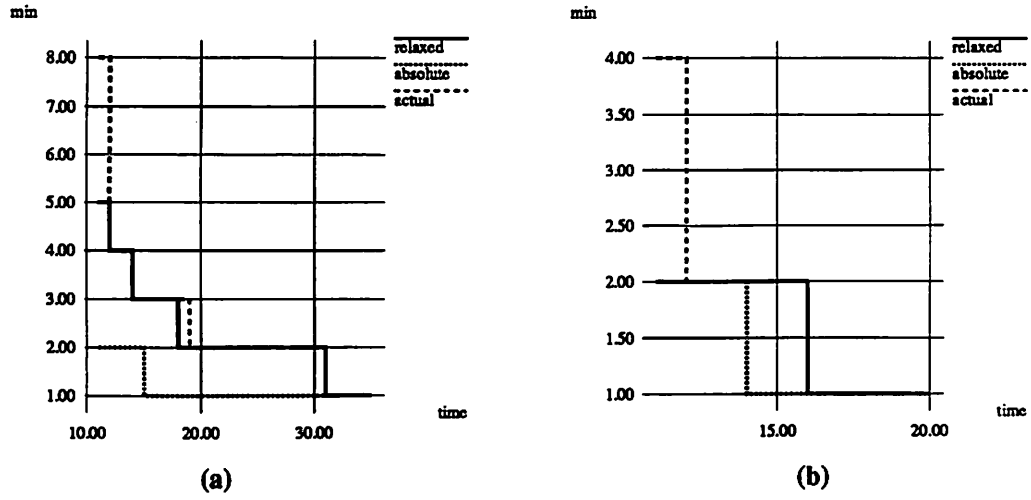


FIGURE 3.11. 7th Order IIR filter: Minbound-Time Trade-Offs for multiplications (a) and add/ subtract (b) (Relaxed, Absolute and Actual)

actual cost-time plot (obtained after allocation, assignment and scheduling) are also included. As can be noticed form the plots, the only major discrepancy between actual cost and relaxed min-bound occurs when $t_{max} = t_{critical\ path} = 11$.

3.3.2 Min Bounds on Execution Units - Hierarchical Graphs

The situation becomes somewhat more complex, when hierarchical graphs are considered. The problem here is that t_{max} is only defined for the overall problem. The distribution of the time over the sub-graphs is unknown and is actually an optimization problem on its own. Without a loss of generality, we will assume from now on that at every hierarchy level, the nodes of the graph are either all hierarchy-nodes or leaf-nodes. This can be achieved by clustering leaf-nodes into sub-graphs, such that all precedences are preserved. We assume also that only one sub-graph can be executed at a time (single thread of control).

One method to perform the hierarchical estimation uses the minbound-time plots, derived in the previous paragraph. Given a graph G(N,E), where each node N represents a sub-graph

G'(N',E'). Assume that for each node N, the minbound-time plots of its sub- graph G' are known as well as the maximum[4] number of iterations on the node.

The minbound-time plot for a resource *r* and for graph G (called $MB_r(t)$) can now be constructed from the minbound-time plots of the sub-graphs ($MB_r^i(t)$, with i = 1..N) in the following way:

Procedure EstimateHierarchy (for resource r):

1. Set η_r to 1.

2. Compute t_{min} for graph G using (EQ 5).

$$t_{min} = \sum_{i=1}^{N} (MB_r^i{}^{-1}(\eta_r) \times iter^i) \qquad \text{(EQ 5)}$$

with $iter^i$ the number of iterations of node i.

3. If $t_{min} > t_{critical\ path}$, increment η_r and go to 2, else stop.

This procedure is illustrated for a simple example in Figure 12. In the case of conditional graphs (if-then-else functions), we take the results of the worse case sub-graph. Procedure *EstimateHierarchy* is repeated for every hierarchy level in the graph in a bottom-up fashion till the top-level is reached. At that level, the available time t_{max} is known and the actual min-bound t_{min}^r can be determined. The minbound-time plots for a 19th-order CORDIC algorithm (containing one hierarchy level) are plotted in Figure 13. The cost plotted here is the sum of the estimated minimum costs of the adder/subtracters (unit cost 3), shifters (cost 4), comparators (cost 3) and multiplexers (cost 1)[5]. A large discrepancy between absolute and relaxed min-bound can once again be noticed. Also noticeable is that the actual cost tends to change in large steps every 19 cycles (which is equivalent to the number of iterations of the loop). This phenomenon is not present in such an outspoken form in the relaxed estimation cost.

---

4. Average number when looking at the average throughput.

5. Those cost ratios are obtained from the actual data-path library of the LAGER-IV system [Shu91].

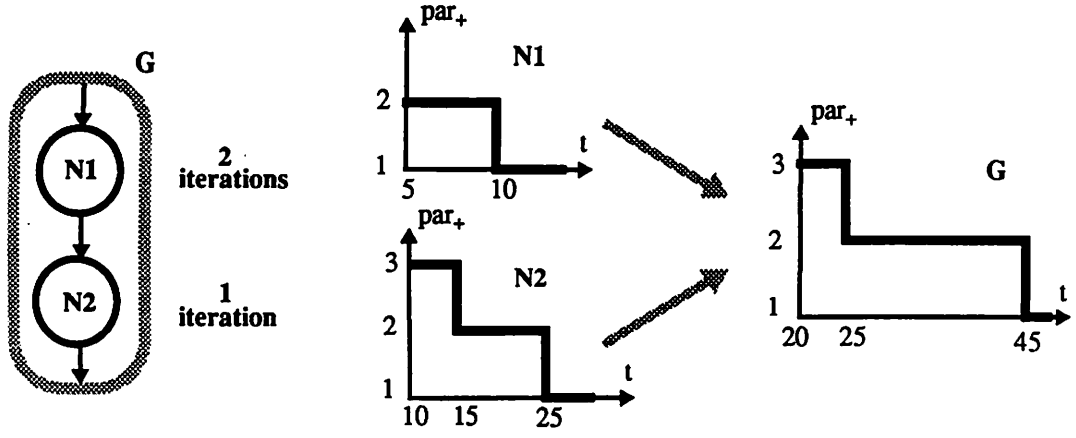


FIGURE 3.12. Construction of Hierarchical Minbound-Time Graphs - Simple Example.

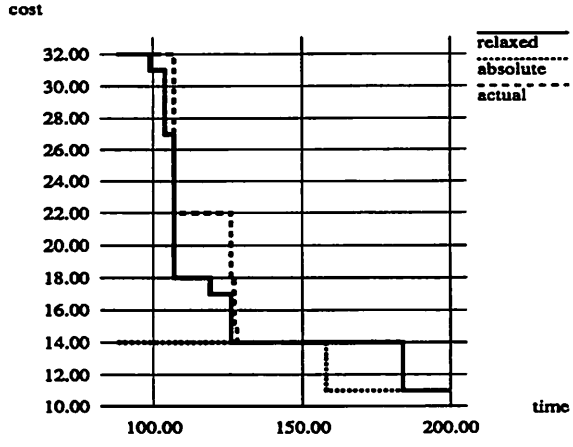


FIGURE 3.13. Estimated Cost of a 19-th order CORDIC algorithm (relaxed, absolute and actual).

The procedure *EstimateHierarchy* in fact introduced another form of relaxation: the actual time allotted to each sub-graph differs from resource to resource and is determined by (EQ 5). In reality, this is clearly not the case. This relaxation might therefore be overly optimistic, as it picks the optimal solution for each resource independent of the other resources. This will hurt especially when the *stress ratio* approaches 1. A more accurate solution can be obtained by considering the resources in a combined fashion. For instance, when considering two resources r1 and r2 simultaneously, (EQ 5) is reformulated as follows:

$$t_{min}^{r12} = \sum_{i=1}^{N} MAX\left(\left(MB_{r1}{}^{i\,-1}(\eta_{r1}),\ MB_{r2}{}^{i\,-1}(\eta_{r2})\right)\right) \times iter^{i} \qquad \text{(EQ 6)}$$

This approach turns the hierarchical estimation problem from an R times 1-dimensional problem into an R-dimensional one. This might sound bad, but isn't in reality. First of all, R is normally rather small (more than 6 different resources is rare). Secondly, only a couple of resources are critical and need more than one instance, which reduces the estimation space in an important way.

3.3.3 Min Bounds on Interconnect and Registers

All the above described techniques can be applied in an identical fashion for the estimation of the lower bounds of interconnect. The only difference is that the routines are applied on the interconnect graph G_c instead of on the computation graph G (Figure 6). As described higher, the slack of an interconnect node is identical to the slack of the source computation node: in the hardware model H, a bus is directly connected to the output of the execution unit and is thus reserved for the duration of the computation.

Estimating the minimum bounds on the number of registers requires a somewhat different approach. In order to find the absolute minimum on the register count, we have to assume that no broadcasting occurs and that each variable is alive for the minimum possible amount of time. The minimum lifetime now depends upon the slack-time of both the producer and consumer nodes. As illustrated in Figure 14, two scenarios are possible:

(1) The slack-periods of the producer and consumer nodes overlap (or $t_{alap}^{p} \geq t_{asap}^{c}$). In that case, the variable E can be consumed immediately after generation. The minimum lifetime is therefore t_{dr}^{c}. The interval where the variable can be alive stretches from t_{asap}^{c} till max (t_{alap}^{p}, t_{alap}^{c}).

(2) The slack periods of producer and consumer nodes do not overlap ($t^p_{alap} < t^c_{asap}$). Here, the variable E has to be minimally alive for a longer period, namely from t^p_{alap} till $t^c_{asap} + t^c_{dr}$. This is also the lifetime interval.

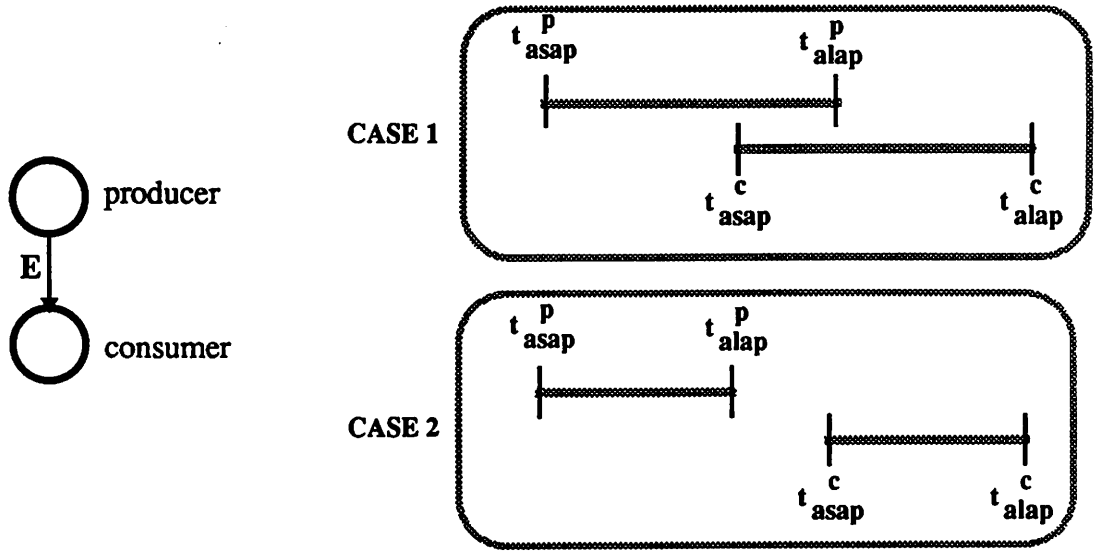


FIGURE 3.14. Minimal Variable Lifetime of Variable E: Possible Scenarios.

Estimating a lower bound on the number of registers, associated to an execution unit r (hardware model H), can now be solved in terms of the following relaxed scheduling problem: given a number of resources η_{reg} (the number of registers available) and two classes of tasks: all tasks of the first class have a fixed, integer duration (t_{dr}) and have integer ASAP and ALAP times. The tasks of the second class have a variable, integer duration but have a fixed scheduling time. Determine the minimum time to execute the tasks on the given resources.

The scheduling of tasks with varying duration is in general NP-complete (in fact, strongly NP-complete) [Sim81]. The fact however that the tasks with varying duration are fixed in time saves us here. The following modification of the earliest deadline algorithm can be used to approximate the solution:

1) Divide all times by t_{dr} and round all ASAP and ALAP times to respectively the next lower and upper integers. This is identical to the approach taken for execution units with duration larger than 1. This translates the problem into a scheduling problem with integer ASAP and ALAP times and task durations of 1.

2) Reserve the slots, needed by the tasks with variable duration (1), but fixed scheduling time.

3) Schedule the remaining tasks using the earliest deadline algorithm. For each time-slot, the number of available resources is equal to h_{reg} minus the number of reserved slots. The complexity of the algorithm is still O(N log N).

The obtained bound is clearly on the pessimistic side, since the actual solution will contain broadcasting. It can also be observed that the actual lifetime of the variables will approach the average value between min and max, rather than the minimal value: a simple explanation of this is that in a typical schedule, decreasing the lifetime of one variable actually increases the lifetime of other ones. We are studying other techniques (amongst others statistical - see future work) to get more precise register bounds.

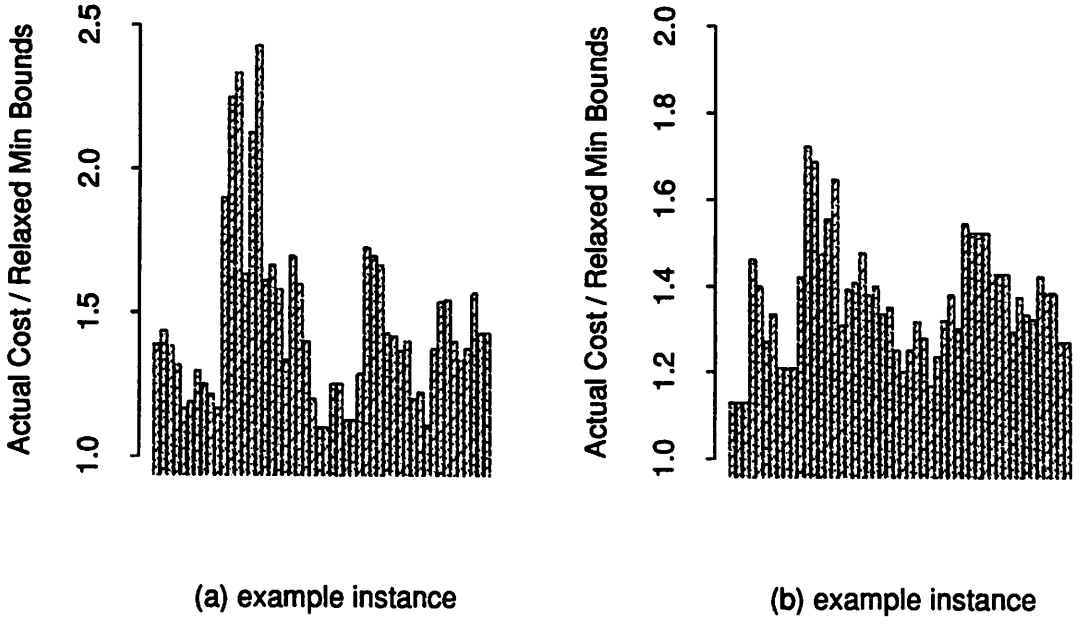


FIGURE 3.15. Ratio between Actual Cost and Min-Bound for interconnect (a) and registers (b) (for 50 examples)

Finally, its should me mentioned that a similar approach can be used to estimate the register count for the hardware model H*. Since this model does not support broadcasting, the estimated register count will be closer to the actual solution.

The performances of the estimation routines for interconnect and registers are analyzed using the benchmark set. The results are plotted in Figure 15.a for interconnect and Figure 15.b for registers. The maximum discrepancy for interconnect is 142%, while the average and median discrepancy are 46% and 39%. The maximum discrepancy for registers is 72%, while both the average and median discrepancy are 39%. The largest discrepancy for interconnect occurs again when the stress ratio is close to 1. The explanation is the same as in the case for execution units. The discrepancy for the number of registers is much less dependent on the stress ratio.

Although the discrepancy is still relatively small, it is not as impressive as in the case of execution units. The higher discrepancy between the min-bounds and the actual cost for interconnect and registers is partially due to the fact that the used scheduler is paying higher attention to execution units than to two other hardware components because the execution elements have higher implementation cost. Also, it seems that the mechanisms for the minimization of the number of interconnects and registers are not in as mature state as in the case of execution units and that there is some, although relatively small, room for the scheduler improvement with respect to two components. Finally, as we already stated the registers estimation seems inherently more difficult problem.

3.3.4 Other Relaxation Approaches

The number of relaxations, which can be introduced to simplify the high level synthesis scheduling problems, are almost unlimited. A large number of scheduling problems are indeed known to be of a polynomial complexity. Of course, the most interesting one are those that offer a good compromise between accuracy and run-times. We will briefly discuss three other approaches: relaxing on the constraints that operations should be scheduled on integer times, ignoring all edges which violate certain graph properties and ignoring the fact that the operation execution should be non-preemptive.

The first approach (and also the most promising one) is based on the fact that the scheduling, resource allocation and assignment processes can be formulated as integer programs [Pap82]. While integer program solvers are taking exponential time (and are hence unapplicable for problems of large size), linear programs can be solved in polynomial time. Turning an integer program into a linear one corresponds to relaxing on the integer starting time of the operations. While such as solution cannot be used in an actual design, it can be employed to provide an accurate lower bound on the execution time of a program, given the hardware resources (the dual estimation problem).

Hu [Hu64] showed that the as soon as possible scheduling algorithm produces the optimal solution, when the computational graph has an in-forest or out-forest structure. A graph can relaxed into this particular format by removing all precedence edge, which violate this constraint. The quality of the prediction depends upon the number of edges deleted and even upon the choice of the edges (since many forest structures can be derived for a single problem).

A max-flow based algorithm for the optimal scheduling of J jobs on M machines has been developed by Federgruen and Groenevelt [Fed86]. They assumed however that tasks can be scheduled in discontinuous intervals, which is, of course, not the case in high level synthesis. The algorithm might however be capable of predicting accurate lower bounds.

Finally, it is interesting to note that sometimes the addition of extra constraints might result in a more tractable problem formulation. Leung [Leu82] succeeded to optimally schedule a graph, when the execution times of the operations are restricted to at most k values. His algorithm, which uses a sophisticated dynamic programming approach, has a worst case run time of $O(\log p * \log m * n^{2(k-1)})$. While it is an impressive algorithmic result, its actual application is limited to cases, where k is small. Since this is often the case when the available time approaches

the critical path, this technique can be used to produce sharper bound for the estimation instances with large stress ratios.

3.4 APPLICATIONS

The proposed techniques have been implemented and incorporated into the HYPER synthesis system, which is targeted at the synthesis of high performance, data path intensive real time applications [Chu89, Rab91a]. Within HYPER, the complexity estimation routines are used in numerous places as will be demonstrated in the sections below. Complexity estimation however has a scope which rises beyond the confines of HYPER. Other applications for which the presented techniques could be useful are the areas of algorithm and design style selection as well system partitioning. A number of those applications will be discussed with the aid of a simple example, being an 8-point Discrete Cosine Transform (DCT). The DCT is used in virtually all video and image compression systems, such as present in HDTV and tele-conferencing. One form of the DCT is shown in Figure 16 [Vet86]. The following paragraphs will discuss the applications of estimation in a top-down fashion.

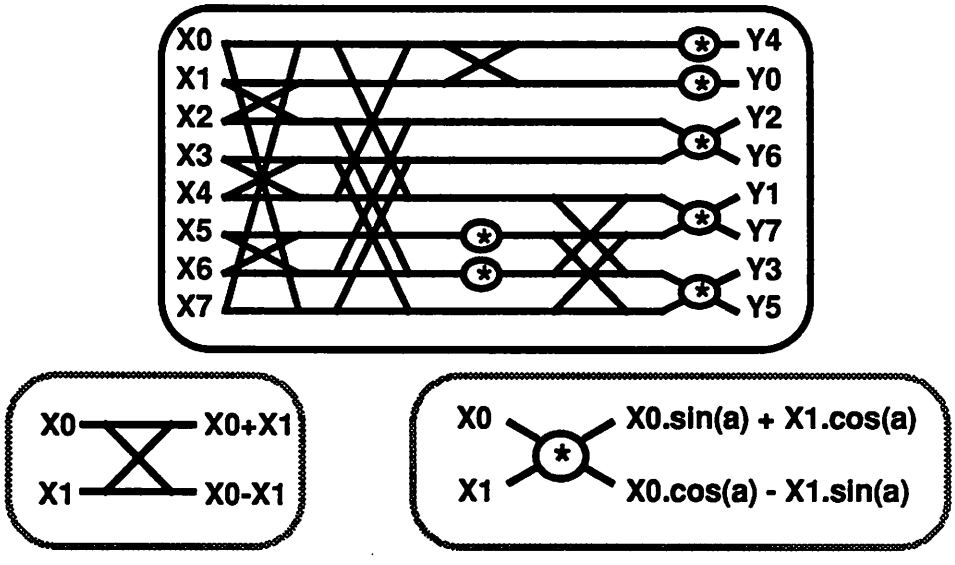


FIGURE 3.16. 8 point Discrete Cosine Transform - Computational Graph

3.4.1 Algorithm and Architecture Selection

Often various computational algorithms are available to perform the same function. The optimality of an algorithm depends upon the required throughput rate, the available input-output and memory bandwidth and the operation library available. The results of the estimation process can help to differentiate between different algorithms over a range of implementation constraints. For instance, while an FFT might require less multiplications than a DFT, the DFT can be advantageous when spectral information is only required for a limited frequency band or when memory is in short supply.

As another example consider the DCT example. An important part of the DCT is a rotation of the input data, requiring complex multiplications. It is well known [Bla85] that the number of multiplications required for a complex multiplication with a constant can be reduced by a reorganization of the computation. In Figure 17, it is shown that the reorganized computational graph reduces the number of multiplications by 1 at the expense of an extra adder and an increased critical path. In order to compare the two approaches, we have used the proposed estimation techniques to compute the implementation cost over the range of throughput speeds. For this and the coming examples, the cost is computed as the sum of the minimal execution cost plus the minimal register cost. The properties of the module library used are shown in Table 1.

	Speed	Cost
Add/Subtract	1	1
Parallel Multiplier	2	8
Serial Multiplier	8	2
Register	–	0.5

TABLE 1. Speed and Cost Properties of Simple Module Library

The results of the estimation process are shown in Figure 17. It is interesting to notice that there is no clear best algorithm. Although the reorganized flow graph reduces the number of (expen-

sive) parallel multiplications by 3, the increased critical path offsets the gain for certain throughputs. These observations are extremely hard to come by using just flow graph inspection.

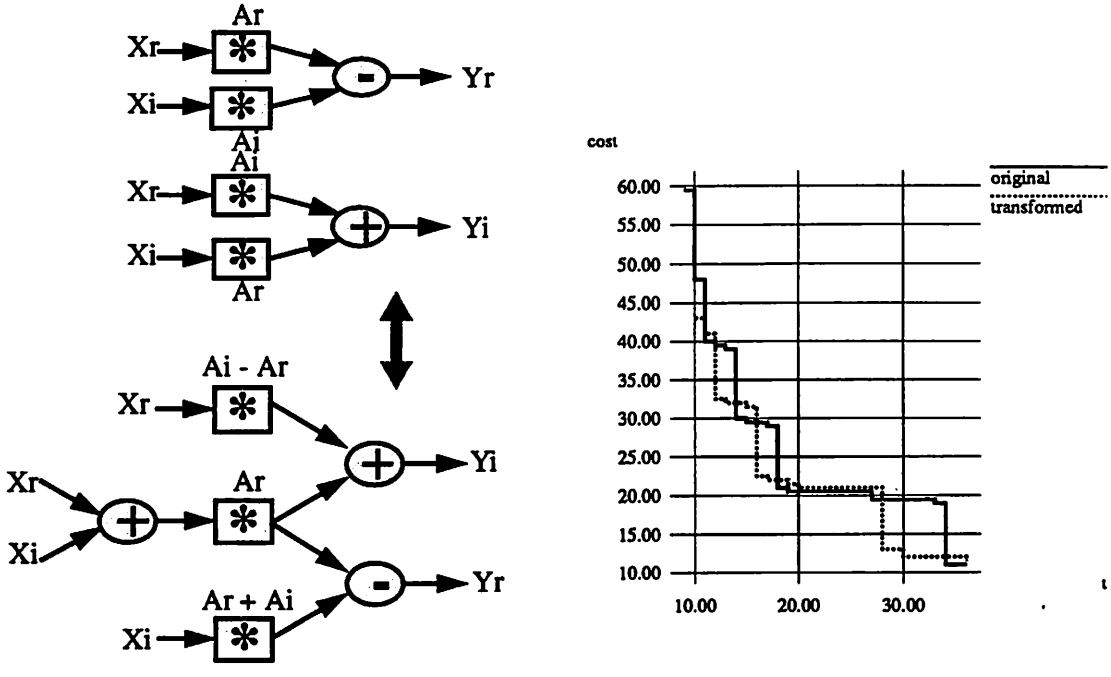


FIGURE 3.17. DCT: Comparison of Cost of Using Traditional or Reconfigured Complex Multiplications

Similarly, the estimation results can help to select between architectural styles, such as general purpose programmable versus custom programmable or hard-wired, time-shared interconnect versus a dedicated interconnect network or bit-serial versus bit-parallel. Different architectures can be compared by modifying both the hardware model as well as the available module set.

3.4.2 Module Selection

Accurate estimation can help to improve the quality of the module selection process, which selects between different alternative versions of an execution unit present in the hardware library. Traditional module selection tools [Jai88] use the absolute min-bound to estimate the cost of a given selection. We have demonstrated [Chu91] that the relaxed estimation can result in more optimal selections.

An example of how accurate estimations can help in the module selection process is given in Figure 18. Here we study the effect on the implementation cost of the DCT if we would use a slower parallel-serial multiplier instead of the fully parallel-multiplier. It can be observed that the serial multiplier solution only becomes dominant for very large values of the available time. For very small times, the parallel solution is obviously the only choice. In the intermediate zone, both solutions are comparable, which is somewhat predictable, since both multipliers have identical Area-Speed products.

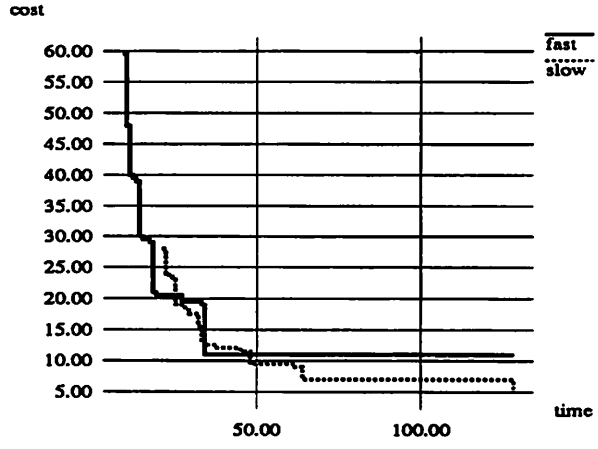


FIGURE 3.18. DCT: Parallel versus Serial Multiplier

3.4.3 Transformations

Optimizing transformations are an essential component of any synthesis environment. Pipelining, retiming, arithmetic laws and loop unrolling are typical examples of transformations, which are often applied to improve the implementation quality. The main questions arising in a transformation environment are what transformation to apply when and what improvement can be expected. Once again, estimations can help substantially to resolve these questions. An example of a combination of transformation-estimation is the *retiming for resource utilization* transformation [Pot91a]. This transformation, based on a probabilistic iterative improvement

approach, uses estimations to both determine the cost-function and hence the next move, as well as the optimal result and its distance from it.

The DCT example is used again to illustrate this point. We compared the original version (Figure 16) with a pipelined version, where three pipeline stages were introduced to increase the throughput. The estimated cost of pipelined and non-pipelined versions is compared in Figure 19. The results demonstrate that the pipelined approach achieves its goal of increasing the throughput, but also that over-pipelining hurts: at lower speeds, adding pipeline stages only adds extra registers.

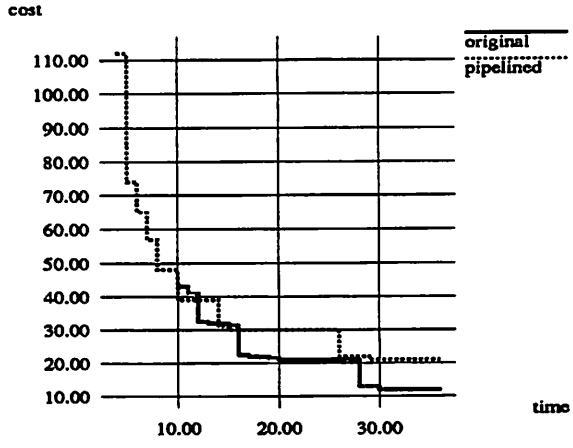


FIGURE 3.19. Pipelined versus Non-Pipelined DCT cost.

3.4.4 Design Space Exploration - Allocation

The derived minimum and maximum bounds delimit the design search space, thus speeding up the hardware allocation process. The minimum bounds can serve as an initial solution for the search. We have experienced that this solution is often very close to the final solution [Pot91b]. Information such as the distribution of the parallelism plots or the distance between the absolute and relaxed min-bounds can be used as a metrics to select an optimizing transformation [Rab90].

When all automatic techniques fail, feedback of the estimation information such as the bounds and the parallelism plots to the designer can help to guide him to through an interactive search process. Figure 20 shows a screen dump of the HYPER user interface. The bounds are printed in the main window. The plots on the left are respectively the parallelism plots for the execution units, registers and interconnect. The example shown here is the 7th order IIR filter discussed higher.

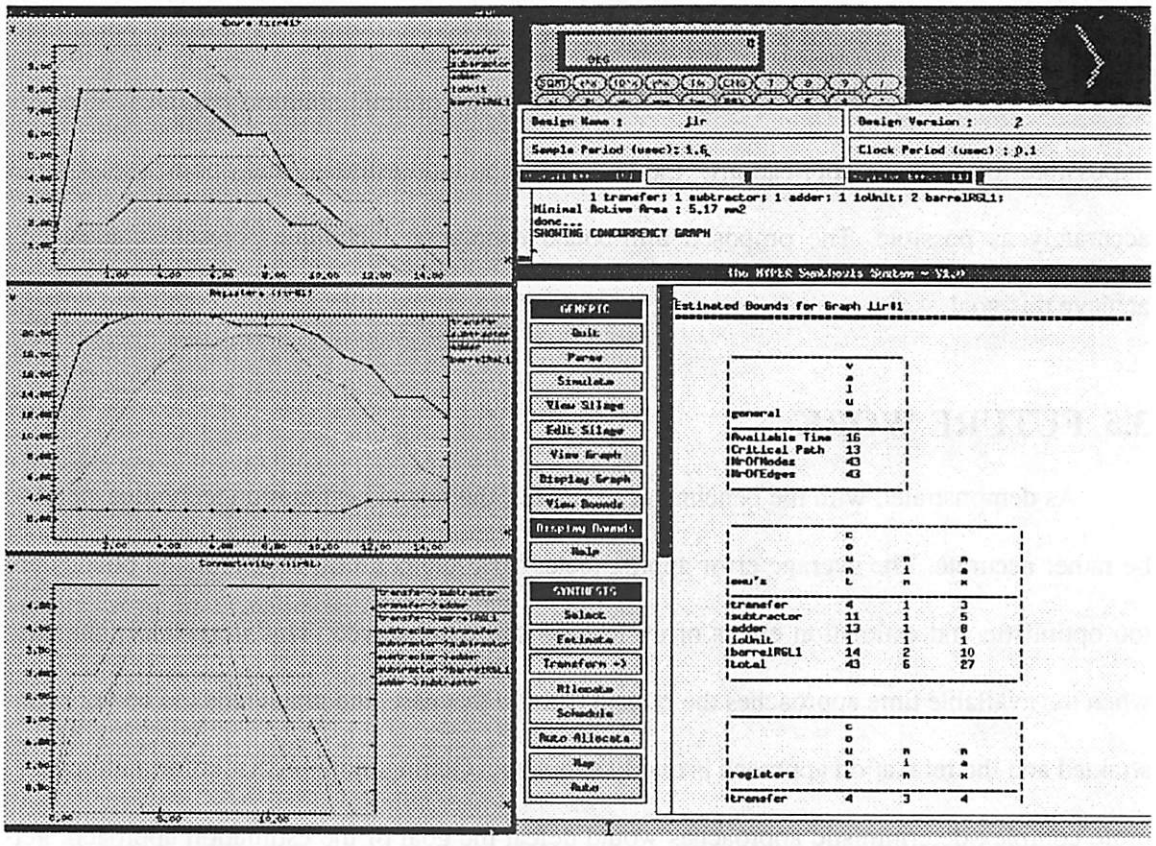


FIGURE 3.20. Screen Dump of HYPER User Interface During Estimation.

3.4.5 Assignment and Scheduling

Estimation can help to improve the quality of assignment and scheduling algorithms. For instance, the HYPER scheduler uses a relaxed scheduling as a heuristic to select a candidate node in a list scheduling process [Pot91b].

3.4.6 Synthesis Algorithm Validation

Most of the problems in high level synthesis (such as allocation, assignment and scheduling) have been proven to be at least NP-complete. As a result, it is hard to judge the quality of a proposed synthesis algorithm. Typically, benchmarks examples are used as a means to establish the performance of a technique. However, algorithms can be over-tuned to one particular benchmark ("the fifth order elliptical filter" syndrome). Furthermore, comparing algorithms against a particular benchmark establishes a relative measure, but does not result in an absolute idea of he quality of the solution. Short of actually determining the optimal solution (which is virtually impossible for complex applications), the best approach is to estimate that optimal solution as accurately as possible. The proposed min-bound estimation techniques present a means to achieve this goal.

3.5 FUTURE WORK

As demonstrated with the benchmark examples, the relaxed estimation techniques tend to be rather accurate. The average error approximates 10%. Sometimes, the predicted bounds are too optimistic and estimation errors of up to 67% can be observed. This is typically the case when the available time approaches the critical path. In that case, the graph tends to be very constrained and the relaxation approach is overly simplistic. Getting more precise information using more complex deterministic approaches would defeat the goal of the estimation approach: getting *accurate* results *fast*. This eliminates all approaches with a complexity larger than $O(N^2)$.

One way to get more precise results without extra computational complexity is to use a statistical approach. The results of the deterministic estimation process can be used as parameters in a statistical model, obtained through the analysis of a large number of examples, which span the complete design space. This approach does not result in bounds, but in an estimation of the location of the actual solution. Early experiments (over the same benchmark set as used

higher) indicate that an average error of 4.6% can be obtained with the maximal error equal to 18.7%. An extended research effort in this direction is currently under way.

We are also studying the use of the relaxation approach for the estimation of background memory bounds. These results will be extremely useful in the memory module selection, memory partitioning and transformation for memory optimization processes.

Finally, the reader might have observed that the majority of the estimation techniques presented focus on the computational part of the implementation. This is justifiable for high performance real time applications, where memory and data paths dominate the chip area. This is not the case however for control dominated applications. Some results in this area have already been reported in [Mli91].

3.6 CONCLUSION

A library of techniques to efficiently and accurately estimate the minimal and maximal bounds on the implementation cost of an application specific circuit have been presented. All algorithms have a complexity not larger than quadratic and the observed results on our benchmark set display an average error of approximately 10%.

We have demonstrated the application of those techniques in a variety of design synthesis areas, such as design space exploration, transformation selection and synthesis validation. It is the authors belief that the major impact of estimation will be in the higher levels of the synthesis process, such as algorithm and design style selection as well as design partitioning. It is our conviction that estimation will be one of the essential components in the system designers tool-box.

4

ALLOCATION, ASSIGNMENT AND SCHEDULING ALGORITHMS FOR HIERARCHICAL CONTROL DATA FLOW GRAPHS

4.1 PROBLEM DESCRIPTION

The path from the high level specification of an application specific design to the final implementation is long and complex. Even when looking only into the architectural synthesis part, it involves a multitude of problems, including the selection of clock frequency and hardware module set, partitioning, transformations, resource allocation, assignment and scheduling. All those tasks are computationally very complex, especially when posed with all constraints and all aspects imposed by real-life problems.

Although a standard terminology has yet to be agreed on, the following definitions are most common and are widely accepted. **Scheduling** is the task, which decides in which control step a given operation will happen. **Assignment** determines on which particular execution unit a given operation will be realized, from which register it will request its data and where it will send the result using which connection. **Resource allocation** is closely related to the above

tasks: it reserves the amount of hardware (in terms of execution units, memory registers and interconnect) needed for the realization. It might also determine the time available for the execution of the algorithm. Obviously, those three tasks are interdependent and the layered structure of NP-complete problems makes the overall problem extremely difficult.

4.1.1 Previous Work and New Issues

Almost all allocation, scheduling and assignment problems, even when posed in a highly restricted form, are at least NP-complete. The complexity of various versions of the scheduling [Joh83, Bla83], allocation [Iba88] and assignment [Gar79] problems has been treated extensively in the literature. Furthermore, those problems have been studied in great detail in the areas of software compilers [Gon77] and operations research [Law90]. However, the specific nature of high level synthesis imposes some very specific demands and constraints (e.g. the relationship between execution units, interconnect and memory), which prevents a direct use of those techniques.

Although high level synthesis is a relatively young area, numerous approaches to the above problems have already been proposed. Like in other CAD areas, all optimization techniques have been tried one by one, usually with increasing implementation complexity, more realistic problem modeling and higher level of success. They can be categorized into several groups, according to the underlying algorithm: explicit [Bar73] and implicit enumeration algorithms [Par86], various heuristics which use as soon as possible (ASAP) and as late as possible (ALAP) scheduling to obtain a global picture of the solution space [Goo87, Pau89, Sto89], integer programming [Bal89], various probabilistic approaches (e.g. simulated annealing [Dev89] and neural nets [Gul87]) and continuous relaxation techniques (e.g. linear programming with manual assistance [Har89] and gradient methods [Shi89]). A good overview of the multitude of approaches is given in [McF90].

The more than 70 published approaches represent both the impressive research efforts and the significant progress achieved. Still, several important issues and aspects of the problems are rarely, if at all, addressed.

First of all, practical experience indicates that an overwhelming majority of the applications require **hierarchical constructs**, being a conditional operation (IF-ELSE) and a loop construct (DO, WHILE, ...). Without them, only a very limited number of applications can be addressed. Flattening the graphs is not a reasonable solution, as it would create graphs with excessive numbers of nodes, resulting both in unrealistic implementations (huge number of states in the controller) and in inflated scheduling times. Although the majority of algorithms described in the literature can be used as a subroutine in a hierarchical framework, significant modifications have to be introduced and a significant number of new issues has to be addressed to achieve effective solutions. The most important of them is the observation that the optimization process has to consider the global, hierarchical graph and that it is not sufficient to optimize at the lowest level of the hierarchy.

Next, in VLSI technology it is essential to **simultaneously** address all **three components of the hardware cost** (being the number of execution units, memory registers and interconnect) [McF87]. Very few scheduling and assignment algorithms are doing this. Furthermore, it is necessary to consider during the allocation and scheduling not only the structure of the algorithm, but also the available hardware and its properties. For example, it is obvious that the allocation and scheduling for a floating point computation should favor both multipliers and adders approximately equally. However, when fixed point computational elements are used, more attention should be paid to reduce the number of multipliers, where the number of adders is of less importance, due to the smaller cost. Even for identical applications, high quality solutions for those two cases will be very different.

Finally, it is important to picture allocation, scheduling and assignment as just a sub-task in the overall synthesis process. In order to be useful, it is important that those tools provide an adequate **information feedback** to the synthesis framework and/or the user. This feedback for instance includes information on eventual bottlenecks in the algorithm (such as insufficient time allotment, lack of parallelism or the under-utilization of hardware) [Rab91a]. This information is especially important for the module selection and graph transformation environments (for instance, should more pipelining be applied?).

4.1.2 Problem Formulation

This section describes the overall formulation of the addressed problem in terms of the input, the constraints and the objective.

4.1.2.1 Control data flow graph

The HYPER high level synthesis system (in which all described algorithms are incorporated) [Rab91a], uses a control data flow graph (CDFG) syntax to describe the semantics of the algorithm (after translation from a high level language). The CDFG represents the algorithm essentially as a flow graph, with nodes, data edges, and control edges. The nodes represent data operations (including memory read and writes and memory address calculations), while the data edges represent data precedences between nodes. In addition, control edges are introduced to enforce extra precedence rules, for example that operation A has to precede operation B with at least K cycles.

Aside from the standard algebraic operations, the CDFG allows a number of macro control flow operations such as loops and if-then-else blocks. The introduction of those control statements result in a hierarchical graph. The body of a loop or a conditional is represented by a sub-graph, which is contracted into a single node at the next hierarchy level up (Figure 1).

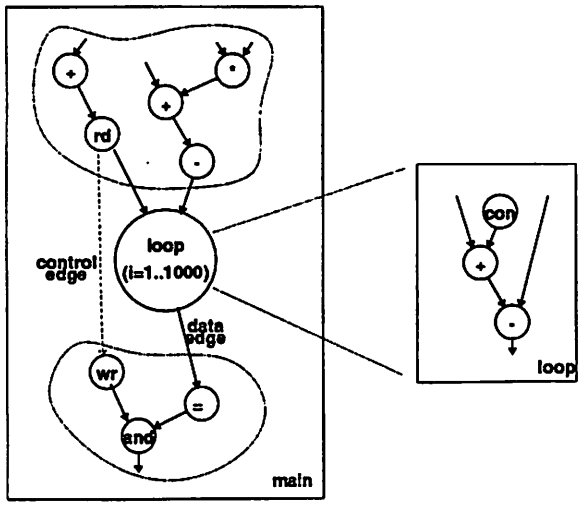


FIGURE 4. 1 Hierarchical Flow Graph Model

In order to handle hierarchical graphs, we first transform the CDFG. Each loop and conditional (as well as subroutine) is treated as a single block. Operations outside the loops are clustered into blocks as well (Figure 1). Now we can represent any program as a hierarchical graph with blocks as nodes. At the lowest hierarchy level (called the leaf graph level from now on), a node will represent only a single operation. At the higher levels in the hierarchy however, each node represents a sub-graph on itself.

4.1.2.2 Hardware graph

The goal of the synthesis process is to produce an architecture which implements the application described in the CDFG. The architecture is fully characterized by the number of execution units, the registers in register files and the list of required interconnects. We assume that a direct mapping between the selected architectural primitives and the silicon area exists in the form of a cost function. Establishing a precise cost function is a difficult task, mostly because of the interconnect component, which involves the complete placement and routing process. We are currently assuming that the cost function is provided by the user. (Research addressing this problem using a statistical modeling technique is also under way.) Notice also that the current estima-

tion does not include the cost of the controller either. This could become a problem for control-oriented applications.

The hardware model assumed in the proposed algorithms is very general: execution units can be multi-functional, take an arbitrary number of control steps, be chained (connected without intermediate registers) and be pipelined to an arbitrary extent.

4.1.2.3 Objective function

The high level synthesis literature defines three different combinations of objective functions and constraints: (1) minimize the execution time, given the hardware constraints; (2) minimize the hardware, given the timing constraints; or (3) find a solution which simultaneously satisfies both the timing and hardware constraints. The third combination can be used as a subroutine during the search through the time space or the hardware space to solve the first two formulations. Therefore, we opted to address only this formulation for the leaf graphs (graph without hierarchy). Using this basic routine, any of the above objective functions can be implemented, both for hierarchical and leaf graphs. In the rest of the paper however, we will assume (without loss of generality) that we pursue the second objective function for the hierarchical problem.

4.1.3 Solution Organization and Strategies

The overall organization of our approach is pictured in Figure 2. The outer loop of the algorithm performs an allocation search through the architectural space. For each proposed hardware solution, it invokes a program to optimally distribute the available time over the leaf graphs. Given a hardware allocation and a time distribution, the leaf graph assignment and scheduling routine is now invoked on each of the leaf graphs. This routine in itself is organized as a loop: multiple assignments are proposed. Only attractive assignments are presented to the scheduler. Both assignment and scheduling routines provide feedback information (for instance

which hardware unit is in short or ample supply or which leaf-graph was hardest to schedule), used to guide the allocation search process.

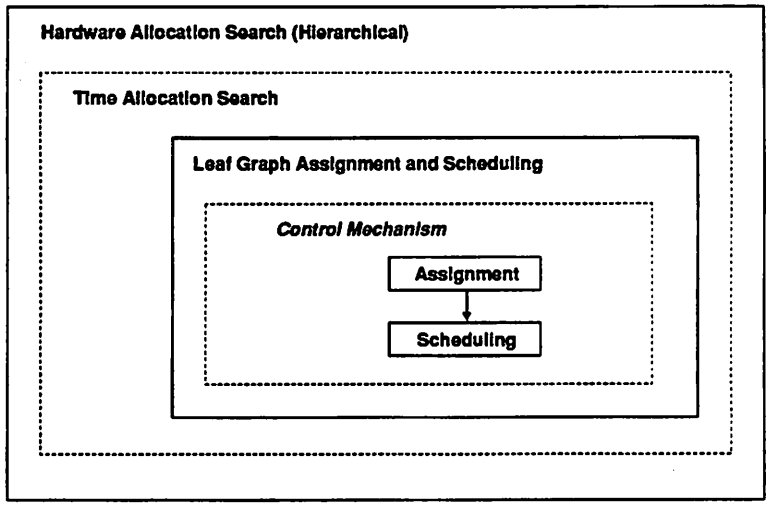


FIGURE 4.2 Overall Organization of Allocation, Assignment and Scheduling Process

Notice that in our approach assignment is performed before scheduling. Although very tempting and often advocated, the idea of combining scheduling and assignment in one step is not necessarily superior to treating them separately. Assessing the cost of scheduling a node on a particular hardware unit is much easier when the assignment is known. This is definitely the case when considering simultaneously EXU's, registers and interconnect. For instance, how to assess the effect of scheduling a node on the interconnect cost, when it is not known where the fanout of that node will go? Therefore, combining both approaches does not necessarily lead to a better solution and will result in a more complex and convoluted code implementation. Furthermore, as already mentioned, both scheduling and assignment are NP-compete problems. The combination of both of them is even more complex and will require a superior optimization mechanism.

It is very important to stress that when the approach is taken to execute the algorithms in sequence, it is essential to anticipate the consequences of decisions made during the first applied algorithm on the second one. This is one of the main reasons why we opted for performing **assignment before scheduling**, in contrast to majority of the published approaches. As it will be

described later, we succeeded to accurately characterize the probability of a successful scheduling for a given assignment. Another reason for this decision is that a hierarchical scheduling environment produces additional constraints of a spatial nature (for instance, the correct interfacing between sub-graphs requires the locking of the input/output variables in fixed registers). Those assignment constraints are more easily handled in an assignment first approach.

Several other high level synthesis systems also perform assignment before scheduling. Most notable among them is CMUDA [McF86]. However, the approach presented here differs from the previously published techniques in several important aspects. Both the form and the validation of the objective function are new, as well as the used optimization algorithms. Furthermore, the resource allocation module enables efficient interchange of feedback information between assignment and scheduling. Finally, and maybe the most importantly, the just mentioned motivation behind this ordering of high level synthesis task is new.

4.1.4 Chapter Organization

The remainder of the chapter is organized in the following way. First we describe the assignment and scheduling algorithms for leaf graphs and the accompanying control mechanism. This is followed by a discussion of the hierarchical framework, which combines the leaf graph algorithms with a global hierarchical allocation search. Finally, experimental results are presented and analyzed.

4.2 ASSIGNMENT

This section describes the two-phase assignment algorithm. The first phase of the process proposes an optimized assignment using a probabilistic rejectionless anti-voter algorithm. The role of the second phase is to analyze the proposed solution in order to:

- discriminate between proposed solutions

- provide feedback information to the hierarchical allocation framework and the scheduling.

After a discussion of the objective function of the assignment process, the two phases will be presented in detail, followed by a analysis of the computational complexity of the algorithm. A statistical computational study will demonstrate the effectiveness of the proposed approach.

4.2.1 Objective Function

The goal of the assignment process is to find, for a given hardware allocation, the assignment which will make a successful scheduling as likely as possible. To measure the likelihood of successful scheduling given the assignment, we introduce a measure *badness* and two properties *disastrous* and *bad* for each CDFG node. The *badness* indicates the predicted level of difficulty to schedule a particular node. It is a function of the number of other operations, which vie for the same resource (which can be an execution unit, register or bus) and which can be scheduled simultaneously. In order to schedule two operations during the same cycle, those operations should obviously be assigned in such a way that they do not require the same resource instance.

Any operation has to be scheduled in the interval between its ASAP and its ALAP times, which can be easily obtained using topological ordering (using depth first search), and leveling according to inputs and outputs. The slack of an operation is defined as the difference between the ALAP and the ASAP times. In order to maximize the overall resource utilization, we want to minimize the overlap of the intervals for operations, which compete for the same resource and which do not have any precedence relationship between them. Therefore, we define the *badness* of CDFG node A using the following formula:

$$badness(A) = \sum_{B \in S} \frac{O_{AB}}{A_{SL} \times B_{SL}} \qquad \text{(EQ 1)}$$

where S is set of nodes B competing with node A for a resource of the same type, O_{AB} is the length of overlap between the slack (SL) intervals of nodes A and B and A_{SL} and B_{SL} are the slacks of operations A and B. The rationale behind this formula is that it is a computationally efficient approximation of the probability to schedule operations A and B at the same time. The badness of nodes, for which there exists no choice (when there is only one available resource of the type needed by the operation or when the assignment is fixed due to global constraints), is set to 0. The total badness of a given assignment is the sum of the badnesses over all CDFG nodes.

When there exists an assignment for a particular node, which would improve (reduce) the badness of that node with respect to the current assignment, the node is denoted as *bad*. This property can be easily determined by comparing the current badness with the worst case badness of that node over all assignments. When a node can not be scheduled, regardless of the scheduling technique, due to an inappropriate assignment, the node gets the *disastrous* property. This is for instance the case when two nodes, bound to happen in the same cycle, are assigned to the same resource. When we want to improve the total badness, it is obvious that the *bad* and *disastrous* nodes are the prime candidates.

The above ideas are illustrated with the example in Figure 3. It is assumed here that each operation takes 1 cycle and that 5 control steps are available. Suppose that at a certain phase in the assignment process, we have obtained an assignment, as given in Table 1. The Table also includes the ASAP and ALAP times for each node. Looking at node B, we see that nodes A, C and D can not be scheduled at the same time (because they are on a direct path to or from B), and therefore they are not on the competitor list of node B. Node I is not a competitor for node B due the fact that its execution does not overlap with the execution interval of node B ($O_{BI} = 0$), while E is excluded because it does not vie for the same resources as B. However, nodes F, G and H are all competitors of B. G is performed on the same unit as B, while F and H are sending their results to the same destination (register file). The normalized influence (or potential bad-

ness computed using (1)) is 1/3 for node G, 1 for node F and 1/4 for H. Node B therefore has a total badness of 19/12. If we change assignment of node B from adder 2 to adder 1, then the badness is reduced to 5/4, due to the fact that now only nodes F and H require the same resource as node B.

Node	Type	ASAP	ALAP	Assignment
A	+	1	2	1
B	+	2	3	2
C	+	3	4	1
D	+	4	5	2
E	*	1	2	3
F	*	2	3	3
G	+	1	3	2
H	+	3	4	1
I	+	4	5	2

Table 2: Assignment Instance for Example of Figure 3

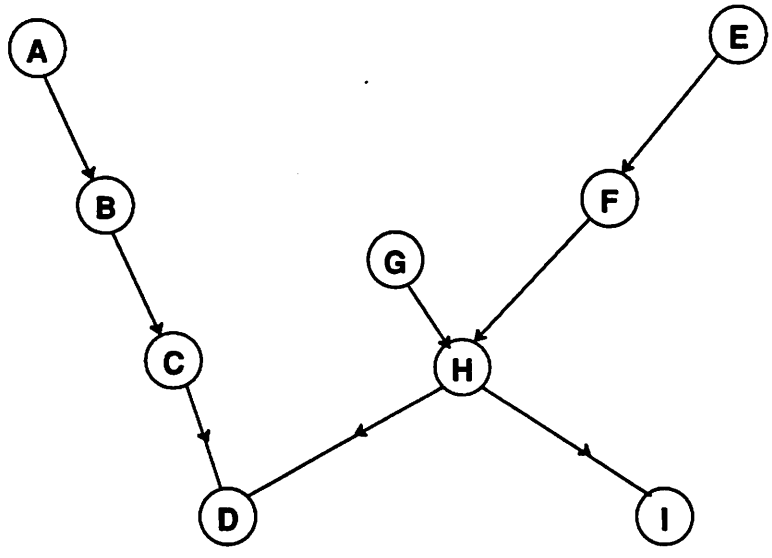


FIGURE 4.3 Example Flow Graph.

4.2.2 Probabilistic Rejectionless Assignment Algorithm (Phase 1)

In order to efficiently solve the assignment problem, a novel probabilistic algorithm has been developed. The basic properties of the algorithm can be summarized as follows: First a random assignment is performed, where each node is assigned to a hardware instances of the required type. After calculating the badness of all nodes, and building the lists of bad and disastrous nodes, we proceed in a way common to probabilistic iterative techniques: the algorithm tries to reach the optimum by applying a sequence of basic moves (in this case, changing the assignment of a particular CDFG node). However this is where similarity with popular methods such as simulated annealing and Boltzmann machine ends.At each step, a node is selected from either the list of bad or the disastrous nodes in a random way, probabilistically favoring disastrous nodes. After that, a move is selected, once again on a random base. The chance, that a certain node and a certain move are selected, is however made proportional to the overall badness of the node and the reduction in badness due to the move (measured as the reduction of the overall badness). It should be realized that moves increasing the overall badness can occur, although the chance for this to happen is small and inversely proportional to that increase.

There are two major differences between this technique and simulated annealing: First, the proposed move is always accepted, where in simulated annealing moves are generated on a totally random base and a large percentage of them is rejected. This technique is therefore *rejectionless* [Gre86]. Secondly, at each step, the algorithm uses all information about how much a node or a move can contribute to the reduction of the objective function (through the badness and disastrous lists). This aspect of the algorithm is inspired by the highly successful *anti-voter* probabilistic algorithms for graph coloring [Pet89]. The combination of the rejectionless and the anti-voter nature of the algorithm improves the efficiency of the algorithm dramatically, and the run time for examples of up to several hundred nodes is less than 0.1 second on a SPARCstation 2. The algorithm is halted when no improvement is detected for at least k steps, where k is the

number of nodes in both the bad and disastrous nodes lists. It can easily be augmented with an annealing and tabu mechanism. However, the observed performance was more than adequate, so that we decided not to use those ideas here.

Anti-voter algorithms have several noble properties, which are preserved in the proposed algorithm. Among them is the algorithm property that it converges to the optimum solution with probability 1, when the running time is sufficiently large [Kar88]. This is proven with the following simple argument. Suppose that we have an assignment. If this assignment is not optimal, then, by randomly picking the correct moves one by one, we can make a transition to the optimal solution. For this we have small, but finite probability. If we are not lucky after at most n steps (n is the number of parameters necessary to fully describe the assignment problem), we can apply the same reasoning. In this way, the probability to reach the optimal solution can be made as large as necessary (by trading off with run time). It is interesting to notice that in practice the algorithm converges extremely fast to a high quality solution.

In summary, the assignment algorithm can be described with the following pseudo-code:

```
Generate Random Initial Assignment;
Form list B of bad elements and list D of disastrous elements;
While (stopping criteria not satisfied) {
  Pick a random element A from B or D;
  Pick a new, random assignment for A;
  Update solution as well as lists B and D;
  }
```

4.2.3 Assignment Evaluation using Relaxed Scheduling (Phase 2)

The probabilistic rejectionless anti-voter assignment algorithm creates assignments based on the ASAP-ALAP information (as is typical in most assignment/scheduling algorithms). Although this is a useful measure, it tends to be overly optimistic: due to conflicts between operations vying for the same resource, some operations will have to be postponed, resulting in a reduced ALAP-ASAP interval (which is ignored in the previous formulation). The effect of the

conflicts over a particular resource on the earliest execution time of all nodes, which need this particular resource, can be easily evaluated by performing a simple scheduling, considering only those nodes which require this particular resource. Optimal scheduling, considering only a single resource, is a problem of polynomial complexity $O(n\log(n))$ [Law76], with n the number of nodes in the sub-graph under consideration, which is normally small. It is well known that this optimal schedule can be obtained by using a list scheduling with the smallest ALAP time as the priority measure (also called slack based scheduling). The obtained ASAP time (called T_{RASAP}) is still optimistic (since only one resource is considered at the time), but is significantly more precise than the original ASAP.

This procedure, which we called *relaxed scheduling* provides more accurate and less optimistic information about the quality of the proposed assignment and helps to generate an effective quality measure, used to discriminate between assignments.

As a result, we can introduce one more CDFG node measure, called the expected *node scheduling difficulty* (SD), which equals (for a node A):

$$SD(A) = \frac{1}{min(T_{ALAP} - T_{RASAP})}, \text{ if } (T_{ALAP} > T_{RASAP}) \qquad \text{(EQ 2)}$$

$$= 1, \text{if}(T_{ALAP} = T_{RASAP})$$

$$= M, \text{if } (T_{ALAP} < T_{RASAP})$$

where M is a large number (>> 1). This measure can be interpreted in the following way:

- SD(A) small: A has a strong chance to be scheduled without violating timing constraints.
- SD(A) larger, but smaller than 1: scheduling is possible, but unlikely.

- SD(A) > 1: scheduling is impossible.

The total expected scheduling difficulty for the whole CDFG is the sum of expected scheduling difficulties over all CDFG nodes. This information can be used by both the allocation as well as the scheduling routines.

4.2.4 Assignment Effectiveness

To verify the effectiveness of the assignment objective function, the expected scheduling difficulty measure as well as the proposed algorithms, the following assessment procedure was executed. For a number of examples, we generated and evaluated 250 fully random assignments. For each of those, the objective function and the expected scheduling difficulty were computed and the scheduler, described in the next section, was applied. The results for one example (a 7th order IIR filter with an available time of 16 control cycles and 2 adders, 2 barrel shifters and 1 subtractor allocated) are shown in Figure 4. A high level of correlation between the objective function and the scheduling success is evident. Similar patterns have been observed for all other examples.

The computational complexity of the *relaxed scheduling assignment* equals $KL_k \log(L_k)$, where K is the number of relaxed schedules (or the number of resources) and L_k is the number of CDFG nodes, which are using resource K. Although the worst case complexity (when all nodes are assigned to the same resource) is quadratic, the average case has almost linear complexity. The overall complexity of the probabilistic assignment is also very low and can be controlled using the stopping criteria.

4.3 SCHEDULING

This section describes the probabilistic constructive scheduling algorithm. It is assumed that both resource allocation and assignment have been performed prior to the scheduling, as is described in the respective sections. At the end of the section we will discuss the computational

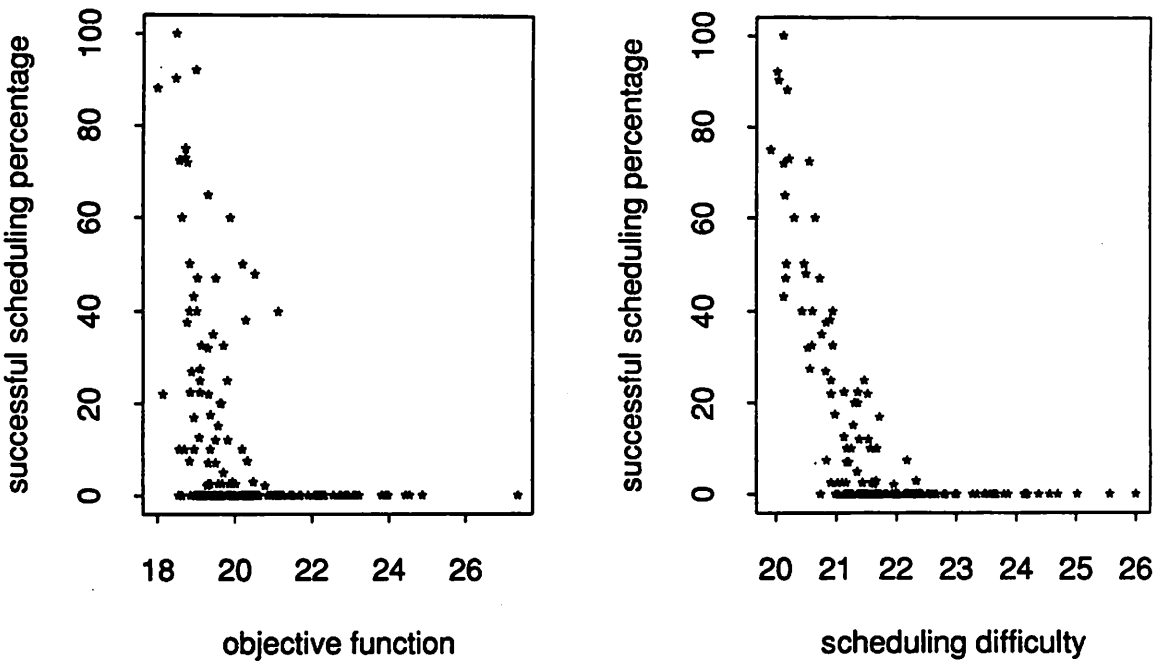


FIGURE 4.4 Correlation between Assignment Objective Function and Scheduling Success and between Scheduling difficulty and Scheduling Success

complexity of the proposed algorithm, while experimental results will be discussed in a separate section.

One of the most controversial issues in scheduling is whether to use a list scheduling framework. The list scheduling framework [McF90] significantly reduces the implementation complexity of a scheduler (versus, for instance, a force directed scheduler), but it is often seen as imposing extra constraints and hence leading to locally optimal solutions. However, a solution obtained by any other scheduling framework can also be obtained by a list scheduler. To achieve a globally optimal solution, the list scheduler has to adhere to two conditions. First of all, decisions for a certain control step have to be made with respect to all nodes in the graph, not just to the candidates for that particular control step. Secondly, it is necessary to have a mechanism to postpone the scheduling of a particular node, even though hardware resources are available dur-

ing the control step under consideration. Postponement is sometimes necessary, either due to multi-cycle units or memory requirements. We will discuss both cases in the course of the ensuing description.

During the scheduling all timing constraints are honored. If the scheduler is not capable to find a feasible schedule satisfying all timing constraints, the resource allocation routine (see Section 5) increases the available amount of hardware.

4.3.1 Constructive Probabilistic Scheduling Algorithm

The scheduling starts with a pre-processing step which aims to establish a global scheduling importance ranking for all CDFG nodes. A transitive fanout list is built for each node using a depth first search (for node A, this list contains a list of all nodes depending upon A, and is called *fanout(A)*). As already explained in the assignment section, all scheduling algorithms, proposed until now in high level synthesis, rely on ASAP and ALAP information. However, the relaxed scheduling executed during the second assignment phase provides us with more precise and less optimistic information for each node (being the expected scheduling difficulty *SD(A)*). This information can be used to build global measure for the scheduling difficulty of a node, called GSD(A):

$$GSD(A) = \sum_{B \in fanout(A)} SD(B) \qquad \text{(EQ 3)}$$

This measure is motivated by the following reasoning: Even, when a particular node is not denoted as critical, its global criticality is also influenced by the criticality of all nodes, which have this node as a prerequisite for scheduling.

After the ranking, the traditional list scheduling scheme is followed: at every time step, a list of candidate nodes is constructed and, as long as resources are available, the nodes with the best ranking are selected. After the exhaustion of the resources, the candidate list and the

resource availability are updated and the algorithm proceeds to the next control cycle (as long as there are cycles or candidates available). The *Global Scheduling Difficulty* (GSD) is used as the prime ranking measure. However, some additional information is used during the candidate ranking:

(1) In order to randomize the approach, a random component is added to the just described deterministic one. During the first scheduling for a given assignment, this component is set to 0, and it slowly increases with further attempts. This randomization often results in small improvements of the produced schedule.

(2) If a node has an ALAP equal to the current step, it is assigned the maximal difficulty.

(3) If two multi-cycled operations A and B, assigned to the same EXU, have overlapping time intervals, an operation with the lowest rank will be postponed to reserve the resource for more critical operation.

(4) The GSD-measure ranks the nodes with respect to EXU and interconnect availability. Memory is only addressed indirectly (since all variables which use a particular interconnect will be stored in the same register file). This ranking ignores the limited size of the register files and the finite life times of the variable. Ranking with respect to memory availability can only be accurately addressed dynamically during the scheduling. We discuss this issue in more detail in the following section.

The scheduling algorithm is summarized with the aid of the following pseudo-code:

```
list_scheduling_framework {
  candidate_ranking();
  candidate_list_initialization();
  for (time = 0 ... max_time){
      update_candidate_ranking();
      while (resource_status == YES){
          schedule_best();
          update_resource_status();
      }
      update candidate_list();
      update_resource_status();
  }
}
```

4.3.2 Register Binding and Estimation

After a successful scheduling, it is necessary to determine exactly in which register within the assigned register file each variable will be stored. This task is called *register binding*. It is easily seen that two variables can only share the same register if they have disjoint life intervals. The register binding problem can be now transformed into either a graph coloring or a clique partitioning problem in the following way (as is well known in the literature): Variables are mapped to nodes of a graph. The problem is transformed into a clique partitioning if an edge is introduced between two nodes, corresponding to variables with disjoint life times. If an edge is introduced when the life times are not disjoint, the problem is transformed into a graph coloring. Although graph coloring is NP-compete for circular arc graphs, which correspond to the register binding problem, there exist very efficient algorithms which guarantee the optimal solution with probability 1 for several broad graph classes [Tur88, Dye89]. We selected Turner's version of the Brelaz algorithm [Tur88], due to its simple and efficient implementation.

Graph coloring [Spr90, Sto91] and clique partitioning [Tse86] are often used techniques for modeling hardware (and in general resource) sharing. It is easy to see that those techniques are equivalent: graph coloring of a graph is equivalent to the clique partitioning of its complement graph and vice versa [Gar79]. Our use of this technique is restricted to the optimization of the number of registers in a register file.

On the other hand, during the scheduling process, we also need information on how many registers are in use in each register file. This will influence the candidate ranking, as discussed in the previous section. This estimation problem is addressed in the following way. At each time step during scheduling we know how many variables are alive in each register file. When a new variable appears (or will appear due to a scheduling decision), we know that it must claim an empty register, not in use by the live variables. As in the register binding, we can transform this problem into a graph coloring one. In contrast to the binding problem, we are not interested in an

exact coloring here. Only information about the total number of colors needed is required. This problem is well understood in the theoretical literature [Bol85, McD91], and the results are easily summarized: To color a graph G containing n nodes and p edges, with probability 1, it is necessary and sufficient to use $\frac{n}{2\log_p(n)}$ colors, where n equals the number of nodes and p is number of edges divided by $\frac{n(n-1)}{2}$. This measure is evaluated dynamically during the list scheduling process and is used to alter the ranking of the nodes in a such way that no overflow occurs on the register files.

4.3.3 Scheduling Algorithm Complexity

The initial node ranking process (building of the fanout lists) needs, in the worst case, $O(n^2)$ time, since each node can have a transitive fanout of at most n nodes. Assuming that the time needed by the ranking and selection mechanism is proportional to the number of nodes in the candidate list for a particular time step, the worst case complexity of the list scheduler can be obtained by using an adverse configuration approach. The scheduler will display the largest run time, when all nodes in the CDFG have an ASAP time of 1 (and therefore an ALAP time equal to the total available time). Then, at the worst, we have $O(n)$ nodes as candidates, and since the number of available cycles is smaller than the number of nodes, once again, $O(n^2)$ time is sufficient.

4.4 LEAF GRAPH SCHEDULING AND ASSIGNMENT CONTROL MECHANISM

The leaf graph control mechanism iteratively invokes the assignment and scheduling processes. The scheduling routine is only fired when the proposed assignment has sufficient chances for success as expressed by the objective function. A careful study of the assignment statistics (as shown for instance in Figure 4), leads to the conclusion that small improvements in the

objective function are rather meaningless. We therefore selected the following threshold function to reject or accept proposed assignments:

$$Threshold = median + 0.1 \times (best - median), \qquad \text{(EQ 4)}$$

where *best* is the best assignment objective function, and *median* is median value of the objective functions during all previous assignments. We use the median measure instead of the average to improve the algorithm robustness (it is only very slightly more computationally intensive than the average calculation). If the assignment includes *disastrous* nodes, the scheduling is not invoked, since there are no chances for a successful completion.

The control mechanism has two parameters: the maximum number of invocations of the assignment (for each allocation), and the maximum number of scheduling attempts per assignment. On all examples tried, we obtained the final solution within 20 assignment cycles and at most 10 scheduling attempts per assignment. Because the scheduling routines provide important information to the hierarchical allocation, we invoke them at least once per allocation, even if all assignments are rejected.

4.5 HIERARCHICAL HARDWARE ALLOCATION

The hierarchical allocation is responsible for three major tasks:

- to propose a hardware allocation with minimal cost such that all sub-graphs can be scheduled and all timing constraints are met;
- to distribute the available time over the leaf graphs in a global and optimal way;
- to combine the assignments and the schedules of the leaf graphs into a hierarchical assignment and schedule, which obeys the temporal and spatial constraints between those sub-graphs: for example, the output of one sub-graph should be put in the right register for usage in the corresponding graph.

While the former two tasks are complex optimization problems, the last one amounts to bookkeeping: Leaf graphs interchange data according to a pattern, dictated by the hierarchical

CDFG structure. During the assignment process for a leaf graph, it is necessary to send the output data to the appropriate register files, such that it can be properly accessed by the fanout in different leaf graphs. Also, it is necessary that reserved resources (memory registers with values required by other blocks) are locked, so that they will not be overridden. Those issues are resolved by building lists of reserved and locked resources.

The hardware and time allocation process is organized as two nested loops. The inner loop search distributes the available time over the blocks as well as tests and generates a final solution using the leaf graph assignment and scheduling subroutines. The outer loop searches the architectural space, proposing different hardware allocations. The search process is organized in this way because the inner search is the more constrained one: in order to allocate more time to a particular block we have to reduce the available time of some other block(s). During the hardware allocation, on the other hand, we have the freedom to add or remove hardware resources without restrictions.

4.5.1 Leaf Graph Time Allocation

Before the assignment and scheduling of the leaf graphs can be attempted, the available time for each of them has to be determined. The goal of this phase is to either produce a feasible schedule for the allocated hardware or to establish the proof that a feasible schedule is non-existing, regardless of the time distribution. This problem can be very efficiently solved using the following approach. As an initial solution, the available time is distributed over the leaf graphs, proportional to the complexity of each of the sub-graphs (using the critical path and the number of operations per cycle as the most important measures). A scheduling is impossible if the overall critical path (obtained as a combination of the critical paths of the sub-graphs) is larger than the available time. If feasible, an assignment and scheduling attempt is executed. The feedback of this process is used to determine the next step: When the scheduling of a leaf graph was unsuccessful, we add as many control cycles as the ratio of the number of unscheduled nodes

over the number of execution units. This time is removed from the successful sub-graphs, which either had spare time or were under a relatively light stress. This procedure is repeated until either a feasible schedule is obtained or no candidates with spare time are left. This procedure converges extremely rapidly and takes at most 2 to 3 iterations for all our benchmarks.

4.5.2 Global Hardware Allocation

The hierarchical hardware allocation process uses once again a probabilistic search process with as prime objective function the minimization of the hardware cost. The search is organized as a two phase process: starting from an initial solution, hardware is added until a feasible solution is obtained (this is checked using the Time Allocation Module, described above). Once a solution is obtained, unnecessary hardware resources are iteratively removed. For the quality of the approach three criteria are essential: the initial solution and the criteria for adding and removing hardware.

The initial solution uses the absolute resource min-bounds as generated in the complexity estimation phase of the synthesis framework [Rab90]. Immediate scheduling success proves that the obtained solution is also the optimal one.

When failing however, feedback information from the assignment and scheduling routines is used to guide the search. During the scheduling, statistical information is collected on how often it happened that the scheduling of an operation was postponed due to the unavailability of a particular resource (EXU, interconnect, memory). Obviously, in order to increase the chances of scheduling, it makes sense to add this resource, which was in greatest demand and shortest supply. On the other hand, the addition of cheap resources should be favored over expensive ones to minimize the implementation cost. To balance between those two requirements, we probabilistically choose among the candidate resources according to the ratio demand over cost, i.e. we favor inexpensive and high demand - short supply resources. The same measure can be

used to decide exactly what type of resource to add (what EXU or bus, or to what register file). For execution units and busses however, a more precise measure is available in the form of the total scheduling difficulty for resource R (with $TSD(R) = \sum_{A \in R} SD(A)$).

Once a feasible solution is reached, the reduction phase is started. Some resources might be in over-supply and can be reduced. The reduction process proceeds in a greedy fashion: we try to reduce each resource class, one by one, in decreasing order of their cost.

The just described probabilistic procedure is very simple and fast. More sophisticated procedures were originally envisioned, such as the described rejectionless anti-voter approach or simulated annealing. The experimental results however suggest that the increased computational time of those approaches is not justified for this problem.

4.6 EXPERIMENTAL RESULTS

One of the most difficult questions in CAD is the assessment of the quality of a proposed algorithm and a corresponding program implementation. Since the problems are usually NP-complete (or worse), it is difficult to find the optimum solution. Sometimes, standard benchmarks have been defined, but those are usually established when the research area is more mature. Even when benchmarks are available, they are more targeted towards the comparison between algorithms, then to answer the question how good the algorithm is by itself.

The assessment of proposed algorithms is an especially acute problem in high level synthesis. The most common procedure is to take a few (often only one or two) examples and to conclude that the proposed algorithm produces a very good solution, due to the fact that it slightly outperforms previously published algorithms with respect to either speed or solution quality (the "fifth order elliptical filter syndrome"). Obviously, this approach does not guarantee that the next example will be solved successfully. To more adequately measure the algorithm

performance, we are using three basic tools: estimations, diverse examples, and robust parameters in the algorithm.

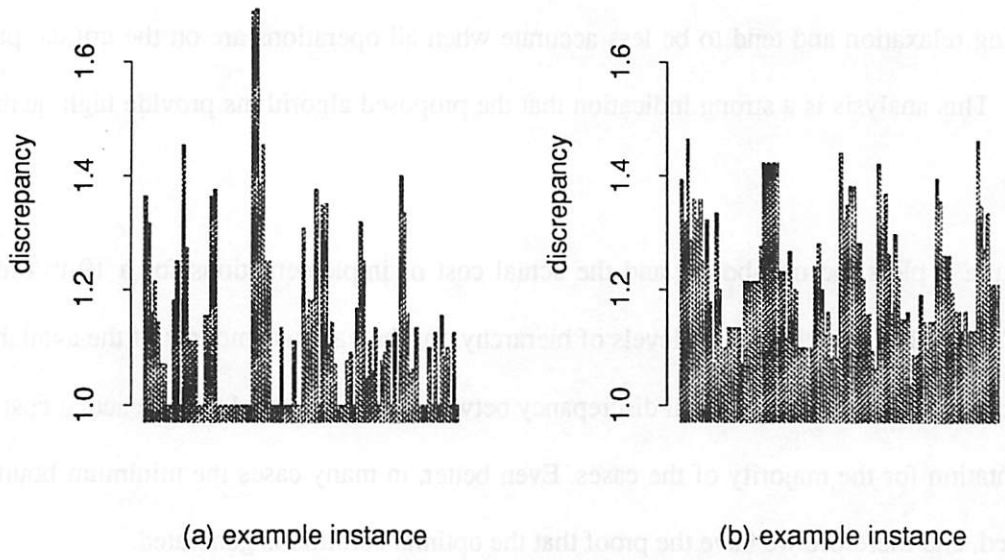
4.6.1 Estimation

As already mentioned, the major difficulty in the performance assessment of algorithms for an NP-complete (or more complex) problem, is the fact that the optimum solution is not known. However, for the allocation, scheduling and assignment problem, it is possible to establish a sharp lower bound on the minimum required hardware, i.e. the best possible solution [Rab90].

When the lower bounds are achieved, we actually have the proof that the algorithm produced the optimum solution. If not, we know that either the minimum bounds are not sharp enough or that the algorithm is not producing the optimal solution. However, if the gap between the bounds and the solution proposed by the algorithm is small, it indicates a high probability that a good solution is generated.

The assessment of the proposed allocation, scheduling and assignment algorithm using lower bounds is illustrated in Figures 5a and 5b. Those graphs plot for 100 examples the ratio of required versus min-bound cost, respectively for execution units (Figure 5a) and registers (Figure 5b). The 100 examples were generated by gradually increasing, for seven standard benchmarks, the available time, starting from the critical path. The ratios for interconnect are not presented, due to the lack of an appropriate cost function.

Several conclusions can be drawn from the analysis of those figures. First of all, the algorithms are very often capable of achieving the minimal bounds. The average discrepancy is 12.54% and the median discrepancy is 9.09% (in 39% of cases the optimum solution is achieved) for execution units. For the registers the average discrepancy is 22.41% and the median discrepancy is 20.8%. Also, the algorithms are very consistent. We noticed only one



(a) example instance (b) example instance

FIGURE 4.5 Ratio of required hardware cost versus the estimated minimum cost for execution units and registers for 100 examples.

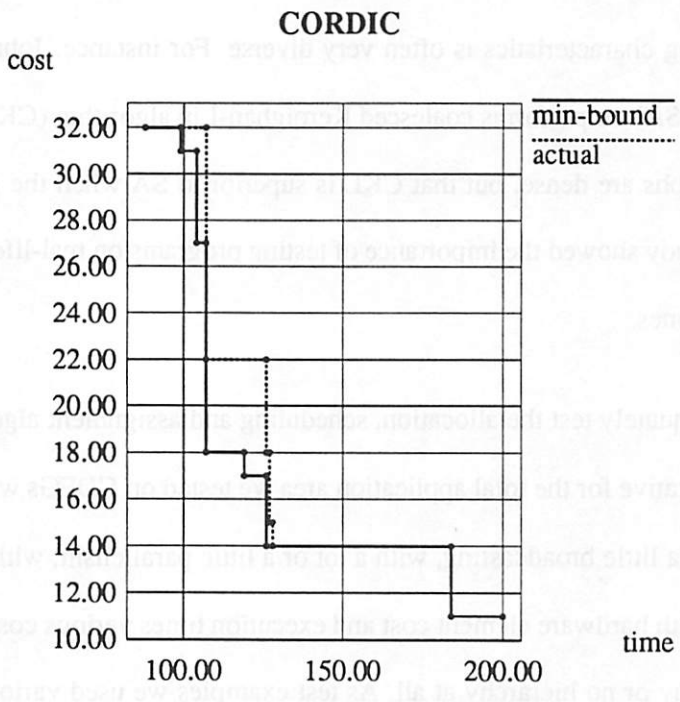


FIGURE 4.6 Discrepancy between the min-bound and the actual cost of implementations for a 19-th order CORDIC algorithm for the various amounts of the available time.

clear area where the algorithms had difficulty reaching the min-bounds: when the available time is close to critical path time. However, in those cases, it is much more likely that the discrepancy

is due to non-sharp bounds than due to the scheduling algorithms used (the bounds are also computed using relaxation and tend to be less accurate when all operations are on the critical path [Rab90]). This analysis is a strong indication that the proposed algorithms provide high quality solutions.

Figure 6 plots the min-bound and the actual cost of implementations for a 19-th order CORDIC algorithm, which has two levels of hierarchy, for the various amounts of the available time. Once again, we see a very small discrepancy between the min-bound and the actual cost of implementation for the majority of the cases. Even better, in many cases the minimum bounds are reached, and therefore we have the proof that the optimal solution is generated.

4.6.2 Diverse Examples

It is a well known fact that the relative performance of particular algorithms for problem instances with varying characteristics is often very diverse. For instance, Johnson showed that simulated annealing (SA) outperforms coalesced Kernighan-Lin algorithm (CKL) in graph partitioning when the graphs are dense, but that CKL is superior to SA when the graphs are sparse [Joh89]. The same study showed the importance of testing programs on real-life instances versus randomly generated ones.

In order to adequately test the allocation, scheduling and assignment algorithms on examples that are representative for the total application area we tested on CDFGs with few and many nodes, with a lot and a little broadcasting, with a lot or a little parallelism, with a few and many timing constraints, with hardware element cost and execution times various cost ratios, with several levels of hierarchy or no hierarchy at all. As test examples we used various DSP, telecommunication, information theory, numerical analysis and algebra tasks. It is interesting to notice that transformations (associativity, commutativity, distributivity, loop unrolling, retiming and

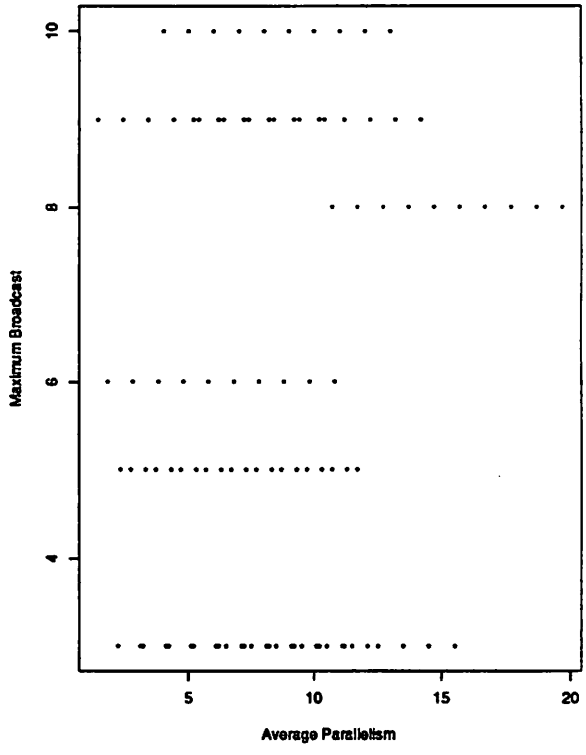


FIGURE 4.7 Diversity of examples: parallelism vs. broadcast

pipelining) are an almost ideal tool for the fast generation of examples with very different structures. On all examples the algorithm achieved consistent results.

The diversity of the examples is illustrated in Figure 7. The x-axis denotes the average amount of parallelism (how many nodes can be executed per control step on the average), the y-axis plots the maximum broadcasting (to how many CDFG nodes the operation result is transferred). Similar plots can be obtained for other relevant parameters, which indicates that the algorithms are able to cover a broad spectrum of examples.

4.6.3 Robust Parameters

The fact that allocation, scheduling and assignment algorithms are usually tested on only a few examples is only part of the assessment problem. It is well know from statistics, that it is

very easy to overtune an algorithm. There are examples where otherwise useless algorithms produce excellent results on a few test examples [Bre83]. To avoid overtuning and lots of "magic" values for "magic" numbers, we tested our algorithm by varying all parameters over large range of values. (Essentially, we changed the various weight factors in objective function and algorithms parameters). We notice very little changes in the quality of the achieved solution. The parameters have a large influence on the run time of the algorithm (up to 100%).

4.6.4 The Effectiveness of Algorithms

The effectiveness of the algorithms can be easily realized with the aid of the following examples. The allocation, assignment and scheduling process for the standard 5th order elliptical filter takes 0.7 sec on a SPARCstation2 (this includes all overhead such as database reading and annotation and providing user feedback). The obtained result is as good as the best published [Sto89]. Since the graph is flat and the solution is identical to the predicted min-bound, the allocation process converges in one step. To get an idea of the performance of the algorithm on larger graphs, a 7th order filter with a flow graph of 113 nodes was allocated and scheduled in 32.5 sec. During the allocation process, the flat graph assignment was called 36 times, while scheduling module itself was called 13 times. After applying two transformations (retiming and associativity), which changed the graph structure, scheduling was performed in 15.8 sec. (1 allocation, 16 assignments, 8 schedules)

Finally, to demonstrate the effectiveness of the hierarchical approach, we have scheduled two complex examples. The first example, a DFT with iterative coefficient generation, consists of a nested loop and would contain 248,642 nodes in a flattened format. The allocation and scheduling process of the hierarchical graph only takes 2.0 sec. As a second hierarchical example, we have scheduled a one-dimensional histogram program, used in Electro-Cardiogram analysis. The problem contains six loops, two of them nested. In flattened format, the graph would

contain 38,867 nodes. The global allocation and scheduling process of the hierarchical graph was performed in 2.2 sec.

4.7 CONCLUSION

An integrated system of allocation, assignment and scheduling algorithms is presented. Both, novel constructive and probabilistic rejectionless anti-voter approaches are introduced. The properties of the algorithms are discussed from both a theoretical and experimental point of view. The quality of the presented algorithms is demonstrated by comparing the results of diverse examples versus the estimated min-bounds on numerous and diverse examples.

5

BEHAVIORAL TRANSFORMATIONS FOR THE SYNTHESIS OF HIGH PERFORMANCE DSP SYSTEMS

5.1 INTRODUCTION

To solve a given DSP computational problem, a large number of algorithms can be used. Often any one of these algorithms can lead to several implementations, each with vastly different execution times, hardware requirements, power constraints, testability and other parameters of interest. The selection of the algorithm best suited for the optimization of those objectives is a crucially important task in the design process of high performance DSP ASICs. An equally important task is to ensure that the potentials of a given algorithm are maximized. This is achieved through the application of optimizing transformations. To maximize effectiveness it is crucial that transformations are globally optimized.

Transformations are changes in control data flow graph structure which improve the final implementation, without altering the input-output relationships. Most of the behavioral transformations are been introduced in the field of software compilers [Aho77, Goo81]. They include

constant arithmetic, common subexpression elimination and value numbering. The most important among them are the loop transformations, such as loop retiming, software pipelining, loop jamming, partial and complete loop unrolling and strength reduction. The latter are especially suitable for real-time systems, in which a program always contains an infinite loop over time and concurrency can be exploited more efficiently. Although majority of those transformations are well known from software compiler literature, an attempt to apply them to high level synthesis poses specific challenges. There are two additional degrees of freedom: hardware parallelism and hardware definition. This chapter describes the application of the following transformations in high level synthesis framework: retiming, pipelining, commutativity and fast implementation of recursive programs.

5.2 RETIMING AND ASSOCIATIVITY

5.2.1 Introduction

5.2.1.1 Motivation

The goal of the high level synthesis process for application specific integrated circuits is to translate the specification of the algorithm (defined in terms of its behavioral semantics as well as its performance requirements) into architectural primitives (being an interconnection of execution units, memory and control units) in such a way that the resulting silicon implementation minimizes a certain function. Most often this function is either the area and/or the power consumption.

Unavoidable high level synthesis tasks comprise of module and clock selection, scheduling, assignment and allocation. However, even when the highest quality algorithms are used for those tasks, the quality of a final result is often constrained by the computational structure of the specified algorithm.

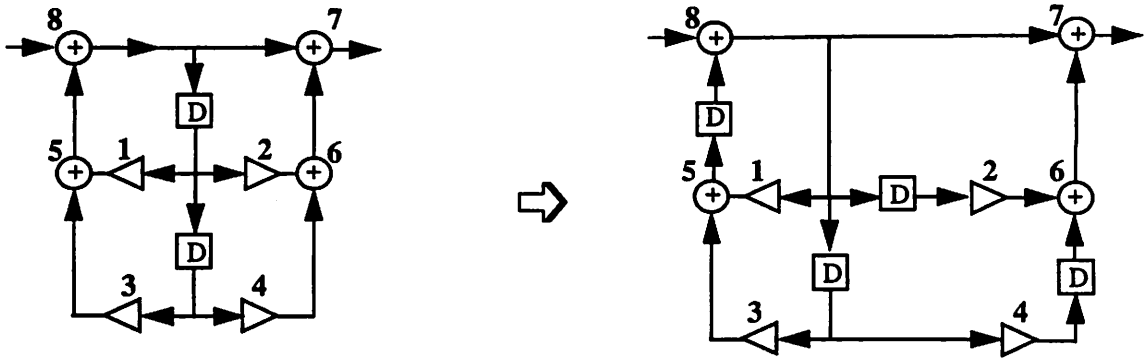


FIGURE 5.1. Biquadratic filter (a) before and (b) after retiming for resource utilization

This is illustrated using the example of Figure 1a. This Figure shows a direct form II biquadratic section often used in the realization of IIR filters. Assume that at most 4 clock cycles are available for the execution of this flow graph and that both multiplication and addition take a single clock cycle. The critical path of this computational graph equals four clock cycles as well. A potential schedule for this filter, requiring a minimal amount of hardware (for the sake of simplicity we will address only execution units here), is shown in Table 1. Regardless of the scheduling technique used, we need at least 2 multipliers, since the graph contains 4 multiplications and no multiplications can be scheduled in the last control cycle. Also, since no addition can be executed in the first control cycle, 2 adders are needed. This can be easily verified using Figure 2, which plots the maximum parallelism available in the graph over time (in terms of the number of additions and multiplications).

The resource utilization is obviously not equally balanced over time. If we define the **resource utilization** as the ratio of the number of cycles a resource is exploited over the total number of available cycles, then the resource utilization for adders and multipliers in this example is 50%. This is an indication of a relatively low quality solution, in this case caused by the inherent structure of the computational graph.

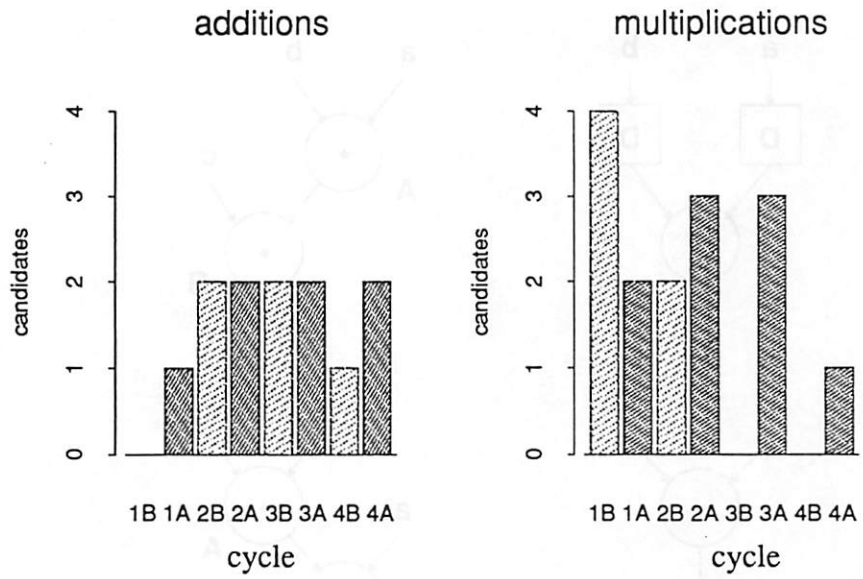


FIGURE 5.2. Available parallelism before (a) and after (b) retiming for resource utilization

	BEFORE		AFTER	
CYCLES	Multipliers	Adders	Multipliers	Adders
1	3 , 4	-	1	8
2	1 , 2	5	3	6
3	-	6, 8	4	7
4	-	7	2	5

Table 1: Possible biquadratic filter schedule before (a) and after (b) retiming

Transformations are the best way to defeat these resource utilization bounds. Two particularly effective transformations to achieve this goal are retiming and associativity, as presented in Figure 3.

❑ Retiming (Figure 3a) uses the distributivity property of the delay operator over most other operators. In other words, when D is defined as the delay operator, the statement D(a) * D(b) is equivalent to D(a * b) and vice versa (with * an arbitrary operator).

Consider now the flow graph, shown in Figure 1b, obtained by moving the delays (retiming) in the original flow graph of Figure 1a. It is easy to check that the resulting graph has the

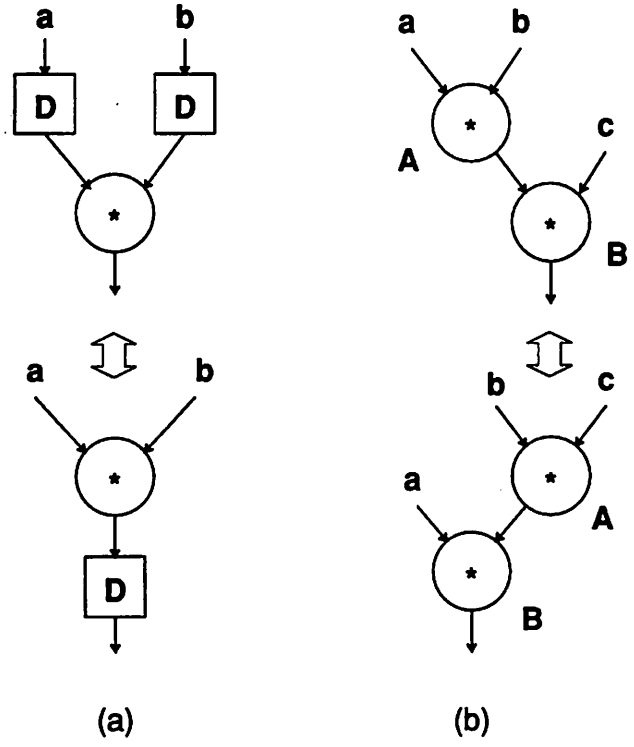


FIGURE 5.3. Basic Retiming (a) and Associativity (b) Moves

same input/output relationship as the graph of Figure 1a. The available parallelism is plotted in Figure 2 of step denoted by A. It is obvious that the resource utilization is far more balanced and a solution with one multiplier and one adder can now be achieved as shown in Table 2. The resource utilization for the execution units now equals 100%.

❑ Associativity (Figure 3b) is a basic property of many algebraic structures, starting from group to vector spaces and matrix algebra [Van50]. Associativity postulates that, in the set over an algebraic structure defined using an operation *, for every a, b, and c, which are elements of the set, holds that a * (b * c) = (a * b) * c.

A simple application of associativity is demonstrated in Figure 4. Implementing the CDFG of Figure 4a in 4 cycles (with both additions and multiplies taking one cycle) would obvi-

ously require 2 multipliers and 1 adder. However, after applying associativity on the adder chain (Figure 4b), the critical path is reduced to 3 cycles and a single multiplier and adder are sufficient for the implementation. This example illustrates the most common use of associativity, namely for the so called *tree height reduction*, although it can contribute in other ways to improving the resource utilization, as will be shown in subsection 5.

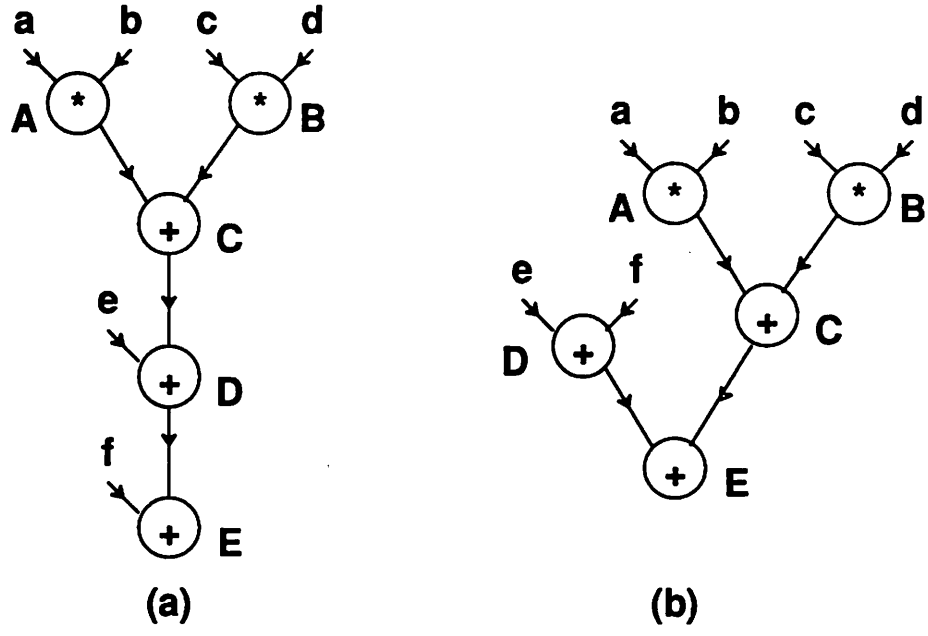


FIGURE 5.4. Applying Associativity to Improve Resource Utilization CDFG before (a) and after (b)

Furthermore, the retiming and associativity transformations are not orthogonal. Their full power can only be exploited when applying them simultaneously. This statement will be discussed in more detail in subsection 5. Some other resource utilization improving transformations (such as distributivity) and their incorporation in the proposed framework will be analyzed there as well.

The above examples illustrate that while transformations are not necessary for a bare minimum high level synthesis system, they are essential when trying to achieve a high quality solution (or sometimes even just a feasible solution).

5.2.1.2 Previous work

While the basic synthesis operations, especially scheduling, [McF90] have been the subject of extensive research efforts, transformations have received significantly less attention. Most synthesis systems apply only the basic software transformations, such as dead code elimination, manifest expression removal and common sub-expression elimination [Tri87, Wal89]. Also pipelining is very often applied [Par88a, Pau89, Hwa91]. Although pipelining is very powerful, it is not a transformation in the strong sense, since it increases algorithm latency. Its application domain is also limited to non-recursive algorithms [Mes88].

Retiming has been successfully applied in several areas of design synthesis and automation. Until recently, it has been exclusively used either to reduce the critical path in a circuit or graph [Lei83], to minimize the number of delays in the graph [Lei83, Goo86] or to optimize sequential networks using combinational logic tools by temporary moving delays to a periphery of a network [Mal91].

Associativity is most often used in conjunction with distributivity for a reduction of the critical path. Valiant [Val83] and Miller [Mil88] presented optimal polynomial time complexity algorithms for critical path minimization. In high level synthesis, it has been used for the reduction of the number of pipeline stages [e.g. Har88].

5.2.1.3 Problem formulation

The resource utilization U_r for a resource r can be defined as the ratio of the number of control steps in which the resource is used over the total number of control steps available. The total resource utilization U_{tot} of a design can then be defined as the weighted sum of the resource utilizations over all hardware primitives.

$$U_{tot} = \sum_{r \in R} w_r U_r \qquad \text{(EQ 1)}$$

with R the total set of hardware resources used. The weights w_r are proportional to the hardware cost of the resource.

The cost of a design is inversely proportional to the resource utilization. Achieving a high resource utilization is in general equivalent to achieving a small design cost. During the design optimization process however, it is easier to measure (or estimate) the resource utilization than the actual hardware cost. In the next chapter, we will discuss how the utilization can be estimated in an efficient and accurate way from the CDFG (combined with information on the hardware library).

The problem discussed in this section can now be defined as a constraint satisfaction problem:

Given: *A control data flow graph and a proposed hardware architecture.*

Goal: *Apply retiming and associativity in a such a way that the resource utilization of the resulting control data flow graph is maximized.*

The solution to the above problem can be used as a subroutine to address both the hardware minimization problem formulation (given the time constraints) as well as the time minimization formulation (given the available hardware). Although we will concentrate on the second formulation (which is the most appropriate one for signal processing applications), we will also discuss some experimental results for the time minimization case.

5.2.1.4 Section organization

The section presents the basic resource utilization optimization framework. The introduction of the two standard transformations, being retiming and associativity, in this framework will be discussed. Unfortunately, applying only retiming for resource utilization on itself turns out to be an NP-complete problem [Pot90]. An iterative probabilistic approach is therefore advocated.

Achieving a high quality solution within such a framework requires the addressing of the following technical difficulties:

(1) deriving a good objective function, i.e. how to estimate the implementation cost for an instance CDFG in a fast and accurate way.

(2) how to efficiently reach the optimal CDFG, given the above objective function.

The objective function is discussed in subsection 2, while the algorithm which achieves the defined goals is described in subsection 3. The high quality of the proposed solution is demonstrated in subsection 4 by statistically analyzing a number of examples. Finally a number of properties of the introduced transformations will be discussed, including how to recognize when those two transformations will lead to the significant improvement and the relationship with other high level synthesis transformations.

5.2.2 Objective Function

As we have already stated, our goal is to apply retiming and associativity to achieve a high resource utilization (measured over all resources, being execution units, memory and interconnect). A good objective function should therefore be highly correlated to the final (unknown) hardware utilization. The objective function also should be easy to compute, since, as shown in [Pot90], it is used in the optimization of an NP-complete problem, and it has to be evaluated many times. A simple yet effective objective function can be constructed, based on the following observations:

(1) It is easier to achieve a high resource utilization when the timing constraints on CDFG nodes are not strict;

(2) It is advantageous to distribute CDFG nodes vying for the same resource (which might be, for example, adder or a particular register file) over the time span;

(3) The critical path should be shorter than the available time;

(4) The number of variables which are alive at the same time, should be smaller than the number of available registers.

Those observations can be quantized in the following way:

Any operation *A* has to be scheduled in the interval between its *As Soon As Possible* ($ASAP_A$) and its *As Late As Possible* ($ALAP_A$) times. Those times can be easily computed, using leveling according to the input and to the output. The slack of a node is defined as the difference between the ALAP and ASAP times, incremented by 1. A CDFG with a lot of operations with a small slack represents a highly constrained scheduling problem, which often results in a poor resource utilization. However, even when the average slack is high, scheduling can still be difficult to achieve if a number of CDFG nodes with a very small slack are present. Therefore, in order to properly scale the expected difficulty and the number of constraints during scheduling we define for each CDFG node property a measure, called the *expected scheduling difficulty* (SD). SD is defined as the inverse of the slack. The total scheduling difficulty of a CDFG, composed of the node-set S, is defined as the sum of the scheduling difficulties over all CDFG nodes:

$$TSD = \sum_{A \in S} \frac{1}{ALAP_A - ASAP_A + 1} \qquad \text{(EQ 2)}$$

At the same time, it is important, that operations which can compete for the same resource (same type of execution unit, same interconnect, or registers in the same register files) are not happening simultaneously. The probability, that two operations *A* and *B* which require the same type of resource will happen simultaneously and therefore during assignment will require another instance of this type of resource, is proportional to the overlap of their ASAP-ALAP intervals (O_{AB}). This probability can therefore be approximated using the following formula:

$$OL_{AB} = \frac{O_{AB}}{SL_A \times SL_B \times IR_{AB}} \qquad \text{(EQ 3)}$$

where SL_A is the slack of operation A, SL_B is the slack of operation B, and IR_{AB} is number of resources of this class. A total overlap measure, called TOL, is defined over all nodes of the graph:

$$TOL = \sum_{A \in S, B \in S, A \neq B} OL_{AB} \qquad \text{(EQ 4)}$$

where S is the set of all nodes in the CDFG.

The retiming and associativity transformations may also change the critical path of the graph. Obviously, no feasible schedule exists if the critical path is longer than the available time. Furthermore, retiming changes the number of delays in the graph. All variables which are associated with delays must be stored simultaneously (during the first cycle) in registers. No feasible schedule is available if the number of delays exceeds the max bound on the number of registers. Both the critical path and the number of delays, are incorporated in the objective function, such that it becomes infinity when one of those constraints is violated. Our experience furthermore indicates that a correlation exists between the number of delays and the number of registers required. Therefore, it is useful to add the number of delays (ND) to the objective function as a measure of the total register cost.

All those factors can now be combined into the global objective function:

$$\text{Objective} = \begin{cases} \infty, & \text{if } t_{crit} > availabe\ time \quad \textbf{or} \quad ND > number\ of\ registers \\ \alpha_1 \times TSD + \alpha_2 \times TOL + \alpha_3 \times ND, & \text{otherwise} \end{cases} \qquad \text{(EQ 5)}$$

Weight factors can be explicitly set by the user. For example, a relatively large α_3 often results in fewer register, but more interconnect and execution units. For all examples discussed in the experimental section, we used the following default settings:$\alpha_1 = 0.8$, $\alpha_2 = 0.1$ and $\alpha_3 = 0.1$.

The close correlation between our objective function and the quality of the obtained solution is depicted in Figure 5 for the example of an 11th order FIR filter. The x-axis shows the value of the objective function, while the y-axis respectively contains the corresponding number of control steps, necessary to execute the graph on a fixed amount of hardware (as determined by the scheduling process) (Fig. 5a) or the amount of hardware needed to execute the graph in a fixed amount of time (here 13 cycles) (Fig. 5b). It is easy to observe that a small value of the objective function invariantly predicts a high quality solution.

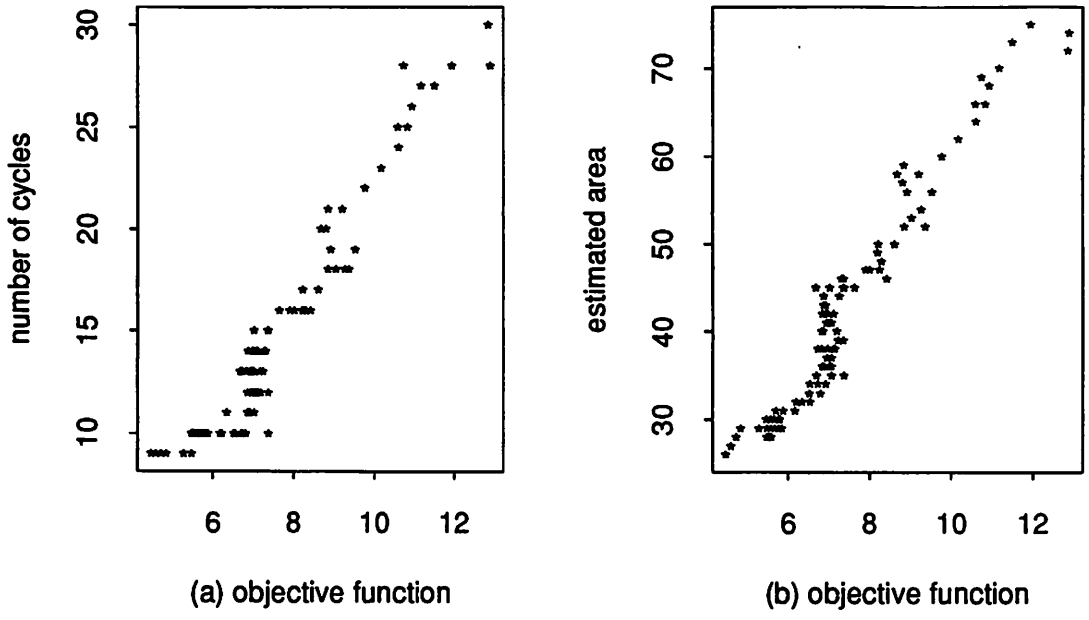


FIGURE 5.5. Correlation between Objective Function and Solution Quality

5.2.3 Learning While Searching Algorithm

While the traditional retiming problems (for critical path and minimum number of delays) have a polynomial complexity, we have proven that the retiming for resource utilization algorithm is an NP-complete problem [Pot90]. It is therefore very unlikely that the worst case polynomial complexity algorithm exists.

In order to efficiently solve the problem, a new probabilistic iterative improvement algorithm has been developed. This chapter describes first of all the set of basic moves applied during the iterative improvement, followed by a discussion of the proposed optimization strategy. Experimental results are presented and analyzed in the next subsection.

5.2.3.1 Basic moves

Two classes of basic moves can be defined, being retiming and associativity moves. The retiming move was discussed in the first chapter and is shown in Figure 3a.

initial	transformed
a+(b+c)	(a+b)+c
a+(b-c)	(a+b)-c
a-(b+c)	(a-b)-c
a-(b-c)	(a-b)+c
a*(b*c)	(a*b)*c
a*(b/c)	(a*b)/c
a/(b*c)	(a/b)/c
a/(b/c)	(a/b)*c

Table 2: Basic Associativity Moves

In order to enhance the power and the application range of associativity, we have expanded its definition, such that it addresses several additional cases, not covered by the standard definition. The generic associativity move was presented in Figure 3b and corresponds to entries 1 and 5 in Table 2. Note that the application of associativity is disallowed when node A has some fanout besides node B. Those basic moves have been extended with 6 additional transformations, as shown in Table 2. Cases 3,4,7 and 8 are especially interesting, since they allow the trade off between respectively the number of additions and subtractions and the number of multiplications and divisions in the CDFG. Notice also that for each move, defined in Table 2, an

inverse transformation can be defined. We call the inverse transformations the *reverse moves*, in contrast to the moves of Table 2, which are called the *forward moves*.

5.2.3.2 Learning while searching algorithm

When applying the above transformations on a typical example, a large number of moves is possible at every point in time. As even more basic moves might be introduced in the future, time-efficient algorithms are definitely necessary. We first tried the popular simulated annealing [Kir83] technique. Although the results were satisfactory, the run times were excessive for large examples, even when adaptive cooling [Cat88] and a rejectionless approach [Gre86, Pot90] were applied. Therefore, we decided to use a new and faster probabilistic approach.

The presented technique is organized as a two phase process. In the first phase, the solution space is scanned in an organized fashion to detect areas where the objective function has a small value. Those areas are then used in the second phase as the starting points for a more elaborate search towards a final solution.

The goal of the first phase is thus to discover (learn) **k** solutions where the objective function has a small value. In order to achieve this goal, we will traverse the search space a number of times, each time favoring one particular direction of traversal. For instance, we will first (probabilistically) favor moves in the forward direction (moving the delays from the inputs to the outputs for the retiming and favoring the forward associativity moves). After 1/4 of the optimization process, the preferred direction is reversed: moving the delays from output to input as well as the reverse associativity moves are now probabilistically favored. Finally, for the last 1/4 of the time, the forward moves are favored anew.

At every point in the optimization process, we select a move in a probabilistic fashion, proportional to the improvement in the objective function, while also accounting for the favored direction at that time. No selected moves are ever rejected. Moves, which increase the objective

function, can be selected, but the chances for this to happen are inversely proportional to that increase. To increase the speed of the design space search, we decided to evaluate the objective function only every four steps, resulting in a speed-up of approximately four times. This is acceptable, since neighboring solutions in the design space (solutions which can be reached in at most **m** moves) tend to display a strong correlation in their objective function values. Since the exact location of a local minimum is only determined in the second phase, no degradation of the solution quality was observed as a result of this simplification. Table 3 shows the minimum, average, and maximum correlation among neighboring solutions for several examples.

m	1	2	3	4	5	6
min	0.971	0.932	0.919	0.902	0.866	0.808
average	0.978	0.952	0.927	0.909	0.878	0.841
max	0.994	0.978	0.959	0.932	0.907	0.879

Table 3: Correlation among solutions on distance of *m* moves

The first phase results in **k** starting points for the second phase. Those are used as the seeds for a greedy search towards the final solution. The objective function is now observed at each step and for all possible moves. The move offering the best decrease in the objective function is automatically selected. For each starting point, the search is concluded when a local minimum is reached. The best of those minima is selected as the final solution.

We have set the number of starting points **k** to 10 for the examples discussed in the next section. The length of the first phase was set such that the number of moves during the first forward traversal equaled 10 times the number of nodes in the graph. Moves in the forward direction were preferred with a ratio 4:3. We have varied these values of large ranges and did not notice any significant changes in the quality of the solution, although the effects on the run times were outspoken. The presented approach can easily be augmented with a cooling mechanism

(phase 1) or backtracking. Experiments have shown however, that those extensions do not produce any significant improvements and have a detrimental effect on the run time.

It is interesting to notice that the presented approach resembles, to some extent, the simulated annealing approach [Kir83] as well as the Kernighan-Lin iterative improvement approach [Ker70]. When only the second phase is applied, the approach is equivalent to Kernighan-Lin (which was almost uniquely used for partitioning problems until now). While phase 1 uses an iterative, probabilistic improvement technique, just as simulated annealing, some major differences with the annealing approach can be observed: First of all, the presented technique uses a directed search, while annealing executes random moves. The major difference however is the combination of probabilistic exploration and greedy solution generation.

5.2.4 Experimental Results

The ultimate proof of the usefulness and effectiveness of a proposed transformation or optimization algorithm is to apply it on real life examples. We have applied our technique on 40 CDFGs, which include common DSP, communication and error-correcting code examples (such as FIR, IIR, adaptive and wave digital filters and simultaneous polynomial division and multiplication). For each of those examples, we varied the available time. We also varied the relative execution lengths of the operators (such as shifts, adds, multiplications and multiplexers).

The primary objective was to measure how much hardware can be saved if the transformation is applied. A secondary task was to evaluate the potential of the transformation to minimize the execution time of an algorithm without increasing the latency (in contrast to pipelining). While improvements in execution units and memory cost can be measured exactly, the cost of interconnect could only be estimated, since precise numbers are only available after time consuming tasks such as floorplanning and routing. We have compared interconnect cost based on the number of busses and assuming that all busses have the same cost.

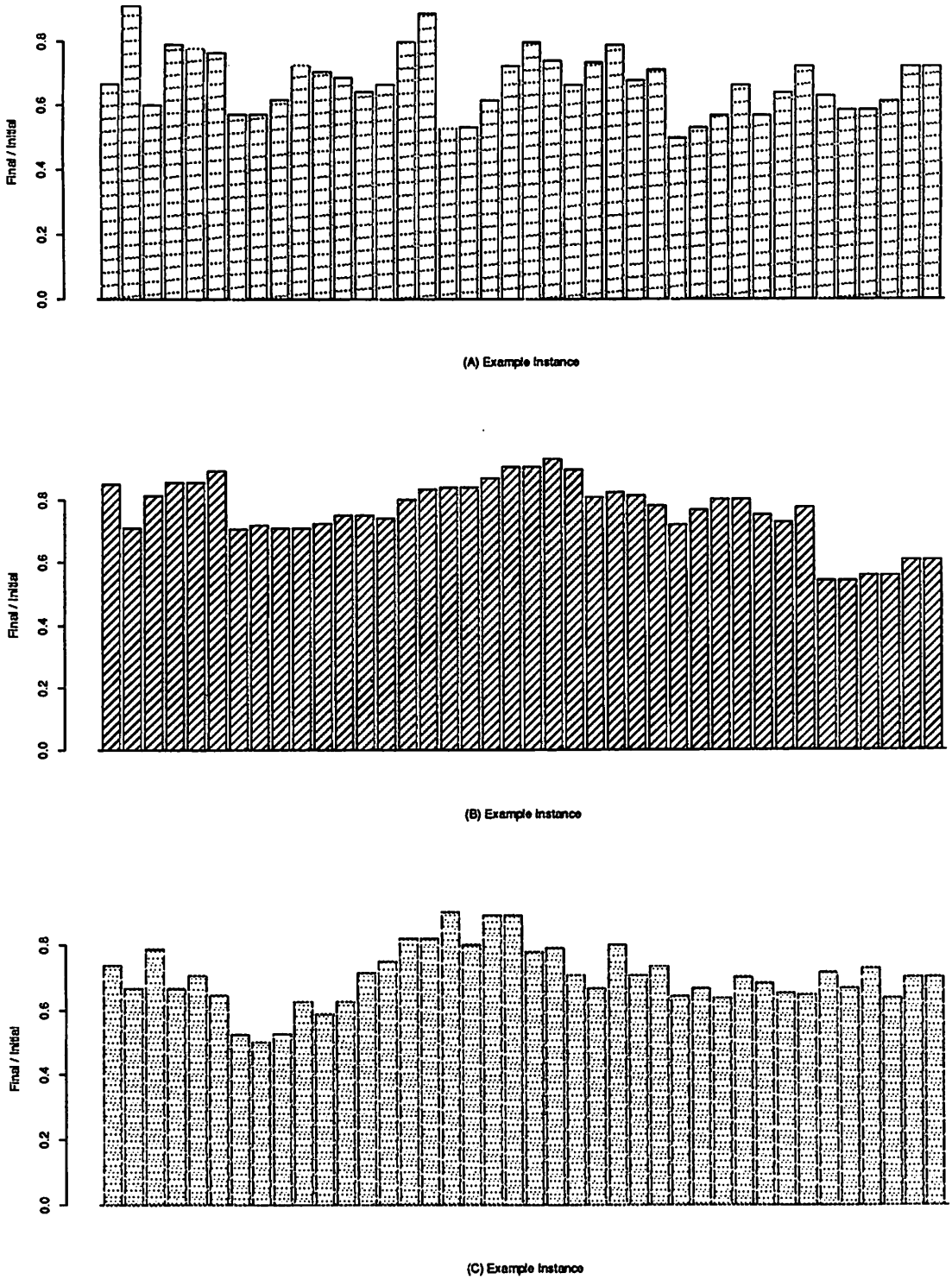


FIGURE 5.6. Ratio of Implementation Cost of Final versus Initial Implementation for Benchmark Example Set (Execution Units (a), Memory (b) and Interconnect (c))

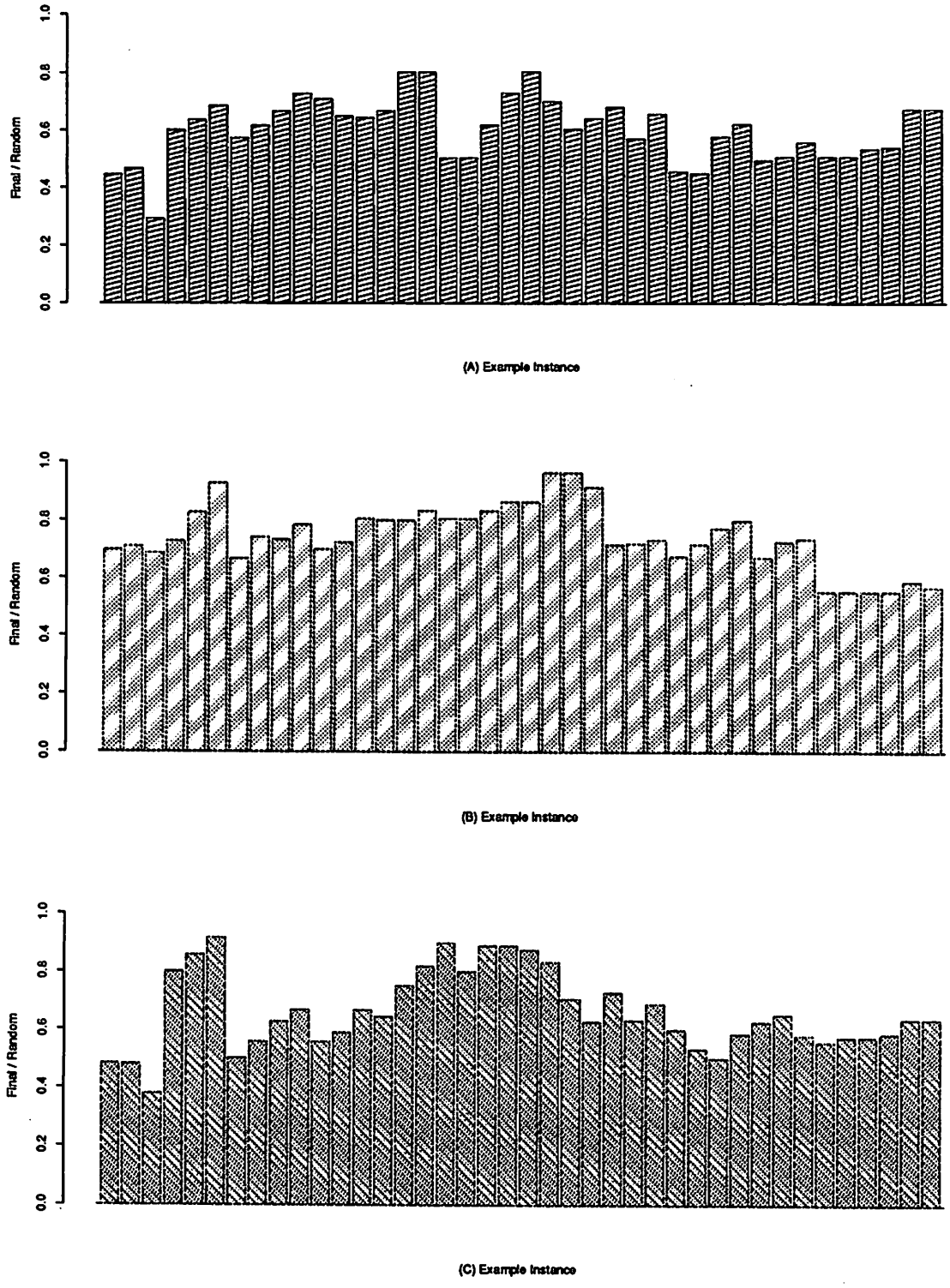


FIGURE 5.7. Ratio of Implementation Cost of Final versus Random Implementation for Benchmark Example Set (Execution Units (a), Memory (b) and Interconnect (c))

The ratio of the implementation cost after the application of the transformation over the cost of the initial implementation for all benchmark examples is plotted in Figures 6a, 6b and 6c (for execution units, memory and interconnect costs respectively).

Comparing the final solution with the initial CDFG, provided by the designer, has only limited significance since this highly depends on the amount of manual optimization, applied by the designer. We therefore compared the cost of the generated solutions against the cost of the median solution among the 20 random solutions (generated by randomly applying retiming and associativity moves) as well. Figures 7a-c show the ratios when comparing against the randomly generated solutions. The average and median savings against the initial implementation and the median random implementation are tabulated in Table 4.

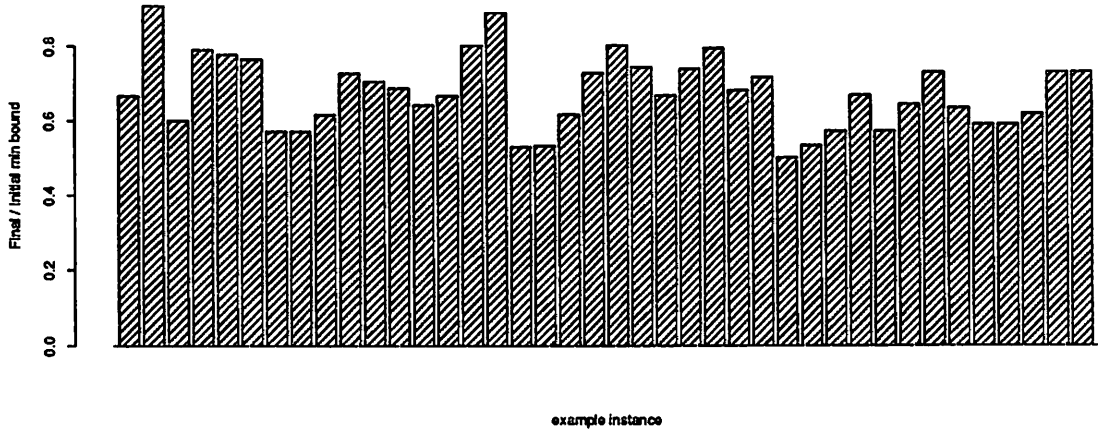


FIGURE 5.8. Ratio of Implementation Cost of Final versus Min Bounds of Initial Solutions for Benchmark Example Set (Execution Units)

The generation of the above results required the use of a particular set of scheduling, assignment and allocation tools. Since those problems are NP-complete as well, it might be argued that the obtained improvements were not the result of the transformation by itself, but are due to the fact that the heuristic scheduler performed better on the transformed graphs. This argument can be discarded using the following simple procedure. For each instance of a CDFG,

it is possible to establish sharp minimum bounds on the resources, by using the facts that during some control steps no candidates are available for scheduling [Rab90].

	EXU		MEMORY		INTERCONNECT	
	Random	Initial	Random	Initial	Random	Initial
average	40.1	32.5	25.4	23.6	33.8	29.7
median	28.5	33.3	26.6	22.2	36.6	30.0

Table 4: Savings against the Initial and Random Implementations (in%)

These bounds can not be outperformed, regardless of the used scheduling. If the application of the transformations results in a decrease of those bounds, then this improvement is a pure consequence of the transformations. The ratios of the final implementation cost versus the estimated min-bounds of the initial solutions for execution units are plotted in Figure 8. The values of the median, average and maximum improvement for execution units area respectively equal 21.6%, 21.7% and 47.1%. Only once was the min bound not reached.

CS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A1	0	1	4			6	7	19	22	9	25	10	12	17	18	13
A2	3	2				21	23	26	14	25	27	31		29	30	33
M1				5	5			8	8			16	16	11	11	
M2				20	20			24	24			28	28	32	32	

Table 5: Schedule for 5th Order Elliptical Filter, using 2 adders (A1 and A2) and 2 multipliers (M1 and M2) in 16 control steps

To evaluate the effects of the transformation on a well known benchmark, we have applied the technique on the popular 5th order elliptical wave digital filter example. The results are tabulated in Table 6 for the available times ranging from 15 to 19 clock cycles (in correspondence with the standard benchmark, we assume that a multiplication and an addition take respectively 2 and 1clock cycle). As can be observed from the table, the average hardware savings due to the

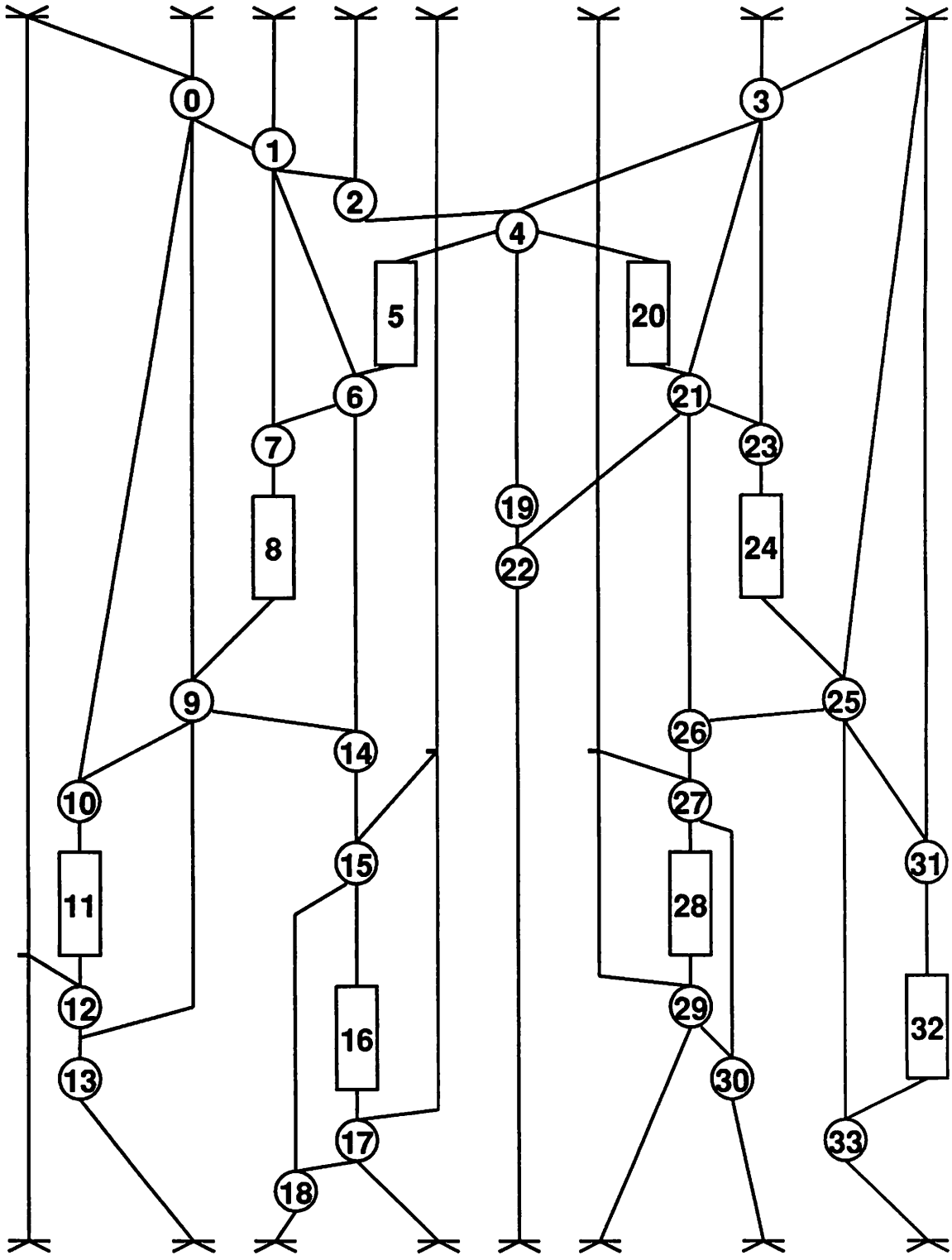


FIGURE 5.9. Flow Graphs and Schedules of Fifth Order Elliptical Filter Before Transformations (for an available time of 16 cycles)

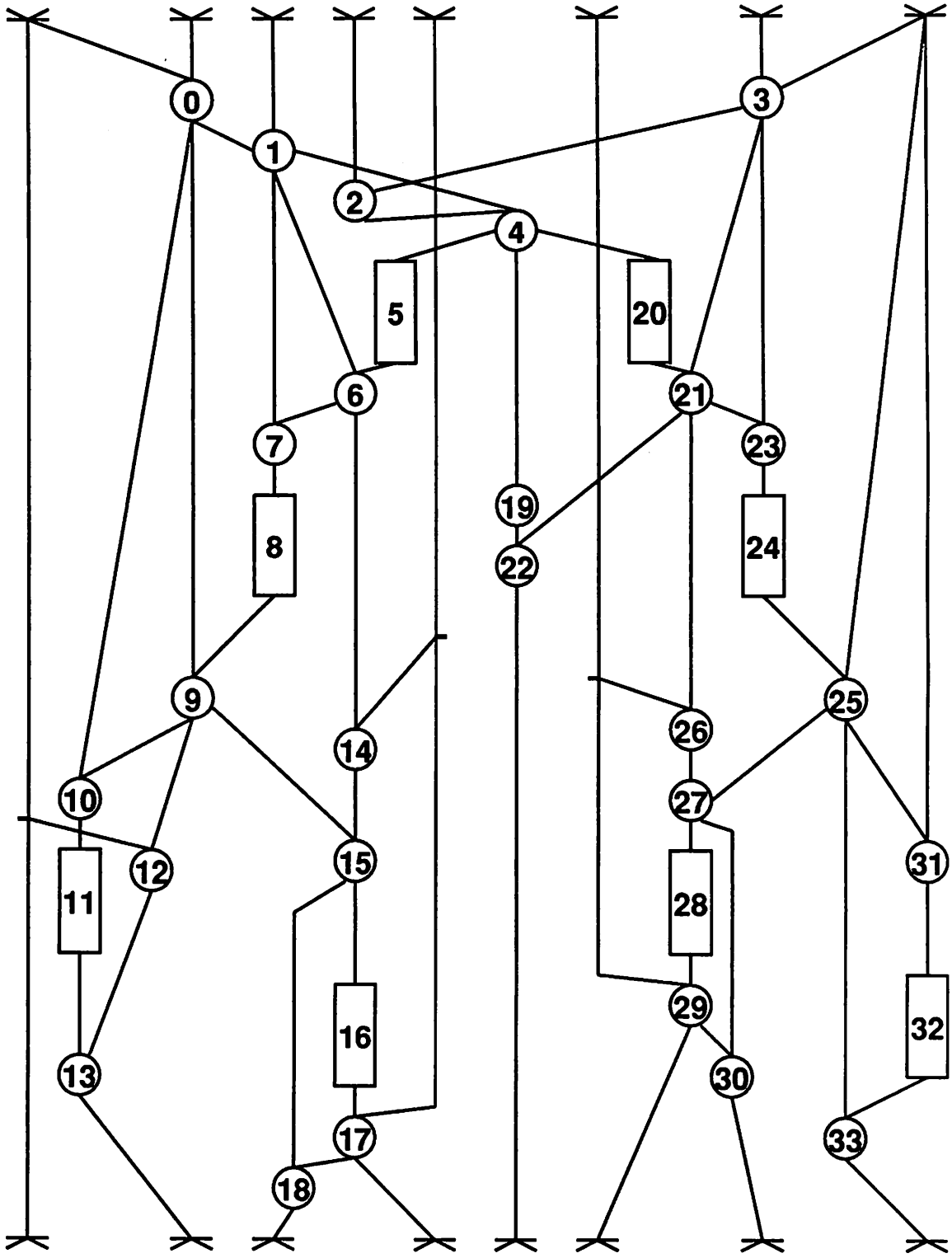


FIGURE 5.10. Flow Graphs and Schedules of Fifth Order Elliptical Filter After Transformations (for an available time of 16 cycles)

transformations are impressive. Finally, to demonstrate how the transformation succeeds in optimizing the resource utilization, we have plotted in Figure 9 and Figure 10 the flow graphs and presented the schedule for the filter in Table 5, assuming that the available time equals 16 cycles. Also interesting to notice is that the fastest solution after retiming for critical path still needs 16 clock cycles [Har89]. The combination of retiming and associativity succeeds in producing a solution which requires only 15 cycles.

	Number of Control Steps	15	16	17	18	19
BEFORE	# of adders / # of multipliers	NA	NA	3/3	3/2	2/2
AFTER	# of adders / # of multipliers	3/3	2/2	2/2	2/2	2/1

Table 6: Number of adders and multipliers used for Implementation of 5th Order Elliptical Filter before and after applying retiming and associativity

The effectiveness of the algorithm is illustrated by the fact that, even for graphs with several hundred nodes, the run time was shorter than one minute. Table 7 shows the run times for several examples (measured in seconds).

Number of Nodes	32	34	40	113
Run Time [s]	5	7	11	34

Table 7: Run times for 4 examples (SPARCstation2, 48MB)

In some cases retiming and associativity for resource utilization result in the spectacular improvements. For example in the case of the second order Volterra filter [Mat91] using the proposed transformation we succeed to reduce three times the area of the implementation. Figures 11 and 12 show the layouts before and after the application of transformations.While the initial implementation required 4 multipliers, 6 barrel shifter, 2 adders and 2 subtractors, after the transformation 1 multiplier, 1 barrel shifter, 1 adder and 1 subtractor were sufficient.The number of used registers were reduced from 38 to 28. It is interesting to notice that while in the first case the critical path was 16 control steps in the second case we actually used only 15 control steps.

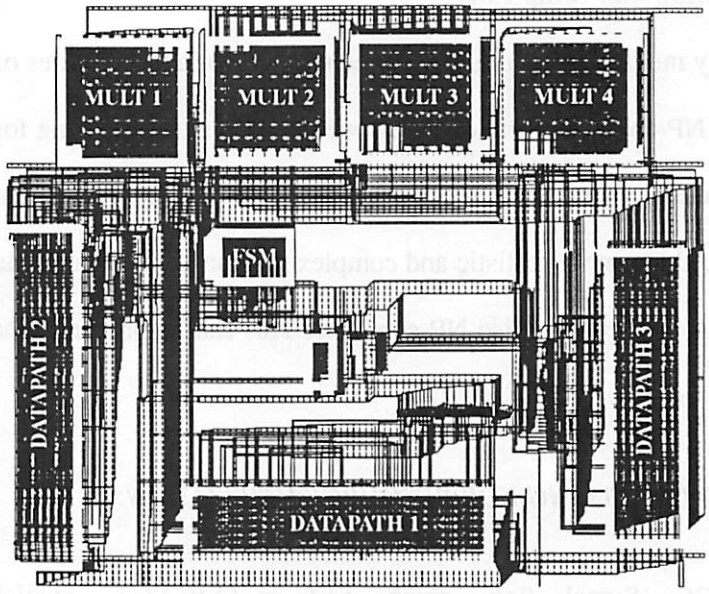


FIGURE 5.11. Volterra filter before retiming and associativity for resource utilization

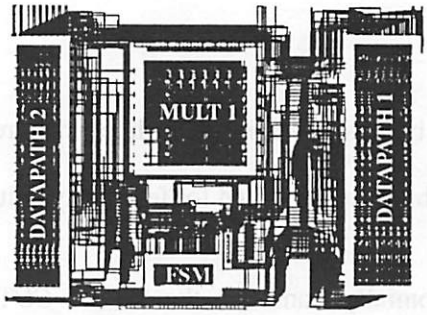


FIGURE 5.12. Volterra filter after retiming and associativity for resource utilization

5.2.5 Transformation Properties

This section will discuss some of the properties of the high level application of retiming and synthesis, including the computational complexity, the range of application and the assessment of the potential. Also, the motivation behind combining retiming and associativity into one transformation is discussed.

5.2.5.1 Computational complexity

As already mentioned before, one of the most important properties of the proposed transformation is an NP-complete problem. Now we will prove that retiming for resource utilization is an NP-complete problem. More precisely, we will prove that the following simplified version is NP-complete. Other more realistic and complex versions, which take into account the cost of interconnect and registers, are also NP-complete. This can be proven without any difficulty by restricting them to this version [Gar79].

PROBLEM: *RETIMING FOR RESOURCE UTILIZATION*:

INSTANCE: Signal flow graph $SFG = (SV, SG)$, available time T, cost $c(v) \in Z^+$ for each $v \in HV$, positive integer L.

QUESTION: Is there a retiming for resource utilization of *SFG* such that its implementation is at most L?

The problem is in NP because a feasible schedule which requires implementation with the cost L, if exists, can be exhibited and checked for feasibility quite easily.

We shall now polynomially transform the *MAX* 3-*CUT* problem to *RETIMING FOR RESOURCE UTILIZATION* to finish NP-completeness proof.

PROBLEM: *MAX* 3-*CUT* [Gar79]:

INSTANCE: Graph $G = (V, E)$, weight $w(e) \in Z^+$ for each $e \in E$, positive integer K.

QUESTION: Is there a partition of V into disjoint sets V_1, V_2, V_3 such that the sum of weights of the edges from E that have its endpoints in different sets is at least K?

Suppose we are given an arbitrary graph G. We will first polynomially transform G to a corresponding *SVG* so that finding a solution in polynomial time to the retiming for resource utilization problem implies a polynomial time solution to the *MAX* 3-*CUT* problem.

For each node V in G, *SFG* contains a disjoint subgraph, which contains as many cycles as the node V has incident edges. An illustration is shown in Figure 14. For example, subgraph B in *SFG* contains 2 cycles, corresponding to the 2 incident edges (1 and 2) to node B in G. Cycle C, corresponding to edge C in G, contains three nodes of different types C', C'', and x. Node x is part of every cycle in the subgraph and acts as enforcer [Gar79]. Each operation takes one control cycle, and the available time is 3 control cycles. Operations $c(C')$ and $c(C'')$ has cost equal to the weight $w(C)$ in G.

The enforcer node x has cost 0. It enforces all delays in a subgraph to be at the same level. Otherwise, no feasible schedule exists.

The problem has been constructed such that in order for two operations of the same type to share the same resources, the delays in their subgraphs have to be positioned at different levels. For instance, the delays in subgraphs A and B have been placed at different levels. Therefore, nodes $1'$ from both subgraphs can be scheduled on the same hardware in cycles 1 (subgraph B) and 3 (subgraph A). Putting delays on different levels corresponds to putting nodes, incident to corresponding edge in G, in different disjoint sets.

A retiming with no hardware sharing obviously has the cost $2M$, where M is the sum of all edge weights in G. This corresponds to a 0 cut for the *MAX* 3-*CUT* problem since all nodes are in the same set. Moving one delay (e.g. on the edge with nodes 5' and 5'' in the subgraph D) will reduce the cost of the implementation by $w(5)$ and increase cutset value by the same amount. Therefore, if we can achieve the retiming for resource utilization solution with the cost $L = 2M - K$,

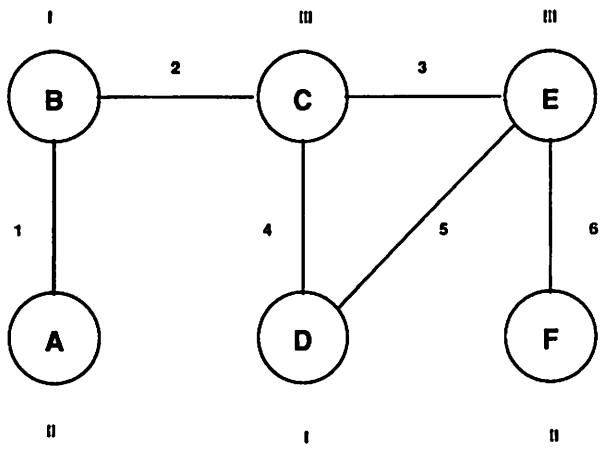


FIGURE 5.13. ***MAX 3-CUT*** **problem**

this corresponds to the solution of the *MAX* 3-*CUT* problem with cost K. Consequently, we have demonstrated that *MAX* 3-*CUT* polynomially transforms to retiming for resource utilization.

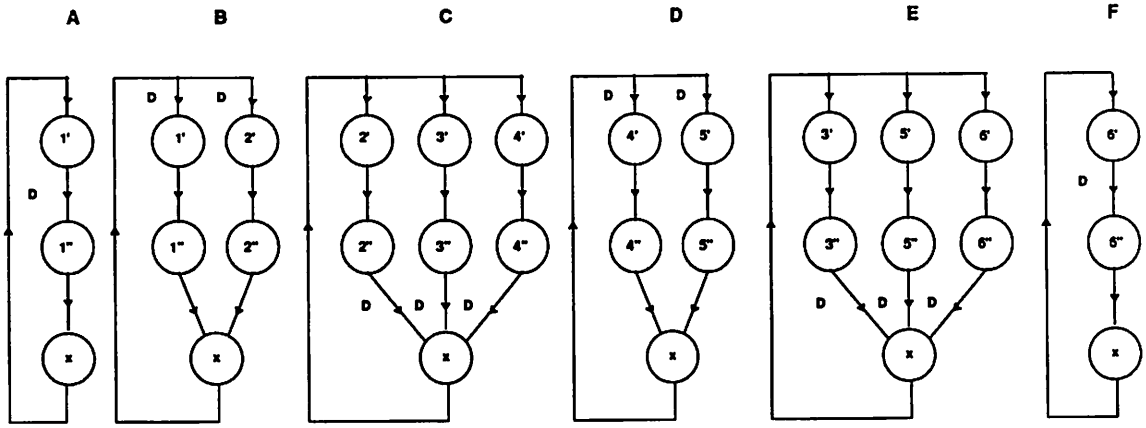


FIGURE 5.14. ***RETIMING FOR RESOURCE UTILIZATION***

5.2.5.2 Application range

Next, one might wonder about the application range of the transformation. The majority of the examples presented in the previous section are filters. Filters are for sure the ideal target for the proposed technique, since they normally have real time constraints imposed on them, since they operate on infinite streams and since they combine information from subsequent samples.

As a net result, delays are naturally present in all filter structures and hence, retiming can be very effective. It should be mentioned that filters are still the most important signal processing components, even though more complex, non-linear or multi-dimensional operations are becoming increasingly important [Bla85].

However, it is important to stress that the application of retiming and associativity is by no means restricted to filters. While this is obvious for the application of associativity, it is important to note that the richest source of delays in a program is the loop construct. Whenever a loop body uses information from a previous iteration, a delay is introduced. In fact, the stream oriented nature of signal processing applications is nothing else than an infinite loop over time. Therefore, retiming is applicable to almost all problem instances which employ iteration or recursion. The application of retiming in those cases can be called *software retiming* in correspondence with the well known *software pipelining* transformation [Lam88, Lam89]. The proposed transformation is therefore also effective in a multitude of other signal processing applications, which rely on the extensive execution of a tight inner loop. Examples of those can be found in the areas of multi-dimensional signal processing, sonar, speech recognition (Markov Modeling), telecommunications (Viterbi search), various signal transformations such as the DFT and DCT and digital audio (error correction, adaptive interpolation). A simple example of the usage of *loop retiming* is shown in Figure 15. Assuming that 2 clock cycles are available per iteration and that both a multiply and an add take 1 cycle, it is easily observed that the first instance requires 2 multipliers and 2 adders, while the retiming graph only needs one unit of both.

Two other important questions (both for a user of the design system as well as for the development of an automatic transformations) can be formulated:

(1) When can this transformation be applied?

(2) What is the potential improvement which can be achieved by applying the transformation?

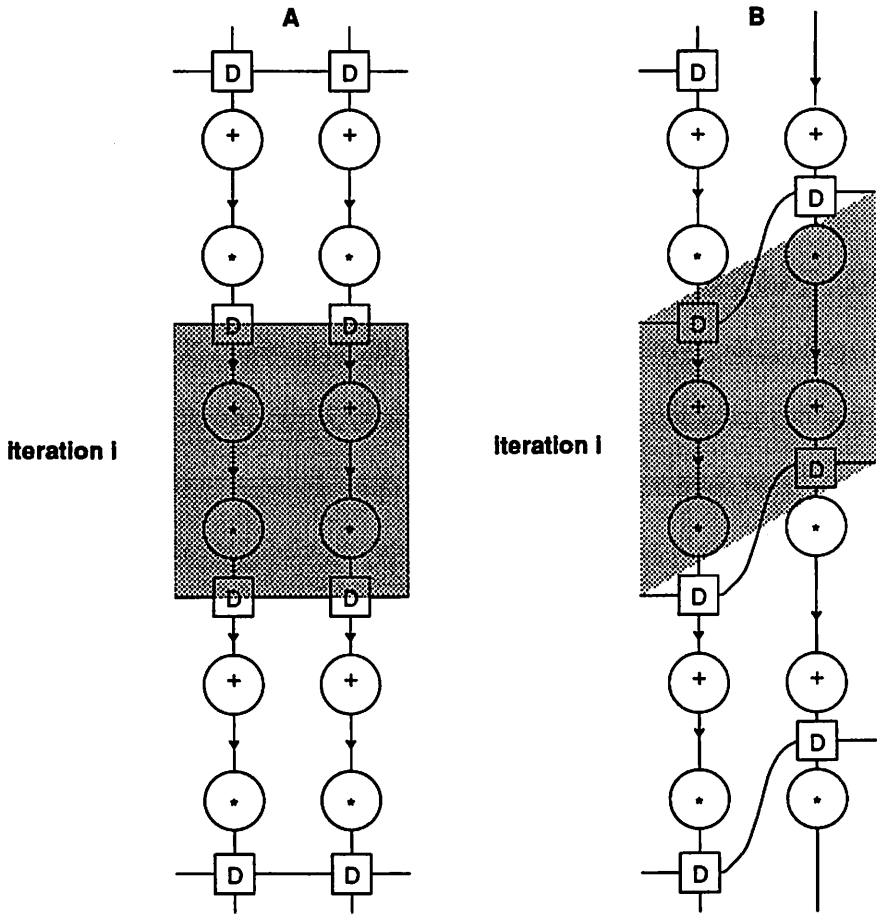


FIGURE 5.15. Software Retiming: before (A) and after (B)

Answering the former question can be extremely difficult for the general case. For instance, for a loop transformation such as loop jamming, it is even unanswerable [Ban88]. However, in the case of retiming and associativity for resource optimization, the answer is pretty straightforward: there exists no restriction on the application. The most promising targets are CDFGs with a lot of delays and structures with chained associative operators.

This brings us to the second question. In the previous section, we have already discussed that the transformation succeeds in lowering the minimal bounds on the resources, hence making it possible to obtain cheaper solutions. The question raised here is what the maximum improvement is which can be obtained by applying the transformation. This can be predicted by compar-

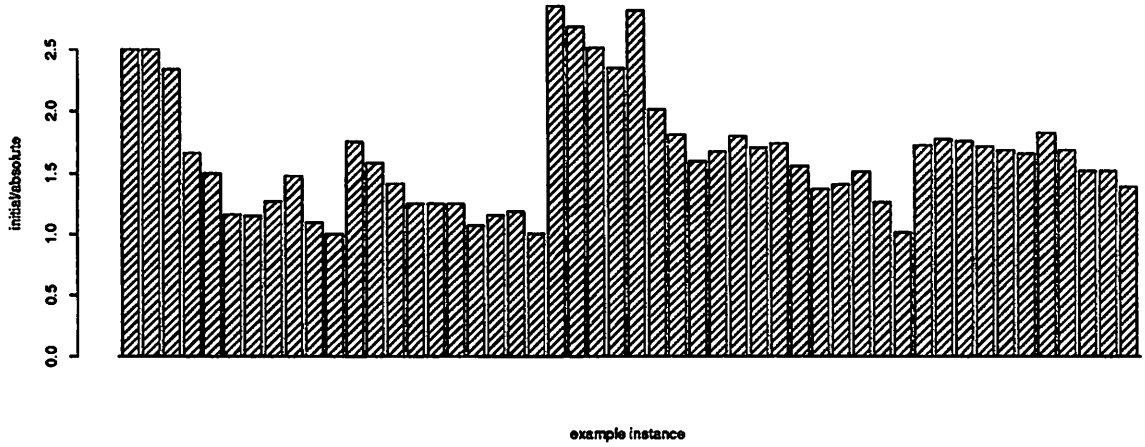


FIGURE 5.16. Ratio of Initial Min Bounds versus Absolute Min Bounds for Execution Units (48 Examples)

ing the estimated minimum bounds of the initial CDFG with the *absolute min-bounds*. These bounds can be predicted using the following formula:

$$N_R^{abs} = \frac{o_R \times d_R}{t_{available}} \qquad \text{(EQ 6)}$$

with O_R the number of operations using resource R, d_R the duration of operator R (in clock-cycles) and $t_{available}$ the available time. The absolute min-bound N_R^{abs} estimates the number of instances needed of the resource R under the assumption that a 100% utilization is achieved. Obviously, this is the best result which can be obtained with the presented transformational approach. (One word of caution here: some of the associativity moves interchange additions for subtractions and multiplications for division. This effectively changes the absolute min-bounds). The potential improvements can hence be measured by comparing the absolute bounds with the estimated min-bounds of the initial CDFG. The results for some of the benchmarks are displayed in Figure 16.

It should be mentioned that even when this comparison predicts no improvement at all, it still might be useful to attempt the transformation: as discussed above, the transformation

relaxes the constraints in the graph and makes it easier to produce a feasible schedule using the minimal hardware.

5.2.5.3 Relationship with other transformations

An obvious question which can be posed is how this transformation relates to the traditional *retiming for critical path* problem [Lei83]. Since a shorter critical path most often means a less constrained graph, a general correlation between the length of the critical path and the cost of the implementation can be observed. The critical path however is only one of the multiple ways of reducing the stress in the graph. For the majority of the examples, discussed in section 5.2.4, the best solution was NOT the one with the smallest critical path.

Pipelining is a transformation, which is closely related to retiming. Retiming has however some distinctive advantages over pipelining: first of all, it does not change the latency of the algorithm. Secondly, retiming can be used in cases where pipelining is totally ineffective, namely to improve on recursive structures (such as IIR filters and recursive loops). Finally, it should be noted that our presented approach can be extended to cover *pipelining for resource utilization* as well.

Two other transformations, closely related to associativity, might also be employed to affect the resource utilization. The *distributivity* move could easily be incorporated in the presented framework. However, due to the fact that distributivity changes the number of nodes in the CDFG, some reformulation of the objective function might be necessary. The *commutativity* transformation will affect only the interconnect part and memory part of the cost function. The effect of commutativity can only be evaluated in conjunction with precise information on the assignment, and is therefore more appropriately combined with the assignment process.

Although the program performance indicates that even more complex transformations can be handled simultaneously in the same framework, it is our conviction that a better approach is

to treat them separately. First of all, all transformations which operate on higher levels of hierarchy (such as loop unrolling and jamming) assume a mechanism for handling hierarchy. Operations on the sub-operator level (bit-level retiming, transformations from multiplications to add/shifts) need different objective functions.

5.2.5.4 Associativity - retiming relationship

On the other hand, it might be asked why we particularly chose to combine associativity and retiming in the same framework. As already mentioned in the introduction, retiming and associativity are not orthogonal. If we want to exploit the full power of those transformations, a simultaneous application is actually mandatory. For instance, the associativity transformation cannot be applied on the following expression:

$$F = D(a + b) + c$$

Moving the delay first using retiming and applying associativity next, results in the following equation, which might produce better resource utilization.

$$F = D(a) + (D(b) + c)$$

A more dramatic example is shown in Figure 17. Figure 17a shows the direct implementation of an *n*-th order FIR filter. The direct form implementation displays a long critical path. Retiming alone does not help to solve this problem: note that neither delay d_{n-1} nor d_n can be moved across additions. However, after applying associativity (n-1) times on the accumulation chain, we can reverse the data flow direction in the chain:

$$y = (((y_1 + y_2) + \ldots + y_{n-1}) + y_n) = (y_1 + (y_2 + \ldots + (y_{n-1} + y_n)))$$

In this way, we obtain the structure of Figure 17b, where delays can be freely moved and recombined (as is shown for d_{n-1} and d_n). The retiming operation can be repeated until all delay operators are moved to the accumulation path (resulting in the reciprocal FIR filter structure).

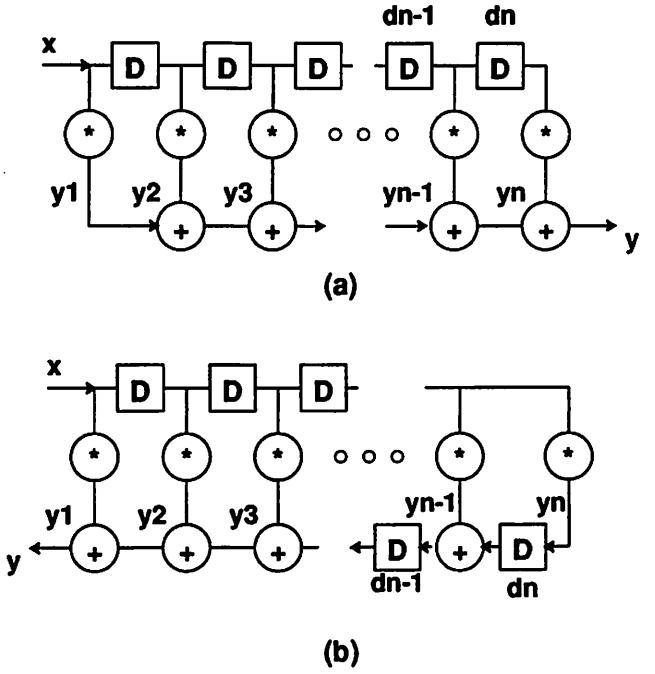


FIGURE 5.17. Combining associativity and retiming (FIR filter)

Although the most visible change in the computational structure is a reduction in the critical path, it is easy to see that the retiming enabled by using associativity will provide a better resource utilization too.

5.2.6 Conclusion

A transformational approach, aimed at improving the resource utilization in high level synthesis, has been introduced. The current implementation combines retiming and associativity in a single framework. This combination of transformations results in considerable area improvements as is amply demonstrated by the benchmark examples. A novel *learning while searching* iterative improvement probabilistic algorithm has been developed and is used to resolve the associated NP-complete combinatorial optimization problem. The proposed algorithm has proven to be very effective both in reaching the optimal solution as well as in run-time.

5.3 PIPELINING

5.3.1 Introduction

Pipelining is probably the most often used transformation in ASIC design, as well as the most discussed one in the high level synthesis literature. An excellent, engineering oriented reference is the book by Kogge [Kog81]. He discusses many important technical questions in detail. Another excellent reference is Chapter 6 of Hennessy's and Patterson's book "Computer architectures". Both books contain an extensive discussion in the field of pipelining research in the context of general computing and long lists of additional references. Therefore, we will limit our overview of previous work to the high level synthesis treatment of pipelining.

Two types of pipelining are most often discussed in high level synthesis, being structural and functional pipelining. Structural pipelining refers to the use of pipelined operational elements (e.g. multipliers). Functional pipelining refers to the partitioning of the control data flow graph into subgraphs that will be performed concurrently. Successive subgraphs, called pipeline stages, are streamed into the pipe so that different CDFG instances are executed simultaneously on the same hardware.

Structural pipelining is the most often discussed feature in scheduling algorithms. While Moritz and Chen [Che91] discuss the simultaneous application of both pipelining techniques, the majority of the work addresses functional pipelining. The major reason why functional pipelining is getting much higher attention is due to its ability to sometimes significantly reduce the critical path, and to increase the scheduling freedom of operation.

Park and Parker produced by far the most comprehensive treatment [Par88b, Par88c]. They discussed both the theoretical foundations and practical implementation issues. Later on, research in Prof. Parker's group has been continued with numerous studies on the effectiveness of pipelining and its relationship to other high level synthesis tasks [Jai89, Mli91]. Casavant and

his co-workers at GE developed the PISYN- high level synthesis system for the application specific pipelined hardware [Cas91]. [Hwa91] used integer programming technique for the optimization of the pipelined designs. Kurdahi discussed in detail how specific logic synthesis techniques can be used for synthesis and optimization of controller in pipelined design. Recent premier computer design conferences, ICCAD and DAC, as well as computer architecture conferences, ASPLOS and ISCA regularly include several papers on pipelining.

However, it is important to stress that in a significant part of the high level synthesis publications, only CDFG's without feedback edges are considered. Actually, even when the original examples do have feedback loops, those are sometimes ignored. This approach highly simplifies the study of pipelining and its application, but have a very limited application range. When it is applied to computations, which do have feedback in their CDFG, this results in a change in input-output relationship and therefore incorrect results.

In the digital signal processing community, pipelining is also a well discussed topic. The accent here is on how to manually apply pipelining in conjunction with other flow graph transformations for specific application areas (e.g. infinite response filters, dynamic programming based calculations such as the Viterbi algorithm), in order to achieve maximal speed-up [Par88a, Lin91, Fet90]. An excellent introduction to this type of research is again Kogge's book [Kog80].

In the compiler research, where a repeated execution of programs rarely occurs and where the most important transformations are related to control, pipelining is most often discussed during its application on loops [Lam88, Lam89, Aik90]. While this technique is often called loop winding or loop folding in high level synthesis, [Goo89, Gyr87] in the compiler literature it appears under the name of software pipelining. Although software pipelining includes additional important issues of loop prologue (initiation) and loop epilogue (finishing) effects, there is strong correlation between pipelining and software pipelining. Jouppi [Jou89] discussed the amount of pipelining potential in various classes of programs.

This section refers to functional pipelining, and treats it as a high level synthesis transformation. This approach enables a simple and straightforward inclusion of pipelining in the HYPER transformational environment, and therefore its combination with other transformations, which significantly enhance pipelining power. In HYPER, structural pipelining is treated during scheduling. Although there is a very close relationship between software pipelining (SP) and software retiming (SR) with the techniques described here, since SP and SR are used mainly in the conjunction with other loop transformation we will not discuss them here.

The work presented here differs from previously reported in both the scope and the used techniques. Four different form of pipelining are identified and discussed. For two of those problems, optimal polynomial complexity algorithms are presented. For the other two an NP-completeness proof is established and new probabilistic algorithms are used to generate a solution. The effects on the resulting hardware implementation for a number of diverse examples are reported and analyzed.

The rest of this section is organized in the following way. After a definition of the four different forms of pipelining, their computational complexity is established. Next, an optimal polynomial algorithm is presented when the goal is the minimization of the critical path. For the case when the goal is the optimal resource utilization, the objective function during pipelining is defined, and a learning while searching algorithm is used for the optimization. After a discussion of experimental results, conclusions are drawn and directions for future work are outlined.

5.3.2 Problem Formulation

In the following section, the following simple, yet effective framework of the pipelining concept will be employed: **"Pipelining with N stages is equivalent to retiming where the number of delays on all inputs or all outputs, but not both, is increased by N"**.

As we have already mentioned we treat pipelining just as yet another transformation. Strictly speaking is not really a transformation, as it changes the phase between input and output signals. However, since it preserves the input/output computational relationship, we will consider it as a transformation in the broad sense.

Depending on the goal and the level of the introduced latency, four different forms of pipelining can be identified. Those four pipelining forms, which are implemented in HYPER, are:

(1) pipelining for the minimization of the critical path;

(2) pipelining for the minimization of the critical path for a given number of pipeline stages;

(3) pipelining for resource utilization;

(4) pipelining for resource utilization for a given number of pipeline stages.

We will denote them as **CP, CP(n), RP** and **RP(n)** respectively, where *n* stands for the number of pipeline stages. Before discussing the algorithms, a precise definitions of those classes is given first.

Pipelining for critical path (CP) produces a minimal stage time. The stage time is defined as the length of the critical path of the CDFG after pipelining.

Pipelining for critical path using n pipeline stages (CP(n)) produces a CDFG with a minimal stage time, but so that exactly *n* pipeline stages are introduced.

Pipelining for resource utilization (RP) produces a CDFG which realizes the minimum area for a given timing constraint.

Pipelining for resource utilization using n pipeline stages (RP(n)) produces a CDFG, which can be implemented using a minimum area, for a given timing constraint. The number of introduced pipeline stages should equal exactly *n*.

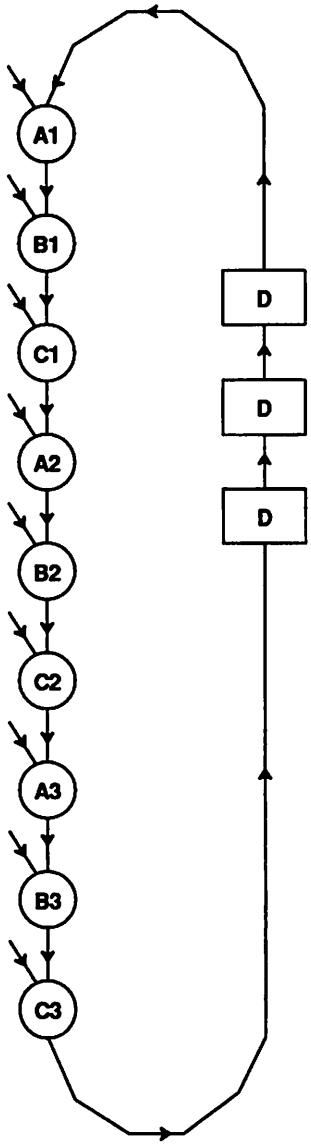


FIGURE 5.1. Initial CDFG

It is sometimes argued that pipelining for critical path and pipelining for resource utilization are the same. The discussion of the computational complexity of the associated optimization problems will denote that it is not true, but the following simple example provides intuitive insight into difference. Their relationship closely resembles the relationship between retiming for critical path and retiming for resource utilization.

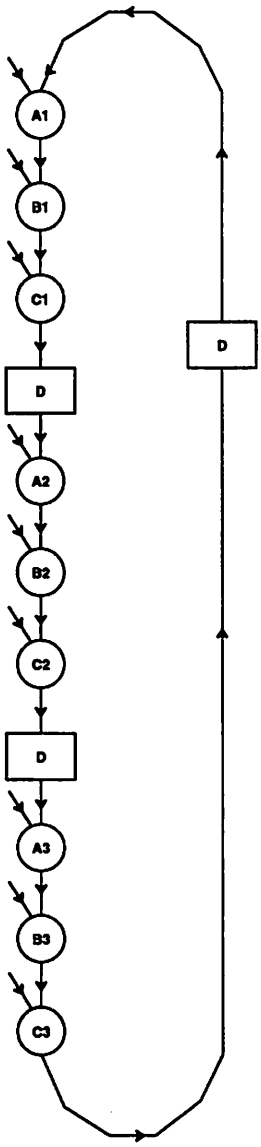


FIGURE 5.2. CDFG after the application of pipelining for the minimization of critical path

Suppose that we have the computation shown on Figure 1 in the flow graph format. Assume that each operation takes one control cycles, and that available time equals 4 control cycles. There are three different types of operations: A (operations A1, A2, A3), B (operations B1, B2, B3) and C (operations C1, C2, C3). For the sake of simplicity, we will only take the cost of the execution units into account. Due to the recursive bottleneck [Mes88], at most three pipeline stages can be introduced.

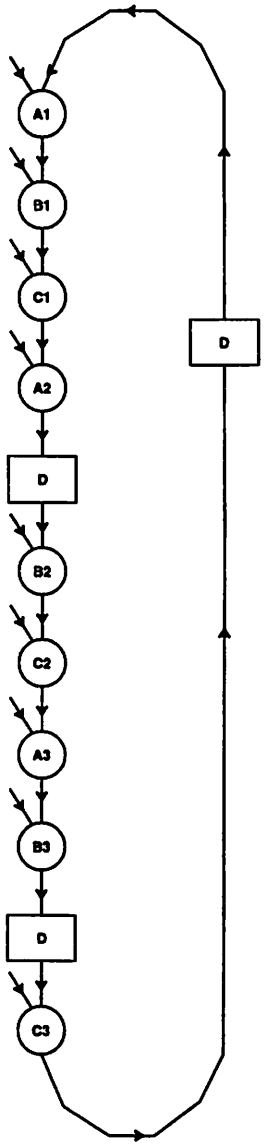


FIGURE 5.3. CDFG after the application of pipelining for the resource utilization

Pipelining for critical path will result in the transformed CDFG shown in Figure 2. The critical path is 3. It is easy to see that the optimal schedule needs at least 2 executional units of each type. A possible schedule is shown in Table 1.

Cycle	A	B	C
1	A1, A2	-	-
2	A3	B1, B2	-
3	-	B3	C1, C2
4	-	-	C3

Table 1: A schedule after pipelining for critical path

The effect of pipelining for resource utilization on the CDFG is shown in Figure 3. The critical path is 4. However, although the critical path is longer than in the former case, only the 3 executional units are needed for the realization. The schedule is shown in Table 2. Therefore, we can conclude, that pipelining for critical path and pipelining for resource utilization are indeed two different transformations.

Cycle	A	B	C
1	A1	B2	C3
2	-	B1	C2
3	A3	-	C1
4	A2	B3	-

Table 2: A schedule after pipelining for resource utilization

5.3.3 The Computational Complexity of Pipelining

Sometimes a very small change in the formulation of a problem can result in a drastic change in its computation complexity. There are many pairs of similar problems, where one is NP-complete and the other can be solved in polynomial time [Gar79]. However, there are very few such pairs in high level synthesis. It is interesting that maximal pipelining and pipelining for resource, as well as maximal pipelining for a given number of stages and pipelining for a resource utilization for a given number of pipeline stages, constitute two such pairs. While max-

imal pipelining problems have a polynomial complexity (see next section), pipelining for resource utilization problems belong to the class of NP-complete problems.

It is easy to prove that pipelining for resource utilization is an NP-complete problem. Actually, since we already proved that retiming for resource utilization is an NP-complete problem, we can use the simplest and most frequently applicable technique for proving NP-completeness: restriction [Gar79]. If we look at the NP-completeness proof for retiming for resource utilization we see that in the used CDFG it is impossible to introduce any new pipeline stages, since all operations are in recursive loops. So, if we have a polynomial complexity algorithm for either RP or RP(n) we will be able to solve the retiming for resource utilization problem, and therefore also the MAX-CUT problem, which is NP-complete [Gar79]. Therefore, we can conclude that both RP and RP (n) are at least as difficult as NP-complete problem. On the other hand, if we have a solution for either PR or RP(n) with a given assignment and schedule, we can easily check the cost of the proposed solution. Thus, both RP and RP(n) are NP-complete problems.

5.3.3.1 Pipelining for minimization of the critical path for a given number of pipeline stages (CP (n))

The definition of pipelining as retiming where the number of delays on all inputs is increased by the number of pipeline stages directly leads to the efficient algorithm in the case of CP(n). The obviously correct and efficient algorithm can be obtain by applying the Leiserson-Saxe retiming algorithm to the CDFG where the number of delays on the input edges is increased by n. Since there is an excellent and extensive literature on the Leiserson-Saxe retiming algorithm, [Lei83, Goo86, Lei87] we will not further elaborate this issue.

5.3.3.2 Pipelining for minimization of the critical path

Only a slightly more complex modification of the Leiserson-Saxe algorithm is needed to generate the fastest possible solution, using only pipelining as the transformational means. We

can add on pipeline stages a very large number of delays (say as many as the number of operations in the CDFG), and obtain the CDFG which will have the minimum stage time achievable by pipelining. If we are interested in the minimum number of pipeline stages needed for the maximal minimization of the critical path, we can use a binary search over the number of introduced delays. In that manner, we can detect the situation where we still achieve the minimal critical part, with the smallest number of pipeline stages. Of course, during the binary search, we can still use the Leiserson-Saxe retiming algorithm to solve the pipelining problem with a given number of pipeline stages.

A more elegant approach, but conceptually very similar, is to directly modify the Leiserson-Saxe retiming algorithm, so that either all inputs or all outputs (but again not both) are not connected to the host node. (See [Lei83] for a detailed and clear description of the algorithm). In this case the algorithm will automatically introduce the necessary minimum number of pipeline stages. The correctness proof of the introduced modification is straightforward.

5.3.3.3 Pipelining for resource utilization with *n* stages

Again, as in the previous two cases, the formulation of pipelining as the retiming with the increased number of delays on all inputs provides a good starting point for the design of an efficient fast algorithm. It is easy to see that it is sufficient to increase the number of delays on all inputs simultaneously by n and then to apply retiming for resource utilization in order to design the algorithm for CP(n). However, in order to make the algorithm as efficient as possible, it is necessary to make several modifications in the algorithm for retiming for resource utilization. Those modifications are needed because in the case of pipelining, especially in cases when a large number of pipeline stages are introduced, the number of delays in the graph grows extensively and therefore the solution space is far larger. Hence, the set of possible moves is growing, decreasing the effectiveness of the algorithm.

We introduced four modifications in the retiming for resource utilization algorithm, to reduce the run time significantly, while having negligible consequences on the quality of the generated solution:

(1) The adaptive search is used during the learning phase of the algorithm (during the solution space scanning). During the adaptive search, the value of the objective function is calculated after m steps. If the value of the objective function is no more then 5% larger than the smallest value of the objective functions detected until this moment, m is either reduced by 1 or takes the default value of 4, whichever is smaller. If it is within 15% of the best evaluated objective function, m will be unchanged. For other values of the objective function m will be increased by 1.

(2) As an initial solution, we use the CDFG obtained by applying the CP(n) algorithm. Although it is not often the case that pipelining for the critical path and pipelining for resource utilization are identical, the application of the CP(n) spreads the delays over the CDFG relatively uniformly, and so, creates a good starting point.

(3) We used the tabu search [Glo90] to augment both the learning and the local phase of the learning while searching retiming algorithm. The idea is to reduce the computational effort by avoiding situations where delays are moved back and forward across the same nodes, and therefore identical or similar solutions are evaluated several times.

(4) In the local search phase of the learning while searching algorithm, only moves which either reduce the critical path, or reduce the number of delays are considered. Again, the rationale behind this modification is the reduction of a run time. This time is based on the observation that relatively rarely the improvement in final phase can be achieved without the use of those two types of moves.

5.3.3.4 Pipelining for resource utilization

The pipelining for resource utilization when the number of introduced pipeline stages is not specified is the computationally most difficult problem. A straightforward method for solving this problem will be to apply RP(n) for various values for n, and select the best solution among proposed. However, as already noticed, when the number of introduced pipeline stages

can vary over a wide range of values, it will involve the solution of many instances of RP(n), and so the procedure will be slow. In order to leverage on already existing and tested algorithm and implementations, we decided to modify the RP(n) algorithm and combined it with an efficient search over n, so that a good compromise between the run time and the quality of a final solution can be achieved. The following two modifications and additions on the PR(n) algorithm were introduced:

(1) The search over the number of introduced pipeline stages is limited to the values between n_1 and n_2. n_1 is the minimal number of pipeline stages needed in pipelining for minimization of the critical path so that the critical path is shorter then the available time. n_2 is the minimal number of pipeline stages needed for the of maximal reduction of the critical path, as obtained by the CP algorithm.

(2) The learning phase of the algorithm is first applied to RP(n_2), then on RP (n_2 -1), and so on until it is applied on RP(n_1). All the time only the list of k (in our experimental studies 10) best overall solutions are maintained for the final local search.

5.3.4 Experimental Results

For testing of the proposed pipelining algorithms we used a set of 15 different instances of 7 designs:

- iir7 - 7th order IIR filter,
- iir5 - 5th order IIR filter,
- fir11 - 11th order FIR filter,
- iir11 - 7th order IIR filter followed by 4th order equalizer,
- dct8 - DCT transform for 8 points,
- decby4 - decimation elliptic filter,
- volterra2 - second order Volterra filter.

The available time in all examples was equal to the initial critical path. All designs (except dct8) were done in two ways: before and after the application of substitution of multiplications by constant with shifts and additions. In the two last cases of the Volterra filter, we also applied

time loop unrolling. The effects of pipelining for minimization of the critical path are shown in Table 3 and Figure 4. The effects of pipelining for minimization of the resource utilization are shown in Figure 5.

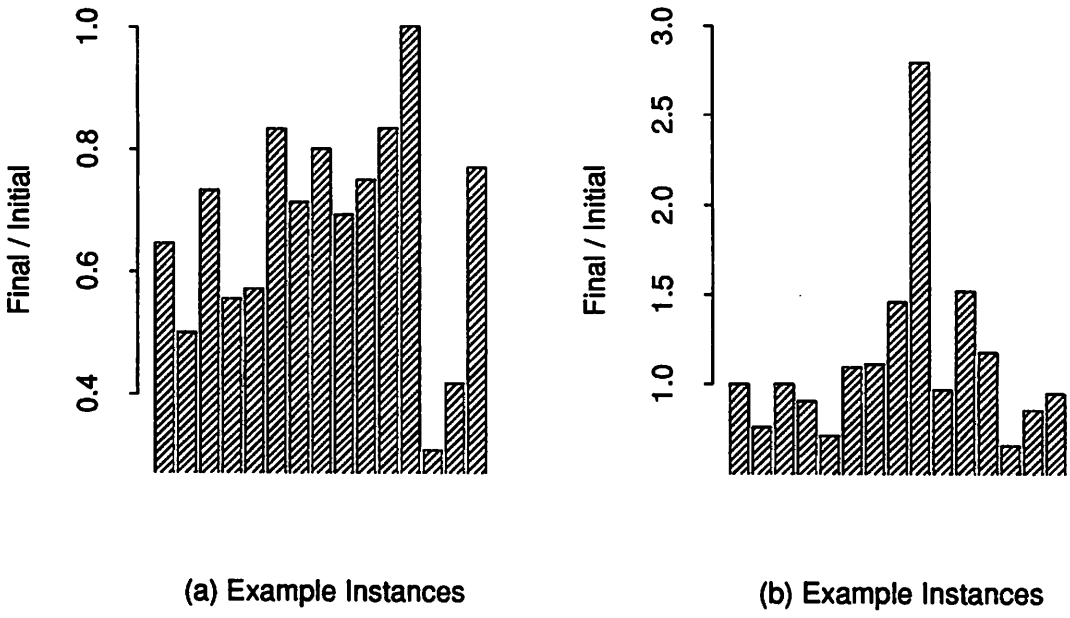


FIGURE 5.4. The changes in (a) execution units area and (b) registers area due to the application of the pipelining for the minimization of critical path

In the case of CP, the average improvement in the area of execution units was 32.5%, the median reduction was 28.6%, best improvement was 69.2%, and in the worst case the area was unchanged. For the registers the situation was very different. The average register area was larger 12.9%, the median value was unchanged, the best improvement was only 33.2%, and in the worst case the register area increased 178.8%.

In the case of RP, the average improvement in the area of execution units was 39.1%, the median reduction was 33.3%, best improvement was 69.4%, and the smallest improvement was 16.7%. For the registers the situation was again very different. The average register area was

only 5.9% smaller, the median saving was 9.4%, best improvement was 36.8%, and in the worst case the register area increased 36.4%.

Analyzing experimental results we can draw many conclusions, but the most important are:

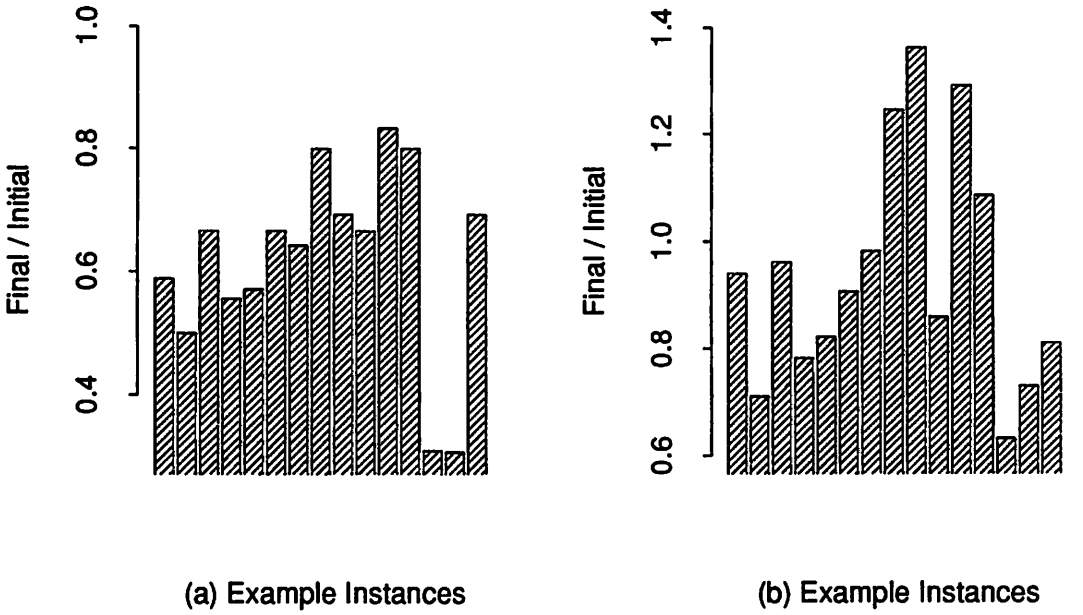


FIGURE 5.5. The changes in (a) execution units area and (b) registers area due to the application of the pipelining for the resource utilization.

(1) That although pipelining is powerful transformation for both critical path reduction, as well as resource utilization improvement, on examples with feedback loops, which have many computational elements and few delays it has very limited efficiency.

(2) Actual prediction of the effects on pipelining application should be always carefully analyzed, since they heavily depend on the ratio of the operational elements' cost to the register cost.

(3) In order to fully explore the potential of pipelining, it is necessary to combine it with other transformations.

Initially							After CP						
Example	CP	M	A	S	BS	R	CP	IPS	M	A	S	BS	R
iir7	10	3	2	1	-	33	3	3	2	1	1	-	33
iir7	13	-	2	2	2	38	6	1	-	1	1	1	29
iir5	8	3	2	1	-	26	2	3	2	1	2	-	26
iir5	10	-	3	3	3	32	6	1	-	2	1	2	29
fir11	11	2	1	-	-	28	1	2	1	1	-	-	20
fir11	11	-	2	1	3	32	4	1	-	2	1	2	35
iir11	20	3	1	1	-	55	3	6	2	1	1	-	61
iir11	57	-	1	2	2	57	6	9	-	1	1	2	83
dct8	7	5	3	3	-	33	1	6	3	2	4	-	92
decby4	14	4	5	3	-	57	3	4	2	2	2	-	55
decby4	34	-	2	2	2	41	11	3	-	1	2	2	62
volterra2	12	2	1	1	-	23	12	0	2	1	1	-	27
volterra2	15	4	2	2	6	38	10	1	1	1	1	1	25
volterra2	10	7	5	3	-	48	5	1	3	2	1	-	41
volterra2	14	1	3	2	4	58	7	1	1	3	1	2	55

Table 3: The effect of the maximal pipelining

CP - Critical Path; M - number of Multipliers; A - number of Adders; S - Number of Subtractors; BS - number of Barrel Shifters; R - number of Registers; IPS - number of Introduces Pipeline Stages.

5.3.5 Conclusion

We presented four different types of functional pipelining in high level synthesis. For the case where the objective is minimization of critical path of the transformed control data flow graph, we presented polynomial time optimal optimization algorithms. Those algorithms are based on Leiserson-Saxe retiming algorithm. When the objective is optimization of resource utilization, we proved the associated problem is NP-complete problem. For this case we developed objective function and probabilistic algorithm for the efficient transformation. All proposed

algorithms are tested on a number of examples, and experimental results are analyzed. The important conclusion of increased implementation cost due to overpipelining is supported by experimental result. Future research work includes investigation of the relationship between pipelining and other high level synthesis task, such as transformations, partitioning and module selection.

5.4 COMMUTATIVITY

5.4.1 Introduction

This section discusses the application of commutativity in high level synthesis. Although commutativity is not as common an axiom in abstract algebra computational structures as associativity (for example, the operation an a group is not always commutative), it is probably the most widely known and, in some sense, the simplest control data flow graph transformation. Unfortunately, when commutativity is treated in the high level synthesis or compiler framework it is only conceptually a simple transformation. We will prove in this section that its optimal application is associated with solving an NP-complete optimization problem. For solving this problem we will use the probabilistic rejectionless anti-voter algorithm.

This section is organized in the following way. First, the motivation behind using commutativity for resource utilization improvement is discussed. After the NP-completeness proof, we describe the objective function during application of commutativity, and then the rejectionless anti-voter based algorithm. Then the experimental results are presented. We also discuss the role of commutativity as testbed for algorithms for NP-complete problems. Finally, we draw some conclusions regarding the role of commutativity in high level synthesis.

Suppose that we want to implement the program given in Figure 1a. x and y are variables which are coming regularly every three cycles; a and b are constants. Each operation takes one cycle and the available time is three cycles. Figure 2 shows the minimum cost architecture. Both

x and y are coming directly from the inputs to the register files. It is easy to see that we need 3 registers in each register file, totaling 6 registers. However, after the application of commutativity on the first addition, the computation has the structure shown in Figure 1b. Now we can achieve the solution which needs a total of four registers, two at each register file. Therefore, the application of commutativity results in the saving of two registers.

$z_1 = x + y$	$z_1 = x + y$
$z_2 = a + x$	$z_2 = a + x$
$z_3 = y + b$	$z_3 = y + b$
(a)	(b)

FIGURE 5.1. The pseudo-code example of (a) before and (b) after the application of commutativity

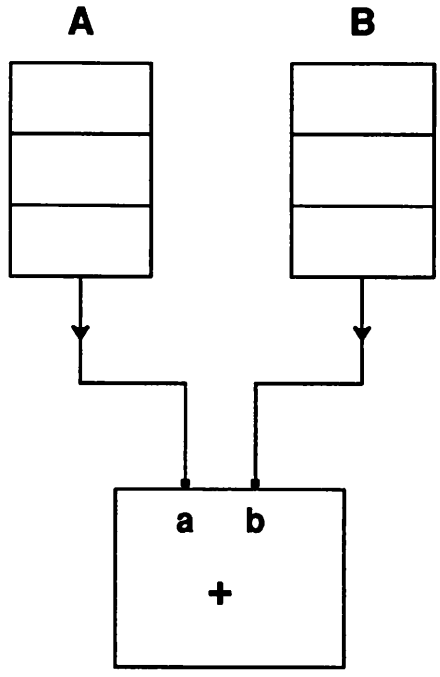


FIGURE 5.2. Minimum cost architecture for the code from Figure 5.1.

Instead of applying commutativity, we can also achieve the solution which needs only four registers if we increase the interconnect network, as shown in Figure 3. This observation can also be interpreted in the following way: if we keep the number of registers fixed, then the net effect of applying commutativity can be expressed through the reduction in interconnect.

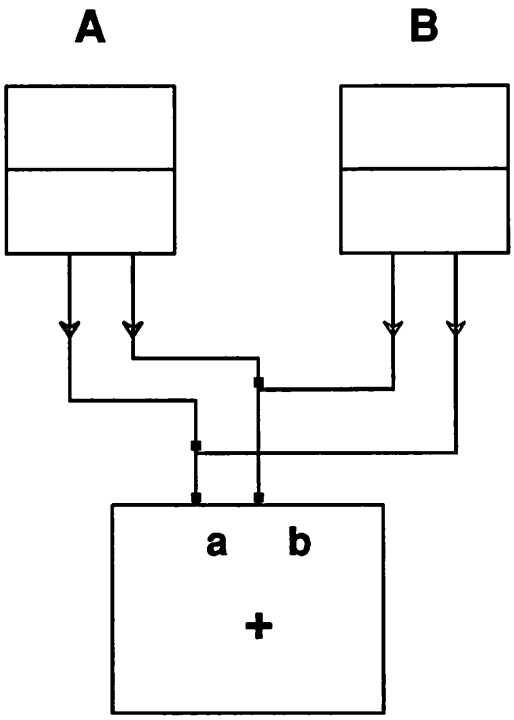


FIGURE 5.3. Another implementation of the code from Figure 5.1.

As we already stressed in Chapter 4 both register and interconnect cost are often of the same order of magnitude, if not larger than the cost of combinatorial logic. If the number of registers is fixed to 2 per register file, and the architectural model is H (as explained in the Estimation Chapter), it is clear that the only way to meet the requirements is through the use of the two adders. Therefore, commutativity can have either direct or indirect impact on all components of the hardware cost.

5.4.2 Objective Function

Our goal is to restructure the CDFG using commutativity so that the final implementation has the highest possible resource utilization. In order to achieve this goal we will use commuta-

tivity in a twofold fashion. We want to minimize both the number of the registers in register files by reducing the data broadcast, and to reduce the interconnect network by reducing the probability of two operations which have strong chances of being scheduled simultaneously sending data to the same register file. Therefore, our objective function, OF, has the form:

$$OF = \alpha_1 \times Memory\_Infulence + \alpha_2 \times Interconnect\_Influence$$

For the sake of simplicity, we will discuss objective function components under the assumption that the H model is used. It is easy to generalize to other hardware models.

If the assignment is known, it is relatively easy to measure the *Memory_Influence* component. We have to send a piece of data to at most one register file of a commutative operation. When we send data to both register files, we can increase the number of used registers. Therefore, we define *Memory_Influence* as the number of data which are sent to both registers of a register file. When the unknown assignment consequences are taken into account, the situation is more complex, but the *Memory_Influence* is still well correlated with the number of used registers. Figure 4 shows a correlation for 100 different values for *Memory_Influence* and the actual number of required registers for a 9th order IIR filter.

The badness with respect to memory of node C, C_{memory}, is defined as the number of different places where the output of node C is distributed. The set of bad nodes with respect to memory, B_M, contains nodes which contribute nonzero components to *Memory_Influence.*

Figure 5 illustrates the motivation behind *Interconnect_Influence.* This figure shows 6 nodes, which are part of a large CDFG. Next to each node are given ASAP-ALAP times. Suppose that we want the solution which uses k adders. Although the multiplication and the & node are of different types and there is no directed path between them, they can not be scheduled simultaneously. This is so because they are sending data to the same register file (the left register file of the adder). It is a non-trivial combinatorial problem to compute exactly the probability

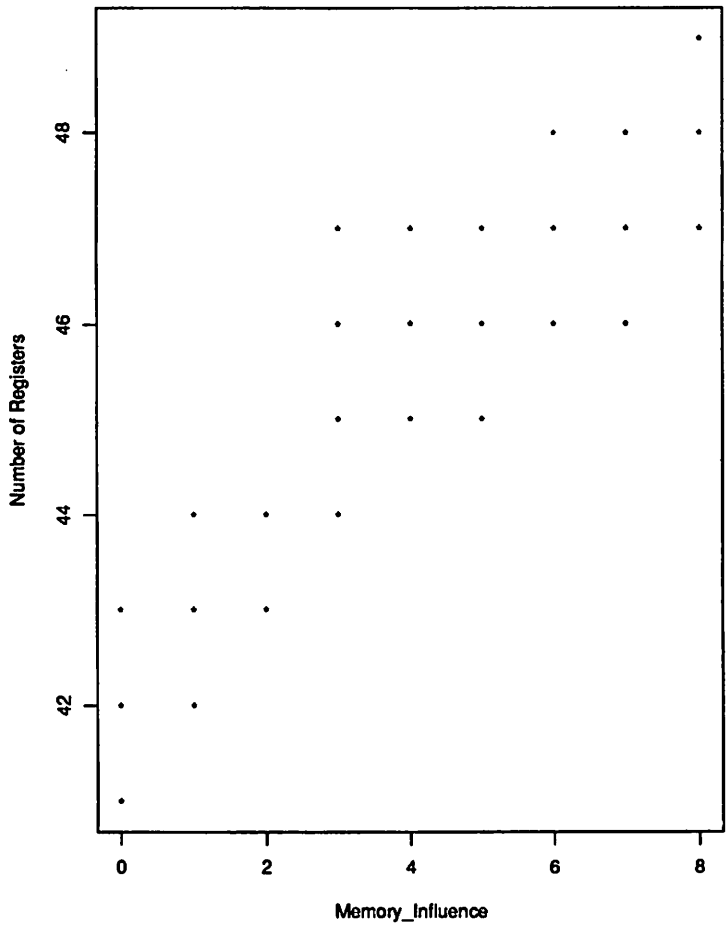


FIGURE 5.4. Correlation for 100 different values *Memory_Influence*

that in some control cycle those operations will be executed simultaneously. This probability can be approximated using the following formula:

$$P_{AB} = \frac{1}{K_k} \times \frac{T_{ABoverlap}}{A_{slack} \times B_{slack}}$$

where K_k is the number of functional units of type k in the final solution, k is the operation where both operations A and B are sending their results as input data, $T_{ABoverlap}$ is the number of

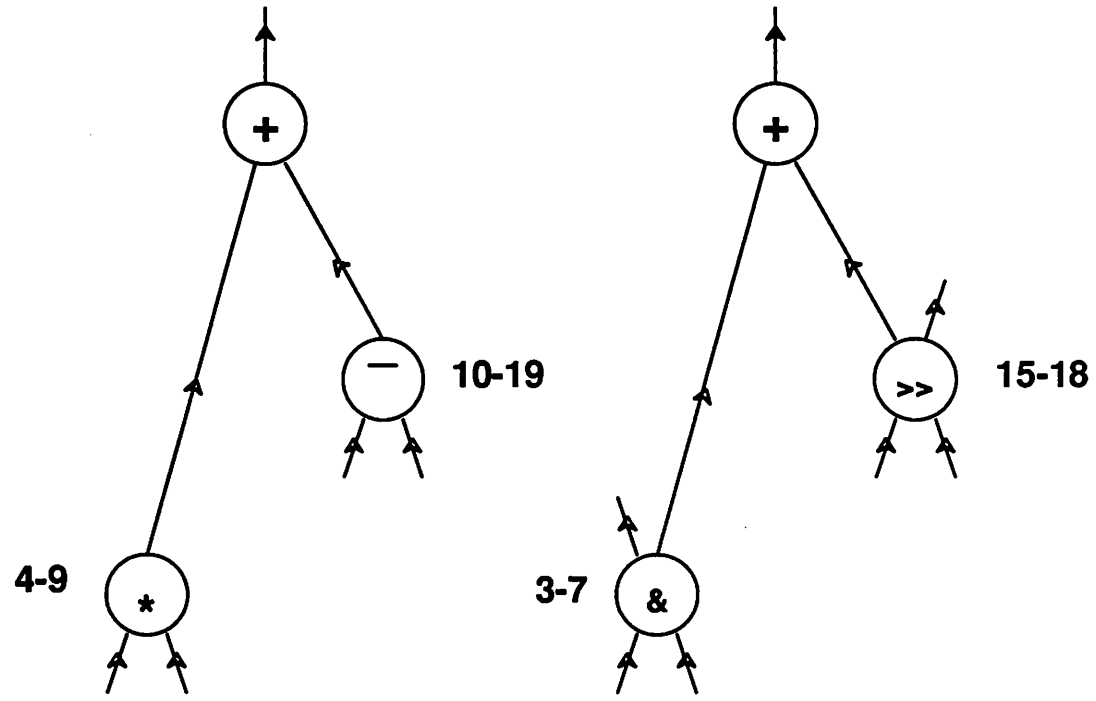


FIGURE 5.5. Commutativity objective function

control steps during which both operations A and B can be scheduled, A_{slack} is the slack of operation *A* (equal to the difference between its ALAP and ASAP time increased by one), and B_{slack} is the slack of operation *B*. In the example shown in Figure 5, $P_{*,>} = \frac{4}{6 \times 5} = \frac{2}{15}$ and $P_{*,>>} = \frac{4}{10*4} = \frac{1}{10}$. If we apply commutativity on one of the additions shown, as in Figure 5, the overlaps for operation * and >> as well as − and & are equal to 0.

The rationale behind this formula is similar to the one used during the development of objective function during assignment. Therefore, we define the badness of a commutative operation *C* with respect to interconnect:

$$badness_{interconnect(c)} = \frac{1}{k} \sum_{D \in SN, D \neq C} P_{CD}$$

SN is the set of all nodes in CDFG. The *Interconnect_influence* is now defined as the badness of all commutative CDFG nodes. When the application of commutativity on a particular node improves (reduces) the badness, that node is denoted as bad. Since there are only two possibilities for the ordering of inputs of each commutative operation, this can be easily detected.

Selection of constants α_1 and α_2 is once again an involved issue and is dependent on many parameters including bit-width, floorplanning, routing technique and registers area. During our experiment we used $\alpha_1 = \alpha_2 = \frac{1}{2}$.

5.4.3 Commutativity is NP-complete Problem

To show that commutativity application in high level synthesis is an NP-complete problem we will transform another NP-complete problem, the equal subset sum problem, to it. It is sufficient to look at the case when the interconnect network is fixed, and the goal is to minimize the number of used registers.

5.4.3.1 Equal subset sum

INSTANCE: Finite set A, size $s(a) \in Z^+$ for each $a \in A$.

QUESTION: Is there a subset $A' \subset A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

The equal subset sum problem is denoted as the NP-complete problem SP12 in [Gar79]. We will use the fact that the equal subset sum problem is NP-complete even in the case when two sets have the equal number of elements, and if those elements are ordered as $a_1, a_2, ..., a_{2n}$, and we require that each subset contain exactly one of a_{2i-1}, a_{2i}, for $i \le i \le n$ [Gar79].

5.4.3.2 Commutativity

INSTANCE: Given a computation graph $G = (V, E, C)$, where E is a set of data edges and C is a set of control edges. Given an ASIC architecture, represented by hardware

graph $HG = (HV, HR, HI)$, where *HV* is a set of executional element, *HR* is a set of register nodes each having some specific number of registers, and *HI* is an interconnect network.

QUESTION: Can *G* be transformed using commutativity to a graph G' so that G' can be implemented on architecture represented by *HG*?

It is easy to check that commutativity is in NP. It is sufficient to take G' , and guess the appropriate assignment and scheduling.

$$\begin{array}{cc} x_1 & y_1 \\ x_2 & y_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ x_n & y_n \end{array}$$

FIGURE 5.6. An instance of the equal subset problem

Suppose that we have an instance of the equal subset sum problem, shown in Figure 6. There are the *n* pairs of numbers, and the total sum of all numbers is *2N*. We will polynomially transform it to the instance of commutativity problem in the following way. The computation graph is formed so that for the pair m ($m = 1, ..., n$) of numbers p_i and q_i we form $p_i \times q_i$ products in *G*:

$$x_{m,i} * y_{m,j}, m = 1, ..., n;\ \ i = 1, ..., p;\ \ j = 1, ..., q$$

Assume that *G* can be implemented using only one multiplier. Also assume that all data for calculating all products have to be stored simultaneously in the multiplier's register files. Finally, we will assume that the only degree of freedom in the final implementation is the num-

ber of registers in the register files of the multiplier and that due to the other computation requirements in G there are exactly N registers in each of multiplier's register files. This is easy to enforce [Gar79], by modeling the dependences in the computation graph.

If we can decide how to apply commutativity on all products so that exactly N variables are in each of the register files of the multiplier, then we also have the solution for how to divide numbers in the equal subset problem so that each set has numbers with the sum of N. This is because any broadcast of any variables x or y will result in the solution with more than $2N$ registers total.

5.4.4 Probabilistic Rejectionless Anti-voter Algorithms

We used the PRAV algorithm to solve the optimization problem associated with commutativity similarly to the way it was used during assignment. We already described the notion of bad node with respect to commutativity. In this case we did not use the notion of disastrous nodes. The stopping criterion was that the number of moves is at most $2n\log n$. The algorithm also was terminated if there were no bad nodes. n is the number of commutative operations.

For the sake of completeness we present the following pseudo-code to describe commutativity algorithm.

```
Generate initial position by applying commutativity to each node with
probability 1/2;
Form lists of bad nodes B_M and B_I;
While (stopping criteria is not satisfied){
  Pick at random node b from lists of bad nodes;
  Apply commutativity on node b;
  Update the solution as well as list of bad nodes;
}
```

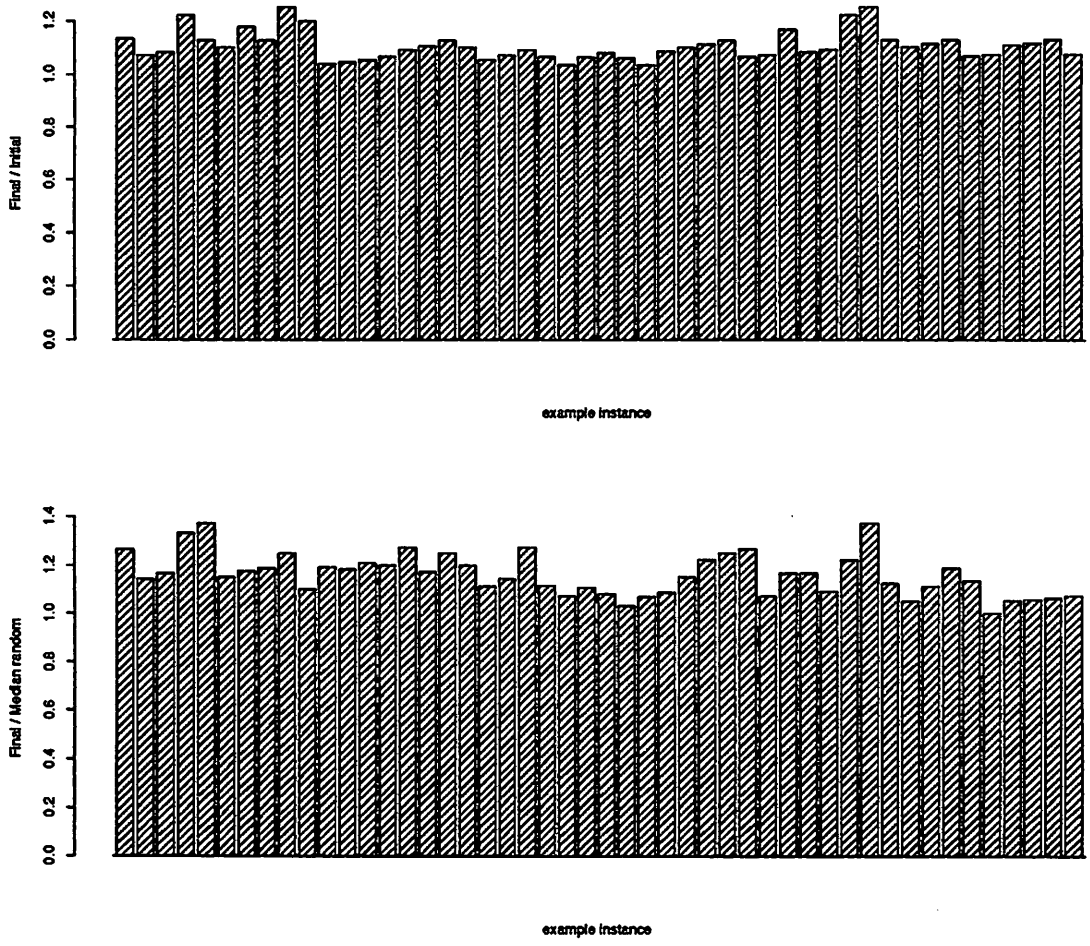


FIGURE 5.7. The benchmark set of 50 different examples

The effect of commutativity is partially due to the interconnect reduction. The interconnect cost heavily depends on the quality of the physical layout tools, which is difficult to estimate. To avoid this problem during commutativity testing, we decided to undertake experimental measurement in such a way to avoid non-deterministic influence of other design synthesis tools. It is achieved in such a way that instead of fixing the data throughput, we fixed the available hardware and measured the necessary time for the completion of the CDFG. For the benchmark set of 50 different examples, the improvement is shown in Figure 7. The median improvement against the initial implementation is 9.5%, and against the implementation where commutativity was randomly applied is 15.8%. The average improvement against the initial

implementation is 10.5%, and against the implementation where commutativity was randomly applied is 16.1%.

5.4.5 Properties of the Solution Space

Since the inputs on each commutative operation can be ordered in two ways, and all orderings are independent of each other, when there is N commutative operations in the CDFG, the solution space contains 2^N points. If the CDFG contains only commutative operations, the solution space is symmetric, in the sense that for any solution there is the solution where the objective function has the same value. This solution can be obtained from the first one by the simultaneous application of commutativity on all operation.

The size of the solution space is well illustrated by the fact that when the CDFG contains only 50 commutative operations, it already has the size of 2^{50}, with more than 10^{15} points. Even if the objective function is calculated at one billion points, fewer than 0.0001% of points are probed. Therefore, the use of the efficient optimization algorithm which explores the topological properties of the solution space is necessary.

5.4.6 Conclusion

Commutativity does not produce as spectacular a design improvement as some other transformations. However, since (1) it has broad range of application (it is applicable not only on additions and multiplications, but also on *and*, *or*, *max* and *min* operations), and (2) its application is to the greatest extent independent of the other transformation effects, so it can be applied as the last transformation before allocation, assignment and scheduling without negative influence on other transformations' effectiveness, it is an important high level synthesis and compiler transformation.

Commutativity is especially interesting as an optimization problem. It is straightforward to enumerate the size of the solution space: when we have n commutative operations in a control

data flow graph, there are 2^n different possibilities. Also, neighbor topological solution space structure is exceptionally well structured: it is an n-dimensional cube. Furthermore it is easy to show that it is symmetric. All these properties make commutativity for resource utilization an excellent testbed for the study of combinatorial optimization algorithm.

5.5 FAST IMPLEMENTATION OF RECURSIVE PROGRAMS USING TRANSFORMATIONS

5.5.1 Introduction

The transformations presented so far are used for the optimization of the resource utilization. Higher resource utilization is achieved either by increasing the throughput while keeping the available hardware resource fixed, or vice versa, by reducing the available hardware resource, while keeping the throughput fixed. All these transformations clearly have the improvement of the performance/cost ratio as a main goal.

While in a number of application specific designs an efficient hardware resource utilization is of primary interest, another common situation in signal processing is that throughput requirements are at the edge of what can be achieved using available technology. In these cases, reaching the throughput requirements is the single most important goal, even if it results in lower resource utilization rates. A general intrinsic property of many signal processing applications is that the increased latency can to some extent be tolerated. This is in contrast with some other areas, such as robotics, where the time between the acceptance of the input and the issuing of the output is most important. Therefore, during throughput optimizations, constraints on the latency should be taken into account.

This section discusses an efficient procedure to enhance the throughput under latency constraint. Although different goals result in rather different objective functions, the tools used in this section (basic algebraic and control transformations) are identical to the ones used previ-

ously. Even the algorithms have a rather similar flavor. However, now the idea is to trade efficiently silicon area for throughput.

The rest of this section is organized in the following way. First, the problem is precisely defined and previous work is discussed. Next, the idea behind the new transformation is explained using a very simple, yet real life example. After that, the computational complexity of the optimization problem associated with this transformation is derived. After a description of the proposed algorithm and a discussion of its properties, and an illustration of the proposed procedure performance on a large real life example, some conclusions are drawn and future work is outlined.

5.5.1.1 Problem formulation

When throughput rate is of primary importance and the CDFG does not have feedback edges, pipelining provides a straightforward solution. It is sufficient to introduce as many pipeline stages as are allowed by latency (e.g., add as many delays on all input, or all output edges, but not on both) and then to retime the resulting graph using the Leiserson-Saxe retiming algorithm. It is important to notice that when CDFG has feedforward edges, an introduction of a dummy transfer operation on those edges is necessary [Nic91]. Of course, the Leiserson-Saxe algorithm will do this automatically.

However, most of the signal processing algorithms have internal recursion. Examples of such algorithms include both relatively simple cases, such as infinite response and adaptive filters, and more complex ones, such as algorithms for solving systems of non-linear equations and adaptive compression algorithms. Graphs involving recursions display an upper bound on the computation rate, called the pipeline stage bound (or iteration bound) [Mes88]. This pipeline stage bound is given by $T_{pipe} = \max\left(\frac{T_l}{ND_l}\right)$. The maximum is taken over all loops l, T_l is

the sum of the computation times of all the nodes in loop l, and ND_l is the number of delay elements in loop l [Mes88].

Several researchers addressed some special program instances (e.g., IIR filters, Viterbi processor, quantizer loops) and achieved a significant progress in reducing the pipeline stage bound [Mes88, Par89a, Par89b, Lin91, Fet90, Par88a]. Our goal is to find an approach that will automatically transform arbitrary programs (including, of course, recursive programs), into a form where the pipeline stage bound is reduced to a minimum for a given latency. This can be achieved in a dual way: reducing T_l by applying algebraic transformations (associativity, commutativity and distributivity) and by moving delays (retiming). In order to provide more possibilities for both approaches, it is often necessary to do partial unrolling of the time loop.

5.5.1.2 Small example

Very often the best way to introduce a new idea is to explain its steps using a simple example. This subsection has the goal of illustrating and explaining how basic transformations can be used to reduce significantly the critical path of the pipeline stage. Neither the example nor the idea of reducing the pipeline stage is new. What is new is that minimization of the critical path in the pipeline stage is achieved by an explicit and systematic application of basic transformations. This provides a framework for the solution of the problem of the fast implementation of arbitrary recursive algorithms.

Consider the example shown in Figure 2, which represents first order IIR filter. This is actually the smallest possible example on which it is still possible to improve pipeline stage time, without going into suboperation level transformations. This filter contains only one loop with two operations: one addition and one multiplication. Both operations are in the loop. Figure 1a shows the pseudo-code. Assume that each operation takes one cycle. Since the loop contains two operations and one delay, the pipeline stage is 2 cycles long.

(a) y = x + a*y@1

(b) y = x + a * (x@1 + a*y@2)

(c) y = x + a*x@1 + a*(a*y@2)

(d) y = x + a*x@1 + (a*a)*y@2

FIGURE 5.1. 1st order IIR Filter: pseudo-code format

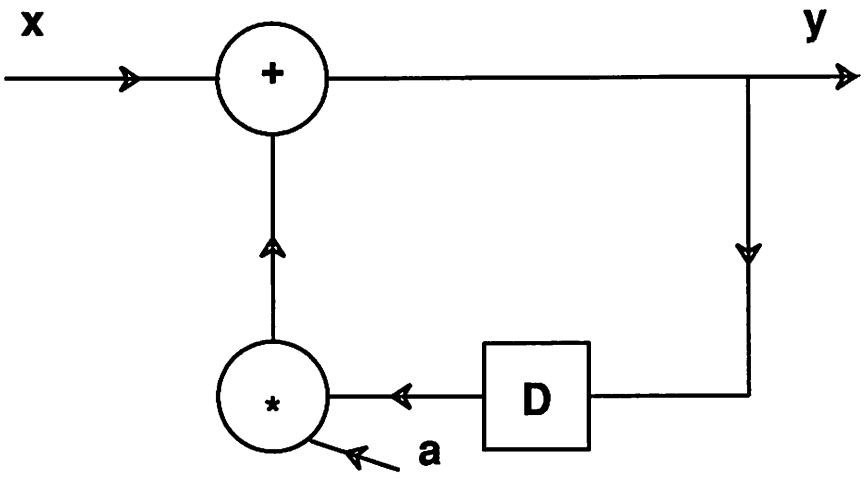


FIGURE 5.2. 1st order IIR Filter: Initial CDFG

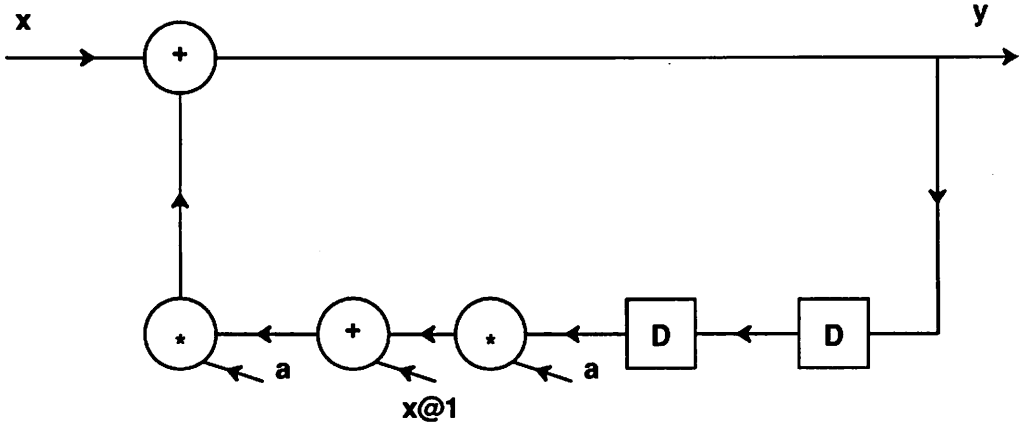


FIGURE 5.3. 1st order IIR Filter: After unrolling

Figure 1b shows the pseudo--code after the loop is unrolled once. Figure 3 shows the flow graph, after the application of this basic transformation. The main effect is that the loop now contains two delays. However, the number of operations in the loop also increased twice, so the pipeline stage, which can be achieved using retiming, is still two. Although we did not immediately profit from the application of the loop unrolling, this transformation is creating a starting position for the other basic transformations.

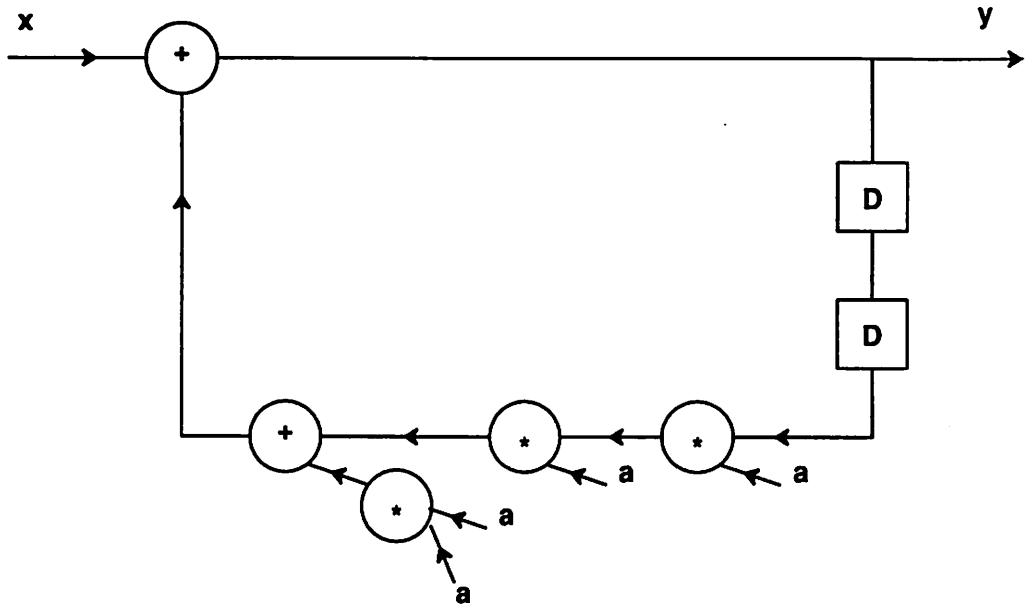


FIGURE 5.4. 1st order IIR Filter: After the application of distributivity

We can see that the associativity cannot be applied to the resulting flow graph. However, we can apply distributivity. The resulting pseudo-code is shown in Figure 1c, while the new flow graph is in Figure 4. Again, the critical path is not reduced; actually, the required amount of the hardware has increased. However, associativity can now be applied to both additions and multiplications to remove one addition and one multiplication from the loop. The effect is shown in the pseudo-code format in Figure 1d, and in the flow graph format in Figure 5. As the consequence of the application of associativity, we now have only two operations in the loop. Since

the loop also has two delays, the pipeline stage is reduced to one. This can be achieved by using retiming as shown in Figure 6. In this way we achieve the maximum throughput increase without going to suboperational transformations.

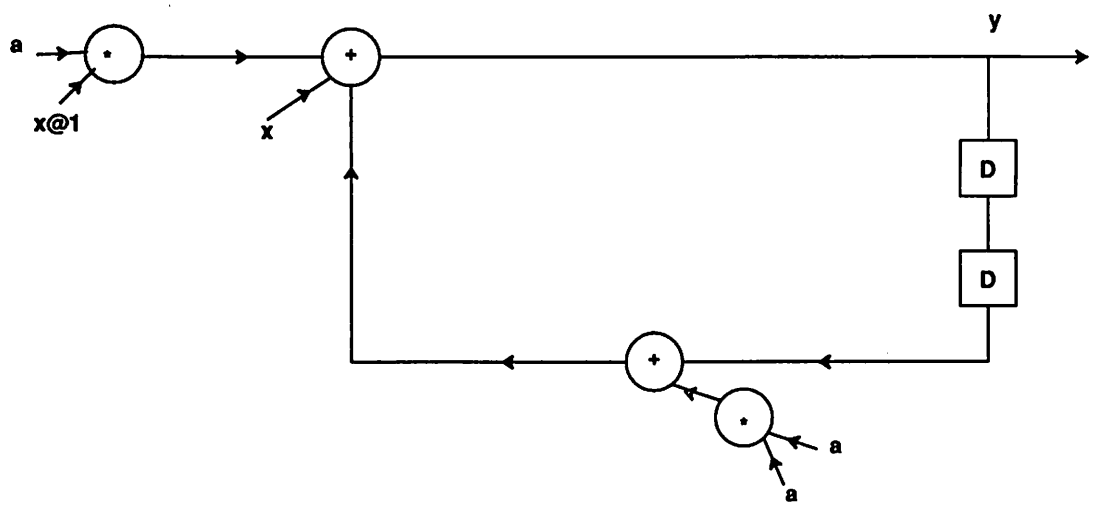


FIGURE 5.5. 1st order IIR Filter: After the application of associativity

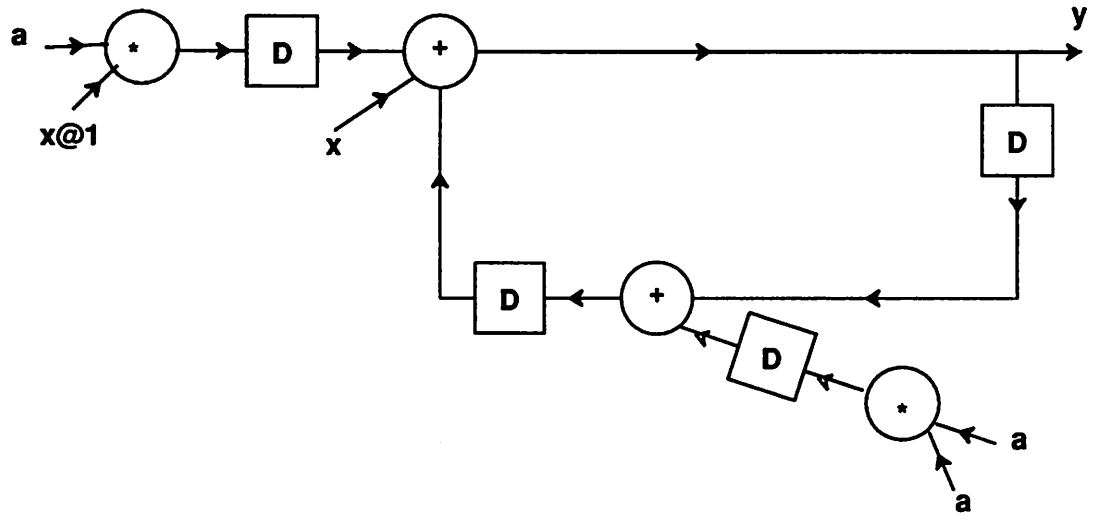


FIGURE 5.6. 1st order IIR Filter: Final CDFG

5.5.2 Computational Complexity

Now we will turn our attention to a discussion of the computational complexity of the problem.

As we already showed in the pipelining section, pipelining for the minimization of the critical path can be solved in the polynomial time using the modified Leiserson-Saxe retiming algorithm. In this process, pipelining is treated as a special form of retiming, when an arbitrary (in the case of maximal pipelining) or a fixed (when the number of pipeline stages is limited) number of delays is introduced either at all the inputs or all the outputs. Critical path minimization, using only algebraic transformations, can also be solved in polynomial time for many important CDFG classes [Bre74, Val83, Mil88, Mil87].

However, when both algebraic transformations and retiming are simultaneously applied, the associated optimization problem is NP-complete. This is true even when only one type of the operation is used in the computation, and therefore only associativity and commutativity are used as algebraic transformations.

The proof uses Karp's classical polynomial reduction approach [Kar72, Gar79]. First we will rephrase the problem in a yes-no question such that if the answer is "yes" there is a polynomial-length proof for the correctness of the solution. Then we will present another NP-complete problem, the *equal subset sum*, and, by transforming it to our problem, prove that the fast implementation of recursive programs is at least as difficult as the subset sum problem and therefore also NP-complete.

5.5.2.1 Fast implementation of recursive program

INSTANCE: Computation graph $G = (V, E, D)$, where each node $n \in V$ is of the same type, E is the set of edges, and D is a set of delays on some edges from E. All operations are of the same type. The operation has the commutative and associative properties.

QUESTION: Is there a graph $G' = (V, E', D')$, which can be obtained from G, using the finite number of applications of retiming as well as associativity and commutativity transformations and has a pipeline stage of at most K cycles?

5.5.2.2 Equal subset sum

INSTANCE: Finite set A with n elements. Size $s(i) \in Z^+$ is defined for each $i \in A$.

QUESTION: Is there a subset $A' \subset A$ such that $\sum_{i \in A'} s(i) = \sum_{i \in A - A'} s(i)$?

$$S(1) = 6,\ S(2) = 7,\ S(3) = 4,\ \ldots\ ,S(n) = 5$$

FIGURE 5.7. Instance of the Equal Subset Sum problem

The equal subset sum problem is denoted as NP-complete problem SP2 in [Gar79].

It is easy to see that once we have graph G', we can find its critical path by using leveling according to input, after reverse topological ordering through depth first search as described in Chapter 3. Figure 7 shows an instance of the equal subset sum problem. Figure 8 shows the instance of the fast implementation of a recursive program which is the result of the polynomial transformation from the instance of the equal subset sum problem. The graph G contains two large loops B and C, each with the very large number of elements, M (M >> 2n). Between the nodes $i_b \in B$ and $i_c \in C$, for $(1 \leq i \leq n)$ there exists another loop. Those loops have two delays. Each of those loops also has only one operation. This operation is also common to a loop i. The loop i has $s(i)$ elements where $s(i)$ is the size of the element i in the equal subset sum problem. All these loops also have only one delay. Both large loops B and C also have exactly one delay.

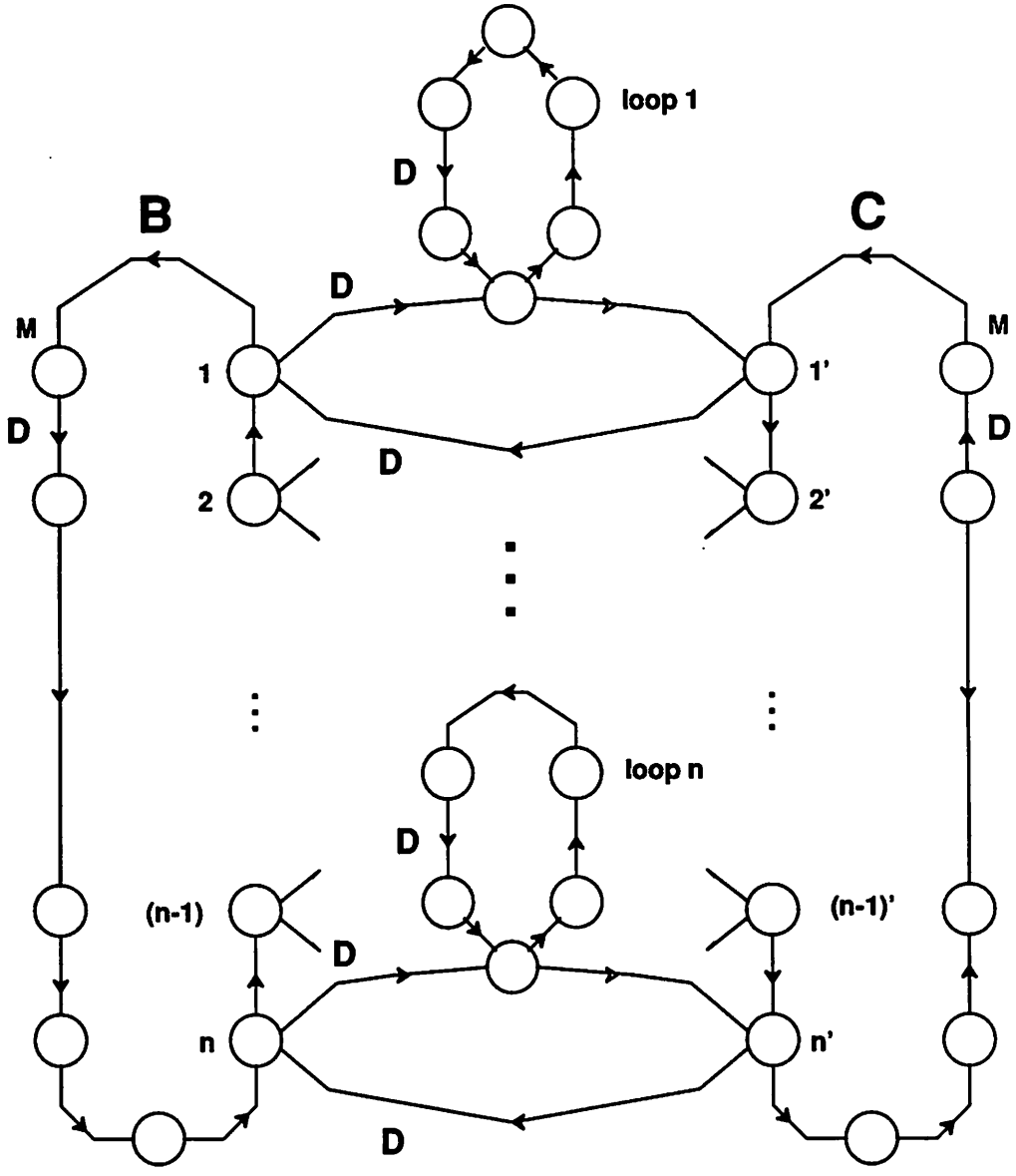


FIGURE 5.8. Instance of the Fast Implementation of Recursive Program

It is easy to see that the graph G is constructed in such a way that after an arbitrary retiming and breaking of loops at delays position, we will always have two disjoint subgraphs. The total number of nodes is $2M + \sum_{i \in A} s(i)$. All elements of a loop i will be a part of one of those subgraphs. The position of delays in loops connecting A and B will determine to which

subgraph all elements of a particular loop i will belong. If we can find a solution which has

$M + \frac{\sum_{i \in A} s(i)}{2}$ elements in each subgraph, we also have the solution for the equal subset sum problem. It is sufficient to say announce that all elements which belong to the subgraph which contains initial elements of loop B are in the same subset.

Therefore, the fast implementation of recursive programs problem is at least an NP-complete problem.

5.5.3 Fast Implementation of Recursive Programs using Transformations: Procedure

A simple, yet efficient procedure for transforming an arbitrary computation graph so that it can be implemented with the very short pipeline stage, can be given using the following six steps, described by the following pseudo-code:

1 *Using distributivity and associativity, move as many operations as possible out of the cycles;*

2 *Unroll the time loop. The number of unrollings is bounded either by hardware or timing constraints;*

3 *Repeat step 1;*

4 *Retime the graph so that the structure of the resulting graph is such that algebraic transformations will have a maximal effect;*

5 *Reduce the critical path of the resulting graph using either the Valiant or Miller algorithm (which optimally applies associativity and distributivity);*

6 *Introduce sufficient pipeline stages (using a revised Leiserson's retiming algorithm);*

Steps 1, 3, 5 are reducing the iteration bound using algebraic transformations. Step 4 is a crucial step, which provides valuable pre-processing for the final iteration bound reduction in step 5. Step 6 is the final auxiliary step.

One of the main problems to be addressed when implementing the retiming step is the selection of the objective function. A simple and easy to compute function would be the number of nodes contained in the largest pipeline stage. It is often possible to reduce the critical path of a pipeline stage close to the optimal $\log_2 n$, where n is the number of nodes in that stage. A more accurate (but computationally more expensive) function can be used for the important class of applications which use only multiplications, shifts, additions and subtractions. Linear as well as adaptive filters are part of this class. In this case, the smallest possible critical path of the graph equals (log d) (log C + log d), where C is the number of operations in the pipeline stage and d the degree of polynomial represented by pipeline stage. The same formula can also be used when the computation only contains addition as well as min and max operations. Examples of such applications can be found in the areas of neural networks, Markov modeling, dynamic programming and fuzzy logic. The same also holds for computations containing only logic "and" and "or" operations (logic synthesis). This isomorphism was first observed by Miller [Mil87]. The retiming transformation itself can be implemented using a statistical approach, identical to the technique described in [Pot91a].

5.5.3.1 Example: Volterra second order polynomial filter

The effect of the procedure is illustrated using a second order Volterra filter (Figure 9). It is a polynomial non-linear filter [Mat91], and previously has not been discussed in the context of fast implementation of recursive programs. We assume that each operation takes one cycle.

The critical path of the initial CDFG is 12, and it is denoted by the bold lines. The application of pipelining cannot reduce the critical path. However, the application of the just described procedure results in a reduction of the critical path to only 4 control cycles, even when unrolling is not applied. An application of unrolling will result in an additional reduction, but for the sake of clarity we will only discuss the case where unrolling is not used.

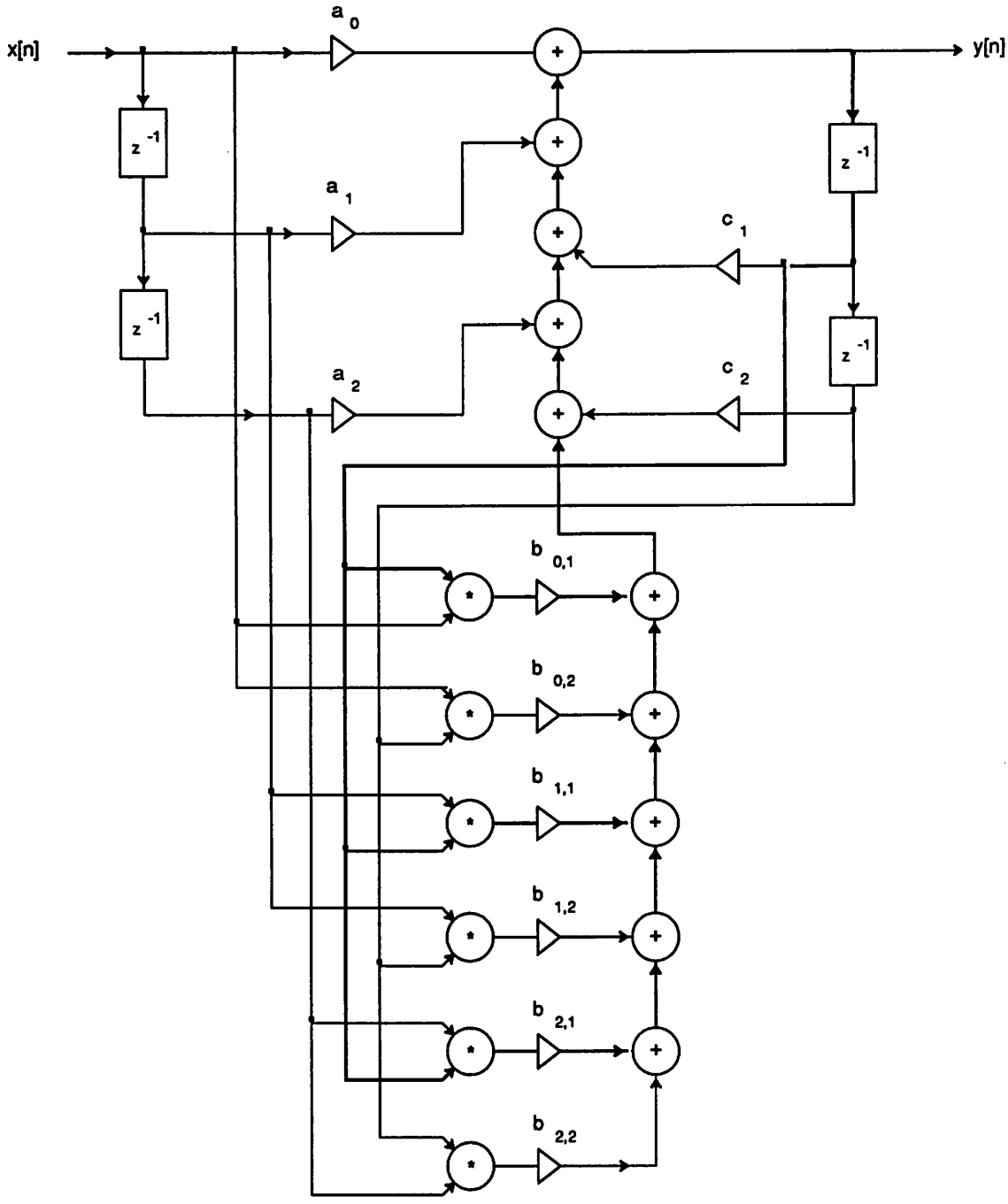


FIGURE 5.9. Volterra Filter: Initial CDFG

Application of step #1 reduces the critical path to 9, as shown in Figure 10. Note that both multiplications by constants and all additions which can be pipelined are removed from the loop which is denoted by the bold line. Since we are not using unrolling, we will skip steps 2 and 3.

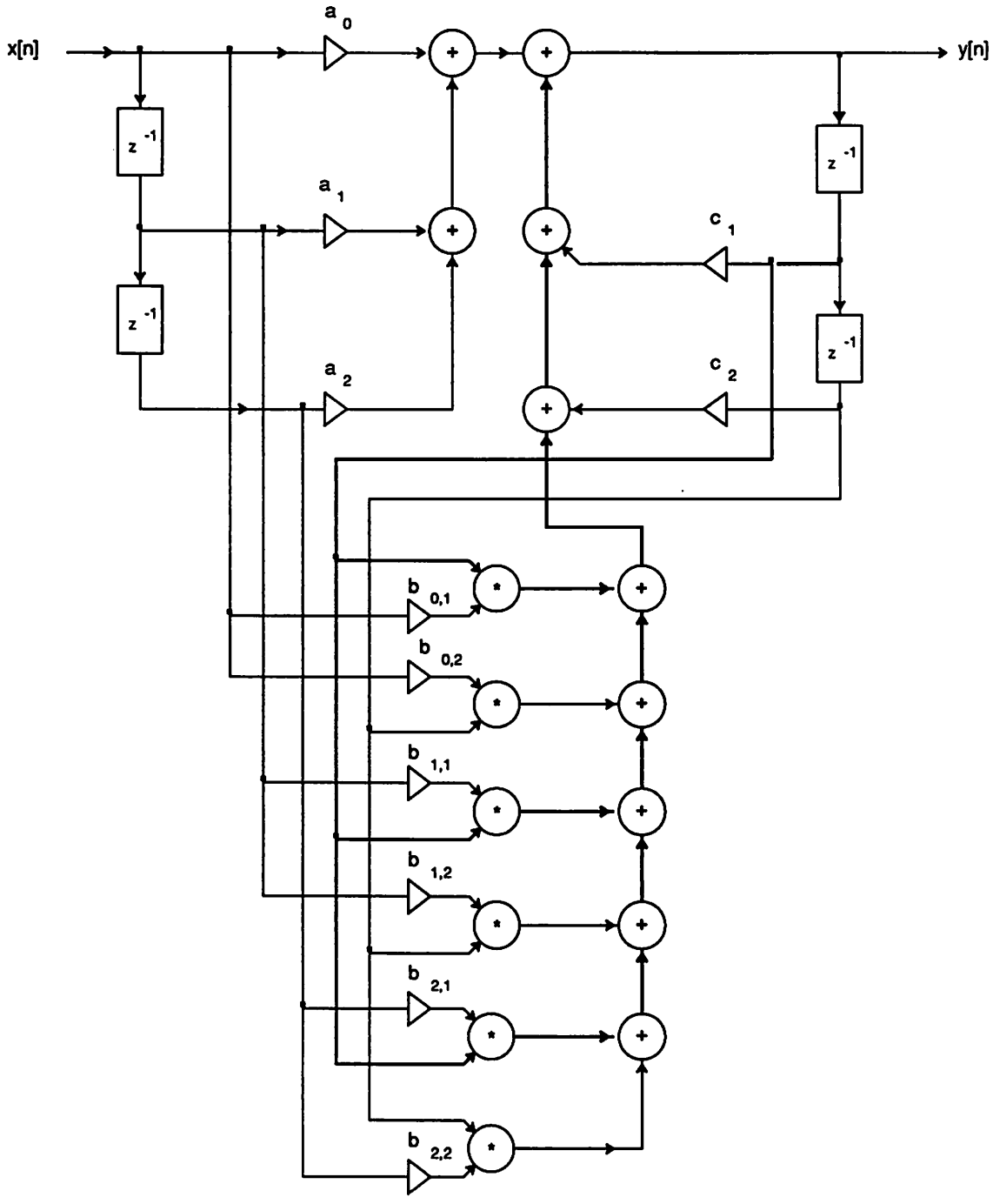


FIGURE 5.10. Volterra Filter: After step 1

After the application of step 4, we obtain the graph shown in Figure 11. Using associativity we have "deconvoluted loops" and using retiming we have separated the two feedback loops. Now the critical path is 7. Finally, by using the associativity applied on the adder tree (the chain of

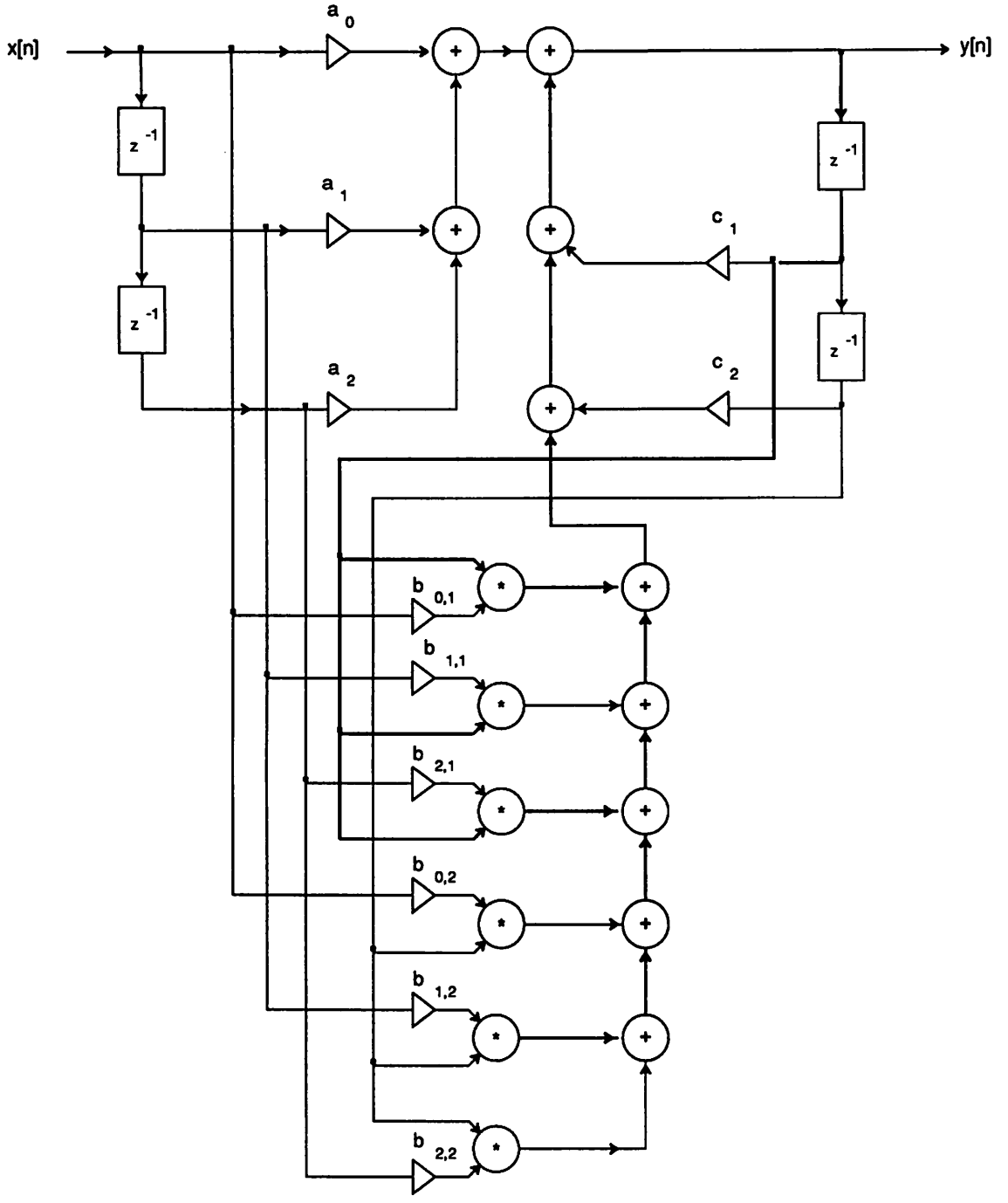


FIGURE 5.11. Volterra Filter: After step 2

additions connected by the bold lines) in Figure 12, we can get the CDFG with the critical path 4. Since all CDFG parts connected to the input can be easily pipelined, and other loops have a shorter critical path, we achieved a threefold reduction of the critical path.

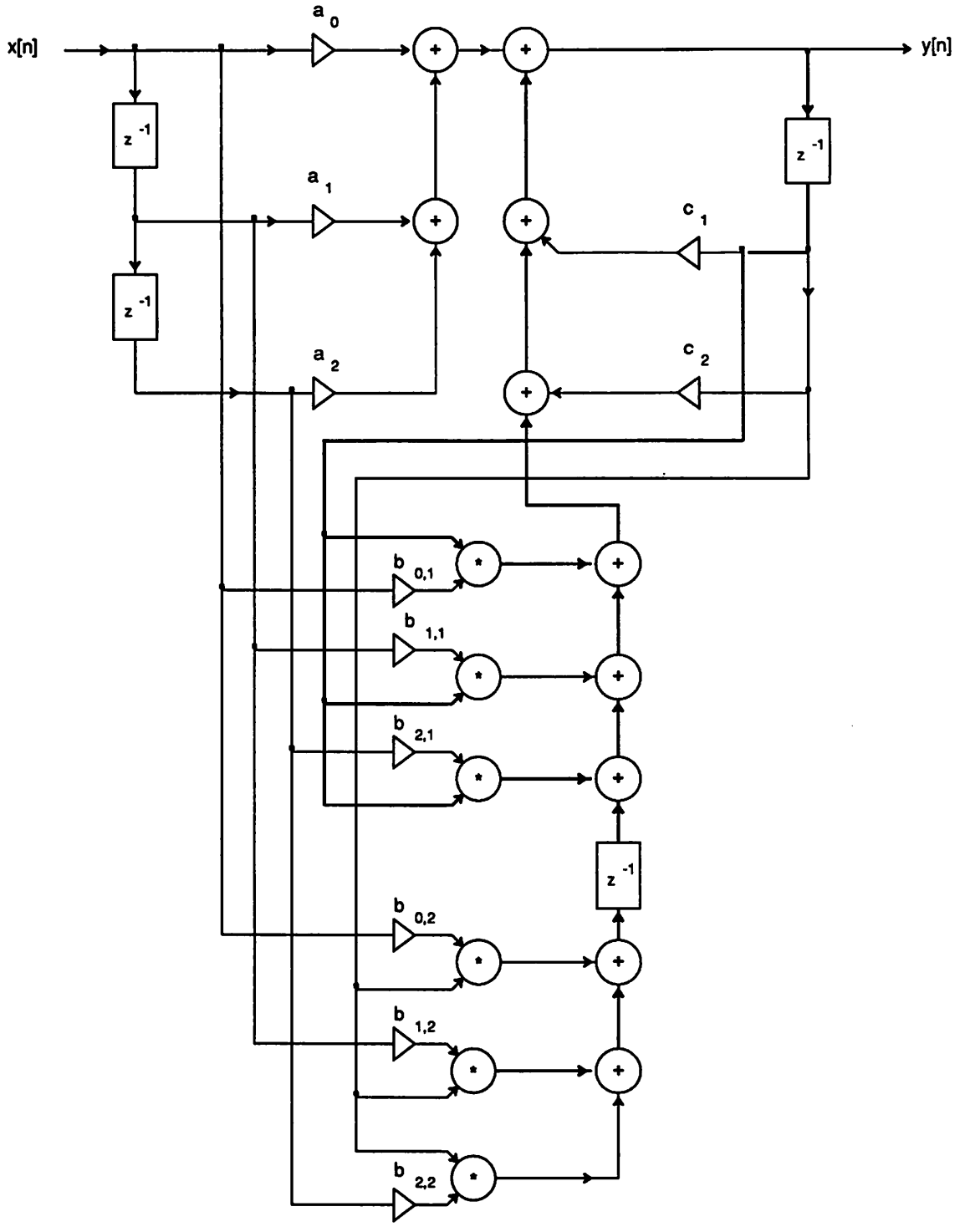


FIGURE 5.12. Volterra Filter: After step 4

5.5.4 Procedure Properties

Several observations can be made about the proposed procedure for transforming arbitrary

CDFG for fast implementation.

During retiming, it is necessary to use both associativity and distributivity in order to get the best results. The rationale is very similar to the one discussed during retiming for resource utilization, i.e., sometimes it is necessary to change the structure of a control data flow graph in order to apply retiming more optimally. Therefore, for this step we can use the same algorithm as for the retiming for resource utilization. The only modification is a different objective function.

During step 4, the key profit often comes from the so-called "loop deconvolution" (as in the Volterra filter example). This refers to the minimization of the number of edges which belong to more than one loop using associativity. This can be by enforced using an appropriate objective function when associativity is applied.

A good objective function is the number of nodes in a pipeline stage. It is often possible to reduce the critical path to near optimal $\log_2 n$, where n is a number of nodes in a pipeline stage. A better objective function can be used in several important special cases: (i) when the computation graph has only additions, multiplication and subtractions; (ii) when it has only min, max and addition operations; and (iii) when it has only "*and*" and "*or*" operations. The first case is important in signal processing, e.g., for many filter structures [Bla85], the second one in a large number of artificial intelligence and game theory applications as well as in fuzzy logic [Kos91]. The third one has a significant application in logic synthesis [Bra84].

5.5.5 Conclusion and Future Work

Fast implementation of recursive programs using associativity, distributivity and pipelining is probably the transformation with the highest potential among the proposed transformations. Besides the appropriate experimental study on its effectiveness and limitations over a wide class of examples, which implies full implementation and incorporation in the HYPER system, there are many other interesting issues. For example:

(i)what other important special cases (computational structures) can be solved more efficiently during step #5 than in a general case?

(ii) what can be achieved by going to suboperational level (the most obvious example is to incorporate pipelinable units, e.g., multipliers)?

(iii) what is the maximum achievable speed-up if the hardware resources are limited?

(iv) how much can be profited by using this transformation for the speed-up of software loops (instead of software pipelining)?

5.6 CONCLUSION AND FUTURE WORK

A number of powerful control data flow graph transformations are presented in Chapter 5. They provide convenient and efficient way to improve significantly performance/cost ratio. Although those transformations are interesting on their own, it is our conviction that the major contribution is more the developed methodology, than the transformations themselves.

At least four directions for farther research can be envisioned:

(1) Development of faster, better optimization techniques for application of these transformations;

(2) Development of objective functions which will more accurately describe wanted effects;

(3) Development of new transformations, in particular those which will combine power of common subexpression and algebraic laws with the control structures of programs;

(4) Development of transformations environment, which will automatically decide in which order and to what extent some transformation will be applied.

6

PROBABILISTIC REJECTIONLESS ANTI-VOTER (PRAV) OPTIMIZATION ALGORITHM

6.1 INTRODUCTION

Until now, the primary research described in this thesis was the development of algorithms for high level synthesis tasks. Now, we turn our attention in a different direction; mainly to investigate other properties of the proposed algorithms. In the previous chapters we proposed several novel algorithms, including *learning while searching* and *heuristics randomization.* In this chapter we will discuss one of the new algorithms, the *probabilistic rejectionless anti-voter* (PRAV) algorithm, as a general purpose tool to solve difficult optimization problems. We already showed in previous chapters that this algorithm can be efficiently applied to two high level synthesis NP-complete problems: assignment and commutativity. Our goal here is twofold: (i) to show that the proposed algorithm presents a competitive option for solving many important NP-complete problems such as graph partitioning and graph coloring; (ii) to demonstrate that the

proposed algorithm provides a suitable framework for the solution of a large class of combinatorial problems.

The rest of this chapter is organized in the following way. First, the importance of the class of NP-complete problems is discussed. Also, four NP-complete problems (graph partitioning, graph coloring, vertex cover and the traveling salesman problem) are defined and their complexity as well as application range are discussed. Graph partitioning is used as an example during this discussion. In the experimental part of this chapter we will compare simulated annealing (as one of the highest quality techniques, according to independent research) and the proposed algorithm using a graph partitioning problem as the test vehicle. Then we will demonstrate how the new algorithm can be also used for solving the graph coloring, the vertex cover, and the traveling salesman problems. This chapter is concluded with a discussion of the properties, the theoretical foundation, and the application range of the proposed algorithm as well as its relationship with other probabilistic algorithms.

6.2 NP-COMPLETE PROBLEMS

Since the mid sixties [Cob64, Edm65], and especially after the early seventies [Kar72, Coo71] impressive progress has been made in the area of computational complexity. Although the most intriguing question about the relationship between classes P and NP is still not resolved, there is a relatively clear computational complexity classification.

The most important computational complexity boundary in the so-called "spectrum of computational complexity" [Tar83], which graphically describes computational complexity classification, divides all problems into two classes: the tractable and intractable problems.

The tractable problem region includes problems for which an efficient algorithm exists. Although the speed of the algorithms for those problems varies a lot, for all of them there exists an algorithm with a running time which is some polynomial function of the problem size. Exam-

ples of such problems are sorting, matrix multiplication and linear programming. Those problems belong to class P in the vocabulary of computational complexity.

The intractable problems include at least three subclasses: undecidable, superexponential and exponential problems. The most difficult among them are the undecidable problems. For problems of this class proof exists that they can not be solved regardless of the used technique. Problems from the two other subclasses can be solved. However, there is always also proof that any algorithms for those problems will take at least superexponential or exponential time respectively. References and examples for intractable problems can be found in [Tar83].

Somewhere on the boundary between tractable and intractable problems are the NP-complete problems. Although the precise definition of the class of NP-complete problems is rather involved [Gar79], a rough intuitive description is relatively simple. NP-complete problems are problems for which the correctness of the proposed solution can be checked in polynomial time, but currently there exists neither a polynomial complexity algorithm nor a proof that such an algorithm does not exist. Furthermore, it can be proven that if we find a polynomial complexity solution for one of them, it implies a polynomial solution for all of them. Due to the work inspired by Karp's research [Kar72], there is a strong belief that there exists no polynomial complexity algorithm for problems of this class.

Unfortunately, a large number of important engineering and scientific problems of both practical and theoretical interest belong to the class of NP-complete problems. It is widely believed that there is no polynomial run time algorithm for them. Therefore, a massive effort has been invested to develop algorithms which are as good as possible. Excellent references are [Gar78, Joh8x].

We conclude this section by describing four NP-complete problems. Later, one of them (graph partitioning) will be used as a testing vehicle for several combinatorial optimization algo-

rithms, including the new probabilistic rejectionless anti-voter algorithm. We will also show how the other three problems can be solved using the proposed approach. The four problems are generic in the sense that a large number of other problems can be treated as a generalization of these. Therefore, an efficient solution for the graph partitioning, the graph coloring, the clique partitioning and the traveling salesman problem is the prerequisite for a successful solution to a broad class of problems.

6.2.1 Graph Partitioning

Graph partitioning is a well known problem with a variety of possible formulations. In this section we will discuss the following version [Gar79]:

INSTANCE: Graph $G = (V, E)$, weights $w(v) \in Z^+$ for each $v \in V$ and $l(e) \in Z^+$ for each $e \in E$, positive integers K and J.

QUESTION: Is there a partition of V into disjoint sets V_1, V_2, ... , V_m such that $\sum_{v \in V_i} w(v) \leq K$ for $1 \leq i \leq m$ and such that if the set of edges have their endpoints in two different sets V_i, then $\sum_{e \in E'} l(e) \leq J$?

The results obtained here can easily be applied to the more general problem where each node and edge is described by vector weight, with appropriate modification in constraints.

Graph partitioning is an NP-complete problem [Hya73]. In Garey and Johnson's book, it is denoted by ND14. The problem is NP-complete for any given K, $K \geq 3$, even if all vertex and edge weights are 1. The problem can be solved in polynomial time for $K = 2$ using matching.

Recently, Johnson, Papadimitrou and Yannakakis introduced a new complexity class, PLS, which provides better insight into the complexity of partitioning [Joh8x]. Partitioning is a

generic problem for this class, and its complexity for finding even a local minimum is higher than polynomial, but less than exponential if P ≠ NP [Joh88].

6.2.2 Graph Coloring

Graph coloring is one of the generic NP-complete problems, and one of the most studied, with a broad spectrum of applications. Several of these applications were mentioned in the HYPER description. The problem can be stated in the following form [Joh79]:

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |V|$.

QUESTION: Is G K-colorable, i.e., does there exist a function f: V $\rightarrow$ 1,2,...,K such that $f(u) \neq f(v)$, whenever u,v $\in$ E?

The problem is solvable in polynomial time for $K = 2$, but remains NP-complete for all K $\geq$ 3. The problem is denoted by GT4 in Garey and Johnson's book.

6.2.3 Independent Set

We already discussed use of the independent set problem in high level synthesis during the calculation of maximum bounds on available parallelism. This problem is especially interesting since it can be used as just another viewpoint on two other widely occurring NP-complete problems, being vertex cover and clique. These three problems are usually defined in the following way:

PROBLEM: Independent set

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |V|$.

QUESTION: Does G contain an independent set of size K or more, i.e., a $V' \subset V$ such that $V' \geq K$ and no two vertices in V' have an edge in E between them?

PROBLEM: Clique

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |V|$.

QUESTION: Does G contain a clique of size K or more, i.e. a $V' \subset V$ such that $V' \geq K$ and no two vertices in V' have an edge in E between them?

PROBLEM: Vertex Cover

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |V|$.

QUESTION: Is there a vertex cover of size K or more, i.e. a $V' \subset V$ such that $K \leq |V|$ such that for each edge u,v in E at least one of u and v belongs to V'?

The Independent set is denoted by GT20, clique by GT 19, and vertex cover by GT1 in [Gar79]. The following simple lemma from [Gar79] establishes the relationship between independent set, clique and vertex cover:

For any graph $G = (V, E)$ and $V' \subset V$, the following statements are equivalent:

(1) V' is a vertex cover for G;

(2) $V - V'$ is an independent set for G;

(3) $V - V$ is a clique in the complement G^c of G, where $G^c = (V, E^c)$, with

$$E^c = \{\{u,v\} : u, v \in V \text{ and } \{u,v\} \notin E\} .$$

6.2.4 Traveling Salesman Problem

The traveling salesman problem is probably the most popular and studied NP-complete problem. Several thousand papers have been published on its complexity and various algo-

rithms. A fascinating, in-depth study is presented in the book: "Traveling Salesman Problem" [Law85].

INSTANCE: Set C of m cities, distance $d(c_i, c_j) \in Z^+$ for each pair of cities $c_i, c_j \in C$, positive integer B.

QUESTION: Is there a tour of C having length B or less, i.e., a permutation $< C_{\pi(1)}, C_{\pi(2)}, \ldots, C_{\pi(m)} >$ of C such that

$$\left(\sum_{x=1}^{m-1} d(C_{\pi(m)}, C_{\pi(i+1)}) \right) d(C_{\pi(m)}, C_{\pi(1)}) \leq B$$

The traveling salesman problem is denoted by ND22 in [Gar79].

6.3 ALGORITHMS FOR NP-COMPLETE COMBINATORIAL PROBLEMS

There are numerous ways for addressing NP-complete combinatorial problems. They include tailored heuristics (often based on the steepest descent paradigm), mathematical programming techniques (in particular, linear and integer programming), dynamic programming, various gradients algorithms, through relaxation to eigenvalue problem, backtracking, and the recently popular and successful probabilistic and neural network algorithms. Among the probabilistic approaches the most popular are simulated annealing, simulated evolution, genetic, tabu and local search.

Among the neural network optimization algorithm back and forward propagation, Boltzmann machine, various gradient inspired, and meanfield have received the most attention. For many NP-complete problems, including the graph partitioning problem, it is shown [Joh91] that simulated annealing is one of the most competitive alternatives.

6.4 DESCRIPTION OF THE NEW ALGORITHM

Probably the best way to introduce a new algorithm is to demonstrate how it can be used to solve specific problems. Before we describe application of the new algorithm to graph partitioning problem we will introduce two definitions.

Definition 1: In the graph partitioning problem solution, a bad vertex a is the vertex which has at least one neighbor vertex b (between a and b exists an edge), which is not in the same group as a.

Definition 2: The badness of vertex a, denoted by $B(a)$ is the number of neighboring vertices which are not in the same group as a.

A more precise definition of how much a given vertex a increases the cost of the solution is probably a ratio of the badness to the total number of neighboring nodes for vertex a, since it is logical to expect that vertices with a large number of neighbors have more neighboring vertices which increase the cost of a solution. However, this calculation will involve division (which is a computationally intensive operation) and more importantly, it seems logical to pay more attention to vertices which are more important in cost.

PRAV treats the problem from the standpoint of groups, instead of nodes. We first choose one group I, according to the probability $P(I) = \frac{f(B(I))}{\sum_J f(B(J))}$, where $B(I)$ and $B(J)$ represent the total badness of all nodes in I and J respectively. In other words, we select a group (or partition) according to the probability that removing a node from it will decrease cost function. After that, we choose from group I of bad nodes one node a which will be moved to the other group according to the probability $P(a) = \frac{f(B(a))}{B(I)}$. Finally, we select a new group J for

the selected node a, according to the probability $P(I \rightarrow J) = \frac{N(a,J)}{\sum_{K \neq J} N(a,K)}$, where $N(a,J)$ and $N(a,K)$ are the number of neighboring nodes to node a in groups J and K. As time proceeds, we can increase the probability of choosing nodes with a greater badness, by changing function $f(.)$.

If during the process some group violates some of the constraints (e.g., the total weight of all the nodes which can be accommodated) we give it the maximal badness, and remove a node from it to satisfy constrains.

The new algorithm can be described using the following pseudo-code:

Get an initial assignments of elements to groups;
Form list of bad elements in groups;
While (stopping criteria not satisfied){
    Pick a random group I;
    Pick a random element a;
    Pick a new, random membership for a;
    (all picking according to probabilities described in the text);
    Update solution, bad lists, and functions f(.);}

When we apply this approach to the graph coloring problem, we first randomly color all nodes with K colors. As bad vertices we consider nodes which as neighboring node have a node with the same color. We are favoring changes to a color more if this color is less represented among the neighboring nodes. When applying the PRAV algorithm to a clique partitioning problem, we first reduce the problem to a graph coloring on the complement graph.

To solve the vertex cover problem we can first choose a random set of K nodes as elements of the vertex cover. The set of disaster nodes includes all non-covered nodes, and they should be included in the set according to the number of other non-covered nodes which will be covered with their inclusion as a priority measure. (Disaster nodes are defined as nodes which have such high badness that they prevent any feasible solution) But before we include some of the disas-

trous nodes in the vertex cover, we need to remove a node from the current vertex cover. Prime candidates (bad nodes) are those which cover nodes which are covered by some other node in the vertex cover. The badness is proportional to the percentage of already covered neighbor nodes.

Finally, in order to solve the traveling salesman problem using the probabilistic rejectionless anti-voter algorithm, we first choose randomly n edges in the graph. The set of disastrous nodes includes nodes which are not connected to at least two other nodes, or which are connected to at least three other nodes. A bad node is the one which is connected to other nodes in the current path with edges which are not the two shortest ones among its incident edges. The badness is proportional to the ratio between the length of currently selected edges to connect the particular node to its neighbors and the distance between this node and its two closest neighbors. We always favor the inclusion of short edges, which reduce the number of disastrous nodes.

6.5 EXPERIMENTAL RESULTS

The test presented in [Joh89, Joh91] is becoming virtually the de facto standard for the comparison of various algorithms for NP-complete problem algorithms in general. Johnson also discussed in great detail the performances of several algorithms for graph partitioning. He uses two classes of randomly generated graphs - the standard random graph and the more structured geometric graph. According to the number of authors, these two classes represent enough variety and similarity to the real applications that they should provide a reasonable testbed for a performance study of the suggested algorithms.

6.5.1 Random Graph (G_N)

The standard random graph [Bol85] $G_{N,p}$ is defined by the parameters N and p. The parameter N defines the number of vertices in the graph, and parameter p specifies the probability that any pair of vertices has an edge between them. Thus, the average degree of the graph $G_{N,p}$ is (N -

1)p. For p fixed, and independent of N, we are faced with a dense graph as N grows large. In this case, the optimization problem becomes trivial from an engineering standpoint, since all solutions have nearly the same value and the ratio of the optimum solution to the mean solution approaches 1 [Pet88].

On the other hand, if we fix the degree of the graph, the ratio of the optimum cutsize is approximately equal for all N. [Pet88] Therefore, fixed degree graphs provide a better discrimination of the algorithm performance as N grows. The fixed degree graphs also seem to be closer to the real problems.

6.5.2 Geometric Graphs (U_N)

Another class of random graphs, which probably is closer to the real applications, is the geometric graph $U_{N,d}$ defined by the parameters N and d, $0 < d < 1$. This type of graph stresses the notion of spatial clustering and local connectivity. A geometric graph $U_{N,d}$ is generated by randomly distributing the N vertices on a unit square. Two vertices are connected if and only if they can be enclosed by a square of length d [Joh88]. Thus, the average degree of vertices away from the border of the unit square is $4Nd^2$. Once again, as in the standard random graph, the fixed degree geometric graphs are probably closer to the real examples. Also, these graphs provide a better discrimination of the algorithmic performance as the ratio of minimal to random cutsize decreases as N increases.

6.5.3 Large Random and Geometric Random Graphs Generation

It is straightforward to generate the described random graphs with sizes of up to several thousand nodes. However, if one wants to generate very large graphs, especially with millions of nodes, a direct implementation is impossible. For 10^6 nodes, we have more than 10^{11} pairs of nodes, and to look at each of them will take months even on the fastest computers. We solve this problem by using the following procedure.

6.5.3.1 Large standard random graph

For the generation of a standard random graph, we use a procedure whose correctness is easy to verify.

First, we generate the number of edges in the whole graph as a sample from Binomial distribution, [Pre88] with parameters n and p, where n is the number of nodes, and p is the probability that any pair of vertices has an edge between them.

Then, we generate a uniformly distributed random number r between 1 and n^2, and assign an edge between nodes l and j such that $l = r/n$ and $j = r - r/n$, if $l \neq j$, if one does not already exist between them.

The presented algorithm has an expected running time linear with respect to the number of edges in the random graph.

6.5.3.2 Large geometric random graph

For the generation of a geometric random graph with one million nodes we first divide the unit square in 10,000 identical subsquares. To determine the number of nodes in each of these subsquares, we again use sampling from Binomial distribution, this time with parameters n = 100, and p as the desired probability according to the formula described in the previous paragraph. Finally, if it is necessary, we can add or delete a number of nodes so that the total number of nodes equals n. Among all nodes in any given subsquare we put an edge. To check for edges among nodes in different subsquares, we should check for nodes from one subsquare to nodes of at most 8 other subsquares.

Once again, it is easy to see that running time is linear with respect to number of edges, and it is easy to verify the correctness of the procedure.

6.5.4 Experimental Results

We tested the PRAV algorithm on examples of standard and geometric random graphs with the size of 20, 100, 500, 2000, and 1 million nodes. All nodes and edges in the generated graphs had the same weight. For the smallest example we partitioned the graph into 4 groups; for three of the medium size, into 10 groups; and finally, the largest example, we set the number of groups to 100. In the following tables we compare the percentage in improvement in performances due to the application of the new algorithm over classical simulated annealing for 4 small graphs. We ran both algorithms 10 times on 10 different random examples for each graph and the presented values compare the medians, means, and best and worst performance. Due to exceptional computational requirements of the largest example, we ran only the PRAV algorithm on 2 examples. It took 18247 seconds (slightly more than 5 hours) to complete the example on SUN 3/60. The solution was more than 500% better than in the case of a simulated annealing application during the same amount of time. (30,217 vs. 166,727 for geometric random graph).

Classical simulated annealing algorithm			
Random graph	**median**	**mean**	**best - worst solution**
U 20	17	17	14 - 24
U 100	39	44	36 - 53
U 500	187	208	179 - 267
U 2000	594	695	547 - 814
G 20	35	35	33 - 40
G 100	166	169	130 - 196
G 500	685	692	629 - 719
G 2000	2942	3056	2761 - 3202

Table 1: Simulated annealing performance

The PRAV algorithm			
Random graph	**median**	**mean**	**best - worst solution**
U 20	14	14	12 - 16
U 100	25	26	24 - 28
U 500	56	54	51 - 71
U 2000	208	212	196-220
G 20	31	31	29 - 32
G 100	137	139	129 - 148
G 500	609	612	596 - 632
G 2000	2503	2508	2465 - 2564

Table 2: The performance of the PRAV algorithm

Comparison Simulating Annealing vs. New Algorithm		
Random graph	**solution size**	**time**
U 20	0.95	1.12
U 100	0.91	0.71
U 500	0.52	0.14
U 2000	0.48	0.017
G 20	0.97	1.37
G 100	0.93	0.89
G 500	0.88	0.16
G 2000	0.86	0.023

Table 3: Running times, simulated annealing running time is normalized to 1

It is very difficult to say anything definite about the quality of the PRAV algorithm, but first results look very promising. We expect further improvements in the performances by tuning

the algorithm parameters. In Table 3 we compare the running times of the new algorithm and simulated annealing algorithm.

We also tested the simulated annealing on two "real life" examples. We partitioned an application specific computer in 4 modules. The best result obtained using simulated annealing had costs of 17 and 12, new algorithm produced solutions with costs of 10 and 5 [Pot89, Chu89].

6.6 PROPERTIES OF THE PRAV ALGORITHM

The PRAV algorithm combines ideas from two sources: interacting particles [Lig85] and a rejectionless technique for simulated annealing [Gre86]. There exists a strong relationship between other interacting particle model (Stochastic Ising Model) and simulated annealing [Lig85], as there is between the Voter Interacting Particle Model and the proposed algorithm [Lig85]. Other interacting particle models (such as the contact model and the exclusion process) are also candidates to be incorporated in the efficient combinatorial optimization techniques.

This connection can be used in proving some theoretical properties of the PRAV algorithm, for example, convergence. D. Walsh was the first to apply the anti-voter model to graph coloring using 3 colors [Pet89], but did not generalize it to the other combinatorial optimization problems. While rejectionlessness does not improve the quality of the solution, it dramatically reduces running time.

6.6.1 Convergence

A very simple argument provides the proof of convergence, under the assumption of an arbitrary large running time. Suppose that we have a solution which is characterized by some set of values for all nodes. If this is not the optimal set of values, then, by randomly picking the correct values one by one, we can make the transition to the optimal solution. For this we have a small but finite probability. If we are not lucky after at most n steps (n is number of nodes) we

can apply the same reasoning again. In such a way, we can make probability as large as we want (as a trade-off to running time) to find the optimal solution.

6.6.2 Rejectionless

As is well known [Gre86], it is very easy to detect that as we are approaching the final solution, we are generating moves which do not change the solution with high probability. The exact ratio is highly dependent on the structure of the graph which we are trying to partition. For random graphs and graphs which model the CAD problems, an empirical observation is that we usually need between $\frac{1}{100}\ n$ and $\frac{1}{2}\ n$ of proposed moves to generate one which we will accept. If n (n is the number of nodes in the graph) is large, then we will have thousands of rejected moves before we find one which will improve the solution. This time is, by far, the major part of the run time.

In the PRAV algorithm, all moves are always accepted, which makes it much faster, and still we do not change the nature of the probabilistic approach. It is achieved by concentrating our attention to the parts of the graph which really make up cost (bad and disastrous nodes). Rejectionless is efficiently achieved due to the fact that for many NP-complete problems it is easy to update bad and disastrous node lists.

6.6.2.1 Explicit constraint satisfaction network

It is interesting to notice that it is relatively easy to cast the PRAV algorithm in a neural network framework. The nature of the algorithm implies that an appropriate name for a new neural network is *explicit constraint satisfaction network.* Essentially, this approach is similar to the transition from the simulated annealing to the Boltzmann machine neural network. This point of view provides a way for the realization of efficient parallel implementation of the PRAV algorithm. A detailed discussion of this viewpoint can be found elsewhere [Pot89c].

6.6.2.2 Relationship to other general purpose optimization techniques

The PRAV algorithm can be easily combined with several other probabilistic search techniques. For example, the cooling mechanism used in simulated annealing and tabu search mechanism [Glo90] can improve performance. These issues are discussed elsewhere [Pot90]. However, since the overhead computation cost is large it seems that this direction is not very promising.

6.7 CONCLUSION

In this thesis several new algorithms for NP-complete problems have been introduced, including the *learning while searching* algorithm and *randomized heuristics*. In this chapter, one of them, probabilistic rejectionless anti-voter algorithm, is discussed. As the test problem, graph partitioning is used. We compared the performance of the PRAV algorithm to one of the most powerful published techniques, simulated annealing. The PRAV algorithm is faster and produced a solution of better quality then simulated annealing. For the testing of the suggested algorithm a procedure for the efficient generation of very large standard and geometric random graphs was presented. Finally we discussed the properties of the proposed algorithm and very briefly its relationship to other general purpose optimization techniques.

7

CONCLUSION

To conclude this thesis, this chapter briefly reviews the major results presented in the previous chapters and discusses the intrinsic properties of the presented methods and algorithms with respect to the research work outside high level synthesis framework. Finally, an outline for future research on the lines provided by this thesis is discussed.

7.1 SUMMARY

This thesis presented several new concepts and new algorithms used in HYPER, the high level synthesis environment for real-time systems with data-path intensive architectures.

HYPER differs from the multitude of other high level synthesis systems by explicitly addressing hierarchy in the signal/control data flow graph, the application of transformations and estimations during design process, and by a consistent use of global quality measure and resource utilization. Actually, the consistent use of the resource utilization as an explicit goal

effectively merges all high level synthesis tasks, including the allocation, assignment, scheduling, transformations, and estimation. It also provides both efficient optimization framework and convenient user interface.

New allocation, scheduling and assignment algorithms for hierarchical control data flow graphs were developed in this framework. A new transformation environment as well as several new high level synthesis transformations (including retiming and associativity, commutativity, and fast implementation of recursive programs) based on novel algorithms were introduced. The application of one of those algorithms, probabilistic rejectionless anti-voter, to a number of diverse NP-complete tasks was also presented.

The effectiveness of the proposed algorithms and the concepts was demonstrated in multiple ways: using standard benchmarks examples, with the aid of statistical analysis and through a comparison with estimated minimal bounds. For a development of sharp minimal bounds, the novel discrete relaxation technique was used.

7.2 RELATED PROBLEMS

Although techniques developed for HYPER are often problem specific, they can be used with minimal adjustment in several other areas. In this section we outline four areas where these techniques and algorithms can be efficiently applied with additional straightforward, although nontrivial, efforts. These areas are:

(1) Compilers for Concurrent Architectures;

(2) Optimization of Complex Hierarchical Problems;

(3) Combinatorial Optimization Algorithms; and

(4) Estimations for other CAD areas.

7.2.1 Compilers for Concurrent Architectures

There is a consensus [Hen91] that the most important challenge in future compiler research is the exploitation of parallelism.

Although right now a close relationship between high level synthesis compilers, such as HYPER, and software compilers does not exist, this will be changed in the near future. The most likely reason for the current situation is that compilation tasks are significantly larger than those addressed in high level synthesis, and at the same time there are much stronger constraints on compilation time. However, with proliferation of superpipelined, superscalar, and very long instruction architectures, as well as high scale multiprocessor systems, exploration of parallelism will become a dominant issue in software compilers. Due to its hierarchical approach, fast algorithms and the fact that allocation and assignment precede scheduling, HYPER can be easily adjusted to those tasks. It is sufficient to remove the allocation procedure, and superimpose, during the assignment, constraints which define the given concurrent architecture. One such project, a refinement of HYPER to compiler for the PADDI system, is underway [Che90, Rab91a].

7.2.2 Complex Hierarchical Problems

Due to the rapidly increasing complexity in various engineering problems and computer-aided design in particular, hierarchy treatment becomes an unavoidable task. One of the main differences between HYPER and many other high level synthesis systems is that it treats hierarchy explicitly. This approach, where for each level in a hierarchy scheme as much information as possible is extracted from lower hierarchy levels using both prediction tools and feedback information, appears both as a simple as well as powerful solution. Also, techniques for gathering information from lower hierarchy levels during estimations using min and max bounds established by discrete relaxation as well as statistical techniques and measuring level of difficulties during task solution at lower level are rather general approaches.

7.2.3 Combinatorial Optimization Algorithms

Although algorithmic techniques proposed in HYPER are not revolutionary, they provide a good compromise between diverse algorithm performances: running time, quality of the generated solution, robustness and difficulty of implementation. In Chapter 6, one of them, the probabilistic rejectionless anti-voter algorithm, is discussed in more detail. It would be interesting to study to a greater extent the other proposed techniques. All of them are rather general techniques that can be easily modified and tailored to a particular application, as well as combined with other techniques.

7.2.4 Estimations and Predictions in other CAD and Optimization Areas

An application of deterministic prediction techniques, where the original computationally intractable problem is relaxed to a problem of polynomial complexity, is by no means restricted to the high level synthesis domain. The design process in diverse application areas can be often naturally divided into a number of successive tasks, as it is done in high level synthesis. The consequences of decisions made in early stages of design process have to be estimated fast and accurately in order to avoid numerous iterations. For all those estimations besides statistical techniques, deterministic bounds can be derived using relaxation techniques as is done in HYPER.

7.3 FUTURE RESEARCH

As often happens in science and engineering, an attempt to answer one set of challenging questions does not only produce answers, but also produces a number of even more challenging questions. During the development of HYPER, several major novel issues arosed which currently are either not addressed at all, or very limitedly addressed in the design automation literature. On the other hand, the attempt to evaluate potentials and limitations of the proposed techniques and algorithms fully also provides interesting avenues for further research.

Direct possibilities to continue research along the lines provided by HYPER are described at the end of corresponding chapters. Here we will discuss more general directions. Even though the current HYPER environment already provides the algorithmic design community with the capability to evaluate and compare various design alternatives and provides the integrated circuits community the capability to select among various algorithmic alternatives in a matter of days, the development of here mentioned new techniques will greatly extend those capabilities and turn HYPER, or other high level synthesis systems, into an effective and complete design environment not only at the chip level, but also at the system level. At the same time, this research will have potential for a profound influence on areas outlined in the previous section.

7.3.1 A Background Memory and Input/Output Optimization

High performance algorithms are often constrained by input-output and background memory bandwidths rather than by computational requirements. The background memory is the memory outside of a datapath.

Almost a decade ago, Buccher and Jordan [Buc83] did an extensive study of the influence of memory organization on resource utilization in a general purpose computing environment. They compared the performance obtained on a Cray X-MP when an ideal secondary memory design is available (a very large secondary memory) or when the secondary memory is a disc. The experimental program was an application of a polynomial conjugate gradient method to the solution of a system of linear equations generated by a 27-point operator applied on a three dimensional grid of the size $45 \times 45 \times 45$. In all cases, input/output transfers were overlapped with computation in order to achieve maximum speed. The findings of the experiment showed that:

(1) when the secondary memory had a size of 1 million words the computation was done at the rate of 1.2 MFLOPS.

(2) when the secondary memory had a size of 8 million words the achieved performance was 58 MFLOPs.

(3) for a secondary memory with 32 million words, computation was constrained exclusively by the data path throughput, and could be done at maximum rate of 161 MFLOPs.

Thus, the appropriate background memory can provide on the real life problems a speed-up of more than two orders of magnitude even on general purpose computing systems. In application specific systems, where we have additional degrees of freedom in datapath design, influence can be even higher. On the other hand, the memory cost can dominate the cost of a system [Sto91].

Therefore, the presence of a memory and I/O optimization environment is essential to obtain high quality solutions. This topic is only marginally addressed in the current HYPER system. Research efforts in this area in a high level synthesis framework have been rather minimal as well. Only recently have we seen some publications in this direction (e.g., [Ver89, Lip91]). Even in a general purpose computing environment, research regarding input/output and memory optimization is relatively limited in comparison with datapath research [Pet90].

A memory and I/O bandwidth optimization environment should have the following parts (for the sake of brevity, we will limit the discussion to memories here):

- *estimation*: Similar to the data path synthesis process, a careful estimation of the required memory bandwidth and the potential access conflicts is essential to achieve a global solution. Techniques such as discrete relaxation (eventually extended with a statistical approach) can be exploited here as well. The results of the estimation drive high level synthesis tasks, as described below.
- *module selection*: Combining the results of the estimation process with memory access patterns (sequential versus random), a specific memory module can be selected from the module library. Techniques similar to the ones used in the data path module selection process can be exploited here.

- *transformations*: Simple transformations can help to minimize the memory requirements and bandwidths, such as the staggering of the memory accesses in time or the temporal storage of variables in foreground memory. On the other hand, extreme memory bandwidths can be solved by partitioning the memories or by duplicating data in multiple units. Flow graph transformations also can minimize the number of memory accesses or create a well defined memory access order (such that cheaper memories such as FIFO's and shift-registers can be used). An example of a particularly promising transformation is value numbering [Wai84].
- *memory unit allocation*: This process determines the number of memory units of each type to use. This part can be integrated in the overall HYPER allocation search process.
- *binding*: The background variables (and arrays) are assigned to a particular memory module and the exact storage position is determined. This requires a careful lifetime analysis of the variables. Although some memory area can be gained by considering individual elements of an array, in general it makes more sense to consider arrays as a single entity when performing life-time analysis. Graph techniques such as graph coloring and clique partitioning are the most obvious choice for the implementation of the binding process.

7.3.2 Design for Low Power

The impressive progress of the integrated circuits technology over the last two decades through technology scaling and system development has primarily resulted in an increasing system performance measured by the system throughput. However, mainly due to the demand for portability and a drastic reduction in silicon area cost, power consumption becomes an increasingly important issue. Several other factors, such as prolonged component life expectancy and demand for inexpensive packaging also provides strong motivation for power reduction. Most likely, in the near future the design goal will be a composite function which combines area and power consumption for many applications.

Recently, among the factors which dominantly influence power consumption besides technology (optimal supply voltage selection and MOS transistor minimum feature size scaling),

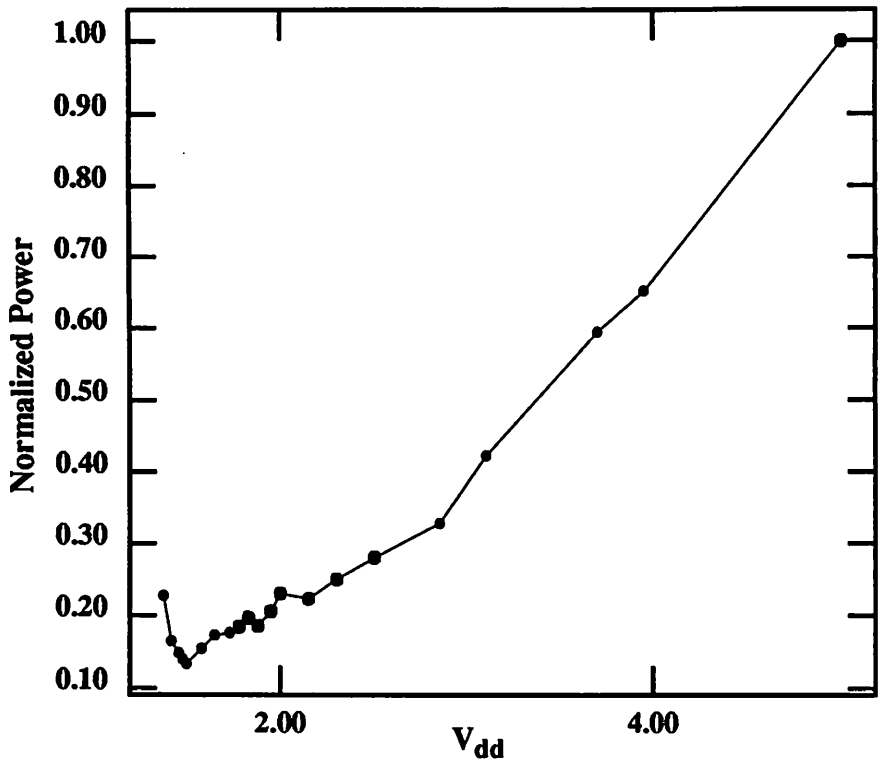


FIGURE 7.1 Plot of Power as a function of supply voltage for a fixed throughput for the 11th order IIR filter

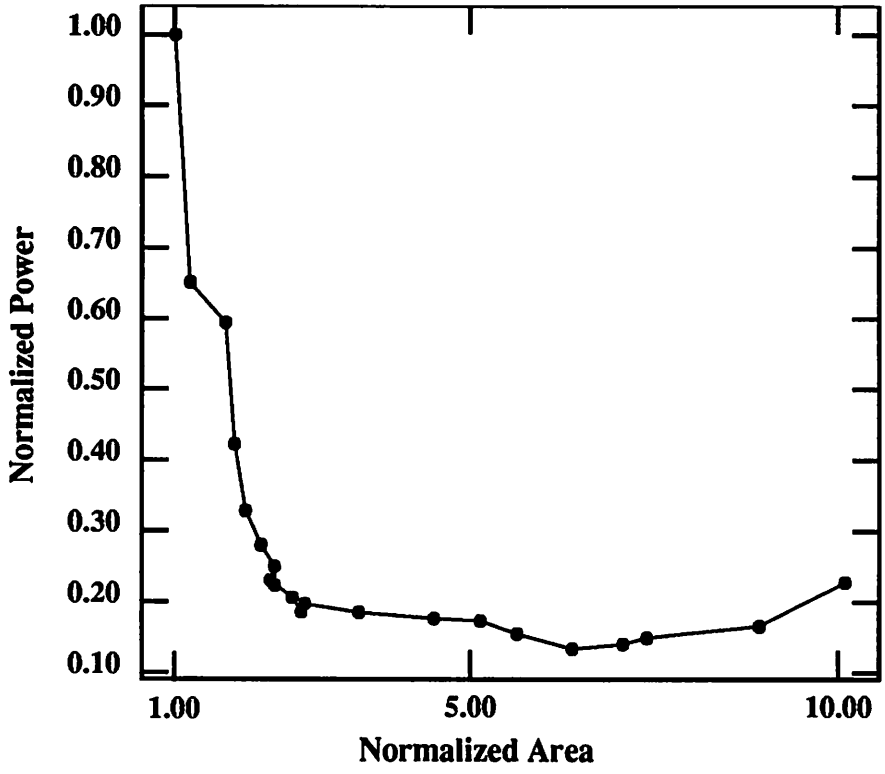


FIGURE 7.2 Plot of normalized power vs. implementation Area for a fixed throughput for the 11th order IIR filter

logic style selection and physical design issues, architectural design and algorithm selection were identified as particularly important [Bro91, Cha91a]. It has been shown that a trade-off relationship exists between power, area and throughput. This trade-off relationship can be optimized using high level synthesis tools. For example, Figure 1 shows the relationship between power and supply voltage for a fixed throughput. Figure 2 shows the relationship between the normalized power and implementation area for a fixed throughput. Both figures are for a 7th order IIR filter. The use of the low power was made possible due to the use of transformations.

In many cases the application of transformation on this example resulted in more than an order of magnitude reduction in power, as well as in a much larger number of design options [Cha91a].

Other interesting topics regarding the relationship between power optimization and high level synthesis include the use of estimation techniques for power prediction, module selection for power minimization, partitioning and algorithm selection and design for power reduction [Cha91b]. First studies show that the HYPER environment provides a suitable starting point for this research technique [Cha91b].

7.3.3 Structured Benchmarks for Scheduling and Assignment

One of the most difficult CAD questions is the assessment of the quality of a proposed algorithm and a corresponding program implementation. The majority of CAD optimization problems are NP-complete or even more complex. Therefore, the time for obtaining an exact solution even for problems of modest size is very long.

The most common procedure is to take a few (sometimes only one or two) examples and to conclude that the proposed algorithm produces a very good solution, due to the fact that it slightly outperforms previously published algorithms with respect to either speed or quality. This

approach does not guarantee that the very next example will be solved successfully. More profound approaches include comparison with sharp lower bounds and benchmarks.

Often the establishment of relatively large benchmark sets of examples from practice is considered the ultimate solution. However, in the case of high level synthesis problems, and particularly scheduling and assignment, there are serious problems associated with a such approach due to the "curse of dimensionality" [Bre84].

To test the scheduling and assignment algorithm it is necessary to use examples whose control data flow graphs have different parameters. Just to name few, we can mention control data flow graphs with few and many nodes, with large and small slack, with large and small available parallelism, highly structured and irregular, with a lot of and with little broadcasting, with few and a lot of timing constraints, for various relationships of hardware speed and cost. Just to have one example with a different value for each of the parameters will make the benchmark set very large.

A structured benchmark can be developed by using the following procedure. First, all available examples are plotted in the multidimensional space of various parameters and then among them the subset is selected in such a way that whole example space is as uniformly covered as possible. The goal is to choose those points which best represent all points in the multidimensional space. The number of text examples in benchmark sets is determined using a validation procedure (for example, cross validation or bootstrap) [Efr82]. These procedures also establish bounds which determine to what extent benchmark results can be trusted.

Probably the most important future application of structured benchmarks is in the design style selection, where they can be used for the prediction of the implementation cost of a given algorithm in different hardware and architectural configurations. This information can be used to speed up search through design space.

7.3.4 Design Style, Architecture and Algorithm Matching

An algorithm described as a program, can be implemented in various technologies (e.g., PLD, standard cells, gate arrays, full custom ASIC, standard off-shelf components) and various architectures (e.g., general purpose processors, DSP processors, systolic arrays, SIMD multiprocessors, superscalar and superpipelined architectures, custom ASIC). For a particular technology and architecture there are often several models from different suppliers. An objective function can be complex and diverse (e.g., combination of performance, implementation cost, power consumption, scalability). The partitioning of a program on suitable heterogeneous platforms so that the given objective function is optimized is often crucial for the final quality of the product.

To address this problem, two components have to be developed: a design framework and an efficient design space exploration mechanism. The framework should be capable to describe and manage hierarchal structures on both design sides (abstract algorithm description and final implementation) to provide an interface between different architectures and design styles, and to characterize precisely all design styles and architectures, so that a design space search can be conducted. The already developed HYPER framework provides a good starting basis for this task. Concerning the design space search, an inclusion of specific knowledge about the design space topology using deterministic and probabilistic techniques is a must for the further search performance improvements. Estimation and in particular the statistical methods provide a well suited tool for this task. To obtain this knowledge, a set of benchmarks has to be developed, which uniformly covers the whole design space. Special attention during system level design has to be paid to memory, input/output and interface organization, as well as to the integration with existing and developing research and commercial design systems, which address other important system design issues such as testing, simulation and lower level implementation.

7.3.5 Algorithm Design for Efficient Implementation

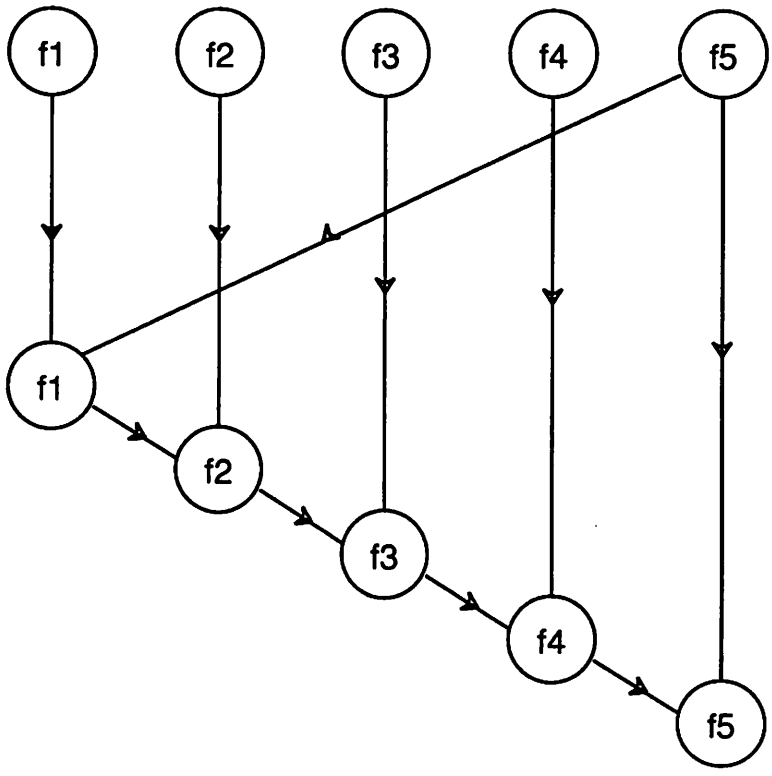
Transformations are an efficient and powerful tool for the enhancement and exploration of parallelism. It has been shown in Chapter 5 that they can have dramatic influence on the quality of implementation. However, their effect is obviously constrained by the algorithm's CDFG structure. For example, it is easy to construct examples where no transformation is applicable at all. When several algorithms are available for the same task, using the ideas outlined in the previous section we can select one which is most amenable for ASIC implementation. This situation is common when the algorithm is widely used, and a substantial effort has been done in the algorithm design community to provide an efficient algorithm for a particular task. For example, there exists a wide variety of fast transform algorithms (e.g., for Fourier and Discrete Cosine Transforms) [Bla85], sorting [Knu73], and in general for statistics [Thi85], numerical analysis algorithms [Dah74], linear system of equation solution [Gol90], integer programming [Nem88], linear programming [Chv83], and so on).

While the automated designing of general algorithms which can be efficiently transformed so that the final implementation requires few fully utilized hardware instances most likely will not be feasible in the near future, there are several ways to do efficient algorithm design for restricted domains. It appears that wide classes of applications use algorithms with a substantial freedom in algorithm design which is well suited for the exploration of parallelism.

Figure 3 and Figure 4 illustrate this point. Iterative methods have been used extensively to solve systems of equations. They have the structure

$$x(t+1) = f(x(t)), \quad t = 0, 1, \quad ...$$

where each $x(t)$ is an n-dimensional vector, and $f(.)$ is some function which has as a domain and a range some subset of n-dimensional space. If the sequence $\{x(t)\}$ generated by the above iteration converges to a limit point x^*, and if the function $f(.)$ is continuous, then x^* is a fixed point of



x1 = f1 (x1, x5)
x2 = f2 (x1, x2)
x3 = f3 (x2, x3)
x4 = f4 (x3, x4)
x5 = f5 (x4, x5)

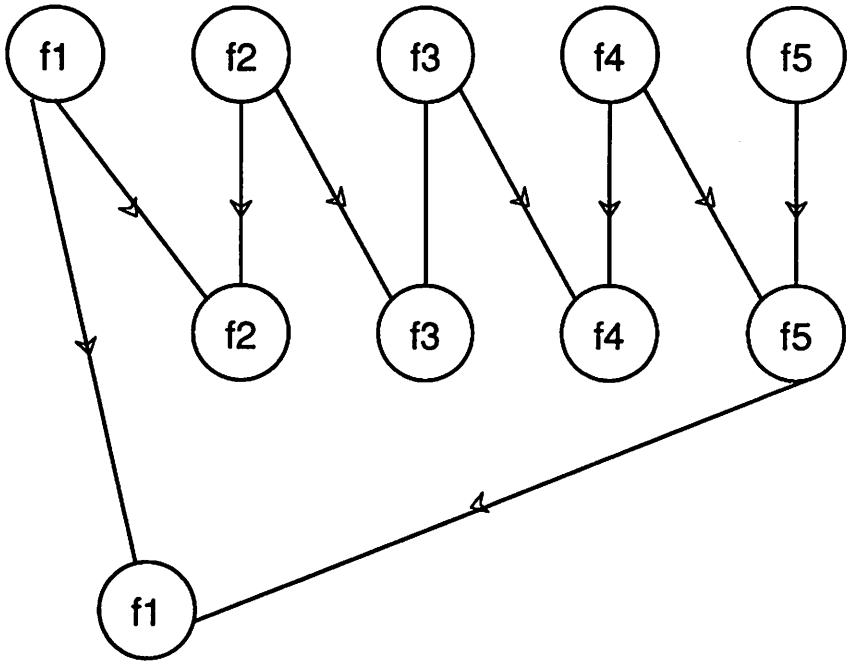
FIGURE 7.3 Parallelism of Gauss-Seidel iterations: The initial updating order

f, which satisfies the relationship $x^* = f(x^*)$. For example, iterative methods are often used for the solution of sparse systems of equations, or for the maximization and minimization of a function, by search for zeroes of the derivatives.

When all the components of x are updated simultaneously, iterations are often called Jacoby or Gauss-Jacoby types. An alternative approach is to update one equation at a time. Then, the previous equation can be expressed in the form:

$$x_i(i+1) = f_i(x_i(t+1), ..., x_{i-1}(t+1), x_i(t), ..., x_n(t)), \quad i = 1, ..., n$$

This type of iteration is often called the Gauss-Seidel iteration type. Since the Gauss-Seidel algorithm incorporates the most recent information at each step [Ort90], it often converges faster than Gauss-Jacoby iterations. It is easy to recognize that Gauss-Seidel algorithms are not well suited for pipelining, and are very amenable to the fast implementation of recursive program transformation. The usefulness of this transformation can be substantiated using the following observation.



x5 = f5 (x4, x5)
x4 = f4 (x3, x4)
x3 = f3 (x2, x3)
x2 = f2 (x1, x2)
x1 = f1 (x1, x5)

FIGURE 7.4 Parallelism of Gauss-Seidel iterations: The increase in parallelism and the reduction of the critical path due to the change in the updating order

In the Gauss-Seidel iterative algorithm, order of updating is not fixed. Instead of starting from x_1 and proceeding forward, we can permute the updating of indices. Of course, in this case

we have different algorithms. Nevertheless, for many systems of equations, a Gauss-Seidel algorithm converges in the limit of a large number of iterations to the same fixed point. Then, by analyzing the speed of convergence and the amount of parallelism we can construct the algorithm which is the best suited for VLSI implementation. For example, it is easy to see that the algorithm in Figure 4 has a significantly shorter critical path than the algorithm in Figure 3. For many instances the algorithm in Figure 4 will be superior. Of course, for the final conclusion it is necessary to develop tools to analyze the speed of convergence, and the parallelism of the algorithm. For the second task, it appears that extension of HYPER estimation tools is a good starting point. Finally, the strong potential for an additional performance improvement is in application of transformations during algorithm design.

Several other aspects in the Gauss-Seidel algorithm can be optimized also. For example, an additional point for optimization during algorithm design for iterative methods is to consider the effect of simultaneously updating more than one component of x.

In many numerically intensive computation areas similar algorithm design selection technique, where both the amount of computation and the parallelism are important criteria, can be used. They include algorithms for solution of ordinary differential equations (such as Runge-Kutta adaptive size step methods, the Bulirsh-Stoer method, or Predictor-Corrector Method), partial differential equation solvers and various minimization and maximization algorithms. One especially interesting class is the probabilistic optimization methods.

While this section had an accent on the algorithm design issues, which are not correlated and achievable using transformations, another promising direction, which also can lead to an automatic algorithm design, is further exploration of transformations. It can be shown that many algorithms in the signal processing area (such as DFT and FFT) can be derived using sophisticated application of algebraic transformations (associativity, distributivity and commutativity) and common subexpression elimination. Although this type of algorithm design is currently

done exclusively by leading researchers, it is plausible to expect that by combining efficient search techniques with powerful computers, the algorithm design of this type can be achieved automatically or with limited human interaction.

7.4 CONCLUSION

This thesis presented a new high level synthesis system, HYPER. HYPER uses a single, global quality measure, called the resource utilization measure, to drive the design space exploration process. This unique approach effectively merges the allocation, assignment, scheduling, transformations, and estimation with hierarchy handling in an unified manner.

New allocation, scheduling and assignment algorithms for hierarchical control data flow graphs based on a probabilistic rejectionless anti-voter technique were presented. A new transformation environment has been developed. Several new transformations (including retiming and associativity, software pipelining and software retiming, commutativity, and maximally fast implementation of recursive programs), based on a novel learning while searching technique, were introduced.

The effectiveness of the proposed algorithms is demonstrated in multiple ways: using standard benchmarks examples, with the aid of statistical analysis and through a comparison with estimated minimal bounds. Sharp minimal bounds based on a discrete relaxation technique are also used.

8

REFERENCES

[Aar87] E.H.L. Aarts, P.J.M. van Laarhoven: "Simulated Annealing: Theory and Applications", D. Reidel Publishing Company, Dordrecht, Holland, 1987.

[Aar89] E.H.L. Aarts, J. Korst: "Simulated Annealing and Boltzmann Machines, A Stochastic Approach to Combinatorial Optimization and Neural Computing", John Wiley & Sons Publishing, 1989.

[Ack87] D.H. Ackley: "A connectionist machine for the genetic hillclimbing", 1987.

[Ack85] D.H. Ackley, G.E. Hinton, T.J. Sejnowski: "A learning algorithm for Boltzmann machines", *Cognitive Science*, Vol. 9, pp. 147-169, 1985.

[Bag87] N. Bagherzadeh, T. Kerola, B. Leddy, R. Brice: "On parallel execution of the traveling salesman problem on a neural network model", *Proc. Int. Conf. on Neural Networks*, pp. 317-324, 1987.

[Bak90] H.B. Bakoglu: "Circuits, interconnections, and packaging for VLSI", Reading, Mass.: Addison-Wesley Pub. Co., 1990

[Bal89] M. Balakrishnan, P. Marwedel: "Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration", *Proc. 26th Design Automation Conference*, pp. 68-74., 1989.

[Ban79] U. Banerjee et al., "Time and Parallel Processor Bounds for Fortran-like Loops", *IEEE Trans. on Computers*, Vol. 28, No 12, pp 660-670, 1979.

[Ban88] U. Banerjee: "Dependence analysis for supercomputing", Kluwer, Boston, 1988.

[Bar88] E.R. Barnes, A. Vannelli, J.Q. Walker: A new heuristic for partitioning the nodes of a graph, *SIAM Journal of Discrete Mathematics*, Vol. 1, No. 3, pp. 299-305, 1988.

[Bar73] M.R. Barbacci: Automated Exploration of the Design Space for Register Transfer (RT) Systems", *Ph.D. Thesis*, Carnegie Mellon University, 1973.

[Bla85] R. E. Blahut, "Fast Algorithms for Digital Signal Processing", Addison-Wesley Publishing Company, 1985.

[Bla83] J. Blazewicz, J.K. Lenstra, A.H.G. Rinnooy Kan: "Scheduling subject to resource constraints: Classification and Complexity", *Disc. Applied Math.*, Vol. 5., No. 1, pp. 11-24., 1983. pp. 650-667, 1984.

[Bol85] B. Bollobas: "Random graphs", London; Orlando: Academic Press, 1985.

[Bra84] R.K. Brayton et al.: "Logic minimization algorithms for VLSI synthesis", Boston: Kluwer Academic Publishers, 1984.

[Bre74] R.P. Brent: "The Parallel Evaluation of General Arithmetic Expressions", *Journal of the ACM*, Vol. 21, No. 2, pp. 201-206, 1974.

[Bre84] L. Breiman, J. Friedman, R. Olshen, C. Stone: "Classification and regression tress", Wadsworth International Group, 1984.

[Bre90] F. Brewer, D.D. Gajski: "Chippe: A System for Constraint driven Behavioral Synthesis", *IEEE Trans. on CAD*, pp. 681-695,Vol. 9, No. 7, 1990.

[Bro91] R.W. Brodersen, A. Chandrakasan, S. Sheng: "Technologies for Personal Communications", *VLSI Circuits Symposium*, pp. 5-9, Japan, 1991.

[Bru87] J. Bruck, J.W. Goodman: "A Generalized convergence theorem for neural network networks and its application to combinatorial optimization", *Proc. Int. Conf. on Neural Networks*, pp. 649-656, 1987.

[Buc83] Y.I. Bucher, T. Jordan: "Polynomial conjugate gradient experiment on Cray X-MP with SSD as a user device", *Los Alamos report No. B265/7-7028*, 1983.

[Cam91] R. Camposano, R.A. Walker: "A Survey of high-level synthesis systems", Boston: Kluwer Academic, Norwell, Mass., 1991.

[Cat88] F. Catthoor, H. DeMan, J. Vanderwalle: "SAMURAI: a general and efficient simulated-annealing schedule with fully adaptive annealing parameters", *Integration*, Vol. 6, pp. 147-178, 1988.

[Cha68] H.R. Charney, D.L. Plato: "Efficient partitioning of components", *Proc. of the 5th Annual Design Automation Workshop*, pp. 16.0-16.21, 1968.

[Cha91a] A. Chandrakasan, M. Potkonjak, R.W. Brodersen, J. Rabaey: "Optimizing Power Using Transformations", *Paper in preparation*, 1991.

[Cha91b] A. Chandrakasan: Personal Communications, September 1991.

[Chu89] C. Chu, M. Potkonjak, M. Thaler, J. Rabaey: "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications", *Proc. IEEE ICCD Conf.*, pp. 432-435, Cambridge, MA, October 1989.

[Chu91] C.Chu, "Hardware Mapping and Module Selection in the HYPER Synthesis System", *PhD Thesis*, University of California, Berkeley, August 1991.

[Chv83] V. Chvatal: "Linear programming", New York: W.H. Freeman, 1983.

[Cob64] A. Cobham, "The intrinsic computational difficulty of functions", *Proc. 1964 International Congress for Logic Methodology and Philosophy of Science*, Y. Bar-Hillel, ed., North-Holland, Amsterdam, pp. 24-30, 1964.

[Coo71] S.A. Cook: "The complexity of theorem proving procedures", *Proc. Third ACM Symposium on Theory of Computing*, pp. 24-30, 1971.

[Cor90] T.H.Cormen, C.E.Leiserson and R.L.Rivest, "Introduction to algorithms", MIT Press, Cambridge, MA; McGraw-Hill, New York, 1990.

[Cul75] J. Cullum, W.E. Donath, P. Wolfe: "The minimization of certain nondifferentiable sums of eigenvalues of symmetric matrices", *Mathematical Programming Study*, Vol. 3, pp. 35-55, 1975.

[Dah74] Dahlquist, Germund and Bjorck, Ake. "Numerical Methods", Englewood Cliffs, NJ: Prentice-Hall, 1974.

[Dah87] E.D. Dahl: "Neural network algorithm for an NP-complete problem: map and graph coloring", *Proc. Int. Conf. on Neural Networks*, pp. 113-120, 1987.

[Dav83] M.Davio, J.-P.Deschamps and A.Thayse, "Digital Systems with Algorithm Implementation", John Whiley & Sons, 1983.

[Dev89] S. Devadas, A.R. Newton: "Algorithms for Hardware Allocation in Data Path Synthesis", *IEEE Transaction on CAD*, Vol 8, No 7, pp. 768-781, 1989.

[Don91] J.J. Dongarra et al.: Solving linear systems on vector and shared memory computers" Philadelphia: Society for Industrial and Applied Mathematics, 1991.

[Don88] W.E. Donath: "Logic Partitioning", in Preas, B., Lorenzetti, M.: "Physical Design Automation of VLSI Systems", pp. 65-86, 1988.

[Dye89] M.E. Dyer, A.M. Frieze" "The solution of Some Random NP-Hard Problems in Polynomial Expected Time", *Journal of Algorithms*, Vol. 10, pp. 451-489, 1989.

[Edm65] J. Edmonds: "Paths, trees and flowers", *Canad. J. Math.*, 17, pp. 449-467, 1965.

[Efr82] B. Efron: "The jackknife, the bootstrap, and other resampling plans", *SIAM*, 1982.

[Fel85] J.A. Feldman, D.H. Ballard: "Connectionist Models and Their Properties", *Cognitive Science*, Vol. 6, pp.205-254, 1985.

[Fet90] A. Fetweis, H. Meyr, L. Thiele: "Algorithm Transformations for Unlimited Parallelism", *IEEE International Symposium on Circuits and Systems*, pp. 1756-1759, New Orleans, 1990.

[Fid88] C.M. Fiducia, R.M Mattheyses: "A linear time Heuristic for Improving Network Partitions", *19th Design Automation Conference*, pp. 175-181, 1982.

[Gaj87] D. D. Gajski, ed.: "Silicon Compilation", Addison-Wesley, December 1987.

[Gar79] M.R. Garey, D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H.Freeman and company, New York, 1979.

[Gol80] M.C.Golumbic, "Algorithmic Graph Theory and Perfect Graphs", Academic Press, 1980.

[Gon77] M.J. Gonzalez, Jr. "Deterministic Processor Scheduling", *Computing Surveys*, Vol. 9, No. 3, pp. 173-204, September, 1977.

[Goo86] G. Goossens, R. Jain, J. Vandewalle, H. De Man, "An optimal and flexible delay management technique for VLSI" in: C.I. Byrnes, A. Lindquist, "Computation and Combinational methods in system theory", pp. 409-418, North Holland, 1986.

[Goo87] G. Goossens, J. Rabaey, J. Vandewalle, H. De Man: "An efficient microcode-compiler for custom DSP-processors", *IEEE CAD*, Santa Clara, pp. 24-27, 1987.

[Goo89] G. Goossens, J. Wandewalle, H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems", *26th Design Automation Conference*, pp. 826-831, Las Vegas, NV, 1989.

[Got86] S. Goto, T. Matsuda: Partitioning, Assignment and Placement, in Ohtsuki, T: Layout Design and Verification, pp. 55-97, 1986.

[Gre86] J.K. Greene, K.J. Supowit: "Simulated Annealing Without Rejected Moves", *IEEE Trans. on CAD*, Vol. 5., No. 1, pp. 221-228, 1986.

[Gri75] G.R. Grimmet, C.J.H. McDiarmid: "On colouring random graphs", *Math. Proc. Cambridge Phlos. Soc.*, Vol. 77, pp. 313-324, 1975.

[Gul87] S. Gulati, S.S. Iyengar, N. Toomaraian, V. Protopopescu, J. Bahren: "Nonlinear neural networks for deterministic scheduling", *Proc. Int. Conf. on Neural Networks 4*, San Diego, pp. 742-752, 1987.

[Gyr84] E. Gyrczyc: "Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Description", *PhD Thesis*, Carleton University, 1984.

[Han76] M. Hanan, O.K. Wolff, B. Agule: Some experimental Results on Placement Techniques", *12th Design Automation Conference*, pp 214-224, 1976.

[Han88] D.L.Hanson, "Interconnection Analysis", in "Physical Design Automation of Electronic Systems", edited by B.T.Preas and M.J.Lorenzetti, pp. 31-64, 1988.

[Har86] D. Harrison at al.: "Data Management and Graphics Editing in the Berkeley Design Environment", *Proc. Int. Conf. Computer Aided Design*, pp. 24-27, 1986.

[Har89] B.S. Haroun, M.I. Elmasry: "Architectural Synthesis for DSP Silicon Compilers", *IEEE Transaction on CAD*, Vol. 8, No. 4., pp. 431-447, 1989.

[Har89] R. Hartley, A. Casavant: "Tree-height Minimization in Pipelined Architectures", *IEEE CAD*, pp.112-115, 1989.

[Hei84] P. Hiedelberger and S. Lavenberg, "Computer Performance Evaluation Methodology", *IEEE Transactions on Computers*, Vol. 33, No. 12, pp. 1195-1220, 1984.

[Hen89] J.L. Henessy, D.A. Patterson: "Computer architecture: a quantitative approach", San Mateo, Calif.: Morgan Kaufman Publishers, 1989.

[Hil85] P. Hilfinger: "A High-level Language and Silicon Compiler for Digital Signal Processing", *Proc. Custom Integrated Circuits Conf.*, IEEE Computer Society Press, Los Alamitos, CA, pp. 213-216, 1985.

[Hin87] G.E. Hinton: "Connectionist Learning Procedures", Carnegie-Mellon University, *Technical Report CMU-CS-87-115*, 1987.

[Hoa92] P. Hoang: "A Compiler for Multiprocessor DSP Implementation", *PhD Thesis in preparation*, University of California, Berkeley, 1992.

[Hoc87] D. S. Hochbaum and D.B.Shmoys, "Using Dual Approximation Algorithms for Scheduling Problems: Theoretical & Practical Research ", *Journal of ACM*, Vol.34, No 1, pp.144-162, 1987.

[Hol75] J.H. Holland: "Adaptation in Natural and Artificial Systems", University of Michigan Press, Ann Arbor, 1975.

[Hop82] J.J. Hopfield: "Neural Networks and Physical Systems with emergent collective computational abilities", *Proceedings of the National Academy of Science*, USA, Vol. 79, 1982, pp. 3088-3092.

[Hwa91] C.-T. Hwang, J. -H. Lee, Y.-C. Hsu:"A Formal Approach to the scheduling problem in high level synthesis", *IEEE Trans. on CAD*, Vol. 10, No. 4, pp. 464-475, 1991.

[Hya73] L. Hyafil, R.L. Rivest.: "Graph partitioning and constructing optimal decision trees are polynomial complete problems", *Report No.33*, IRIA-Laboria, Rocqquencourt, France, 1973.

[Iba88] T. Ibaraki, N. Katoh: "Resource Allocation Problems", The MIT Press, Cambridge, MA, 1988.

[Jai88] R. Jain et al, "Module Selection for Pipelined Synthesis", *Proc. Design Automation Conference*, Anaheim, pp. 542-547, 1988.

[Jai89] R. Jain, "High-Level Area-Delay Prediction with Application to Behavioral Synthesis", *Technical Report 89-23*, University of Southern California, 1989.

[Jai91] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for experimental design, measurement, simulation, and modeling", Wiley, 1991.

[Joh83] D.S. Johnson: "The NP-Completeness Column: An Ongoing Guide", *Journal of Algorithms*, Vol. 4, No. 1, pp. 189-203, 1983.

[Joh8x] D.S. Johnson: "The NP-completeness column: an ongoing guide", *Journal of Algorithms*, 1981-1989.

[Joh88] D.S. Johnson, C.H. Papadimitrou, M. Yannakakis: "How Easy is Local Search?", *Journal of Computer and System Sciences*, Vol. 37, No. 4, pp. 79-100, 1988.

[Joh89] D.S.Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon: "Optimization by simulated annealing: An Experimental Evaluation: Part I, Graph Partitioning", *Operations Research*, Vol. 38, No. 6, pp. 865-892, 1989.

[Joh91] D.S. Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon: "Optimization by simulated annealing: An Experimental Evaluation: Part II, Graph Coloring and Number Partitioning", *Operations Research*, Vol. 39, No. 3, pp. 378-406, 1991.

[Jou89] N. Jouppi and D. Wall, "Available Instruction-Level Parallelism for Super-Scalar and Super-Pipelined Machines", *Proc. 3d International Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, pp.272-282, May 1989.

[Kar72] R.M. Karp: "Reducibility among combinatorial problems", in Miller,R.E.: "Complexity of Computer Computations", Plenum Press, 1972, pp. 85-103.

[Kar88] R.M. Karp: "Lecture Notes for 292f", University of California, Berkeley, Spring 1988.

[Ker70] B.W. Kernighan, S. Lin: "An efficient heuristic procedure for partitioning graphs", *The Bell System Technical Journal*, Vol. 49, pp. 291-307, 1970.

[Kir83] S. Kirkpatrick, C.D. Gellat, Jr., M.P. Vecchi: "Optimization by simulated annealing", *Science*, Vol. 220, No. 4598, pp. 671-680, 1983.

[Knu73] D.E. Knuth: "Sorting and Searching" vol. 3 of "The Art of Computer Programming", Reading, Mass: Addison-Wesley, 1973.

[Kod72] U.R. Kodres: Partitioning and Card Selection, Breuer, M.A.: Design Automation of Digital Systems, pp. 173-212, 1972.

[Kog81] P.M. Kogge: "The architecture of pipelined computers" Washington: Hemisphere Pub. Corp.; New York: McGraw-Hill, 1981.

[Kos91] B. Kosko: "Neural networks and fuzzy systems: a dynamical systems approach to machine intelligence", Englewood Cliffs, NJ: Prentice Hall, 1991.

[Kuc72] D. Kuck, Y. Muraoka, S. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and their Resulting Speed-up", *IEEE Trans. On Computers*, Vol. 21, No 12, pp. 1293-1310, 1972.

[Kuc90] K.Kucukcakar and A.C.Parker, "BAD: Behavioral Area-Delay Predictor", *Tech Report 90-31*, University of Southern California, 1990.

[Kuc91] K.Kucukcakar and A.C.Parker, "CHOP: A Constraint-Driven System-Level Partitioner", *28th ACM/ IEEE DAC*, San Francisco, pp. 514-519, 1991.

[Kur87] F.J. Kurdahi: "Area Estimation of VLSI Circuits", *PhD thesis*, University of Southern California, 1987.

[Kur89] F. Kurdahi and A. Parker, "Technique for Area Estimation of VLSI Layouts", *IEEE Trans. On CAD*, Vol. 9, No 9, pp. 938-950, 1990.

[L'E88] P. L'Ecuyer: "Efficient and Portable Random Generators", *Communication of the ACM*, Vol. 31, No. 6 pp. 742-751, 1988.

[Lam88] M.S. Lam: "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", *ACM SIGPLAN*, 1988.

[Lam89] M. S. Lam: "A systolic array optimizing compiler", Boston: Kluwer Academic; Norwell, Mass., 1989.

[Law69] E. Lawler, K.N. Levitt, J. Turner: Module clustering to minimize delay in digital networks", *IEEE Trans. on Computers*, Vol. 18, No. 1, pp. 47-57, 1969.

[Law76] E. Lawler: "Combinatorial Optimization: networks and matroids", Holt, Rinehart and Winston, 1976.

[Law85] E. Lawler: "The Traveling salesman problem: a guided tour of combinatorial optimization", Chichester [West Sussex]; New York: Wiley, 1985.

[Law90] E. Lawler, J. Lenstra, A. Rinnooy Kan, D. Shmoys: "Sequencing and Scheduling: Algorithms and Complexity" in S. Graves, A. Rinnooy Kan, P. Zipkin (editors): Handbooks in Operational Research and Management Science, Vol. 4: Logistics of Production and Inventory. North-Holland Publishing Co., New York, 1990.

[Lei83] C.E. Leiserson, F.M. Rose, J.B. Saxe, "Optimizing synchronous circuits by retiming", *Proceedings of the Third Conference on VLSI*, pp. 23-36, Computer Science Press, 1983.

[Lig85] T.M. Liggett: "Interacting particle systems", New York: Springer-Verlag, 1985.

[Lin91] H.-P. Lin, D.G. Messerschmitt: "Finite State Machine has Unlimited Concurrency", *IEEE Trans. on Circuits and Systems*, Vol. 38, No. 5, pp. 465-475, 1991.

[Luc69] F. Luccio, M. Sami: On the Decomposition of Networks to Minimize Delay in Digital Networks, *IEEE Trans. on Circuits Theory*, Vol. 16, No. 2, pp. 141-148, 1969.

[Mal90] S. Malik, E. Sentovich, R.K. Brayton, A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: "Optimizing Sequential Networks with Combinational Techniques", *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Science*, vol I, pp. 397-406, 1990.

[Mal91] S. Malik, E.M. Sentovich, R.K. Brayton, A. Sangiovanni- Vincentelli: "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Technique", *IEEE Trans. on CAD*, Vol. 10, No. 1, pp. 74-84, 1991.

[Mat91] V.J. Mathews: "Adaptive Polynomial Filters", *IEEE Signal Processing Magazine*, Vol. 8. No. 3, pp. 10-26, July 1991.

[McF86] M.C. McFarland: "Using Bottom-Up Design Technique in the Synthesis of Digital Hardware from Abstract Description", *23rd Design Automation Conference*, Las Vegas, NV, pp. 474-480, 1986.

[McF87] M.C. McFarland: "Reevaluating Design Space for Register-Transfer Hardware Synthesis", *IEEE ICCAD*, pp. 262-265, 1987.

[McF90a] M.C. McFarland, A.C. Parker, R. Camposano: "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 301-317, February 1990.

[McF90b] M.C. McFarland, T.J. Kowalski: "Incorporating Bottom-Up Design into Hardware Synthesis", *IEEE Trans. on CAD*, pp. 938-950, Vol. 9. No. 9, 1990.

[Men74] A. Mennone, R.L. Russo: An example computer logic graph and its partitioning and mappings", *IEEE Trans. on Computers*, Vol. 23, No. 12, 1974.

[Mes88] D. Messerschmitt, "Breaking The Recursive Bottleneck", in Performance Limits in Communication Theory and Practice, Kluwer Academic Publishers, 1988.

[Mil87] G.L. Miller, S. Teng: "Dynamic parallel complexity of computational circuits", *Proc. 19th Ann. ACM Symp. on Theory of Computing*, pp. 254-263, 1987.

[Mil88] G.L.Miller, V. Ramachandran, E. Kaltofen: "Efficient Parallel Evaluation of Straight-Line Code and Arithmetic Circuits," *SIAM Journal on Computing*, Vol. 17, No 4, pp. 687-695, 1988.

[Mjo89] E. Mjolsness, D. Sharp, B. Alpert: "Scaling, Machine Learning and Genetic Neural Networks", *Advances in Applied Mathematics*, Vol 10, No. 2, 1989.

[Mli91] M.J.Mlinar, "Control Path/Data Path Trade-offs in VLSI Design", *Technical Report 91-16*, University of Southern California, 1991.

[Mou74] J. Moussouris: "Gibbs and Markov random systems with constraints", *Journal of Statistical Physics*, Vol 10, pp. 11-33, 1974.

[Nem88] G.L. Nemhauser, L.A. Wolsey: Integer and combinatorial optimization, New York : Wiley, 1988.

[Nic84] A. Nicolau and J. Fischer, "Measuring the Parallelism Available for Very Long Instruction Word Architecture", *IEEE Trans. On Computers*, Vol. 33, No. 11, pp. 968-976, 1984.

[Nic91] A. Nicolau, R. Potasman: "Incremental Tree Height Reduction for High Level Synthesis", *28th ACM/ IEEE Design Automation Conference*, pp. 770-774, 1991.

[Ort90] J.M. Ortega: "Numerical analysis: a second course", Philadelphia: Society for Industrial and Applied Mathematics, 1990.

[Pap82] C. Papadimitriou and K. Steiglitz, "Combinatorial Optimization: Algorithms and Complexity", Prentice Hall, 1982.

[Par86] A.C. Parker, M. Mlinar, J. Pizarro: "MAHA: A program for data path synthesis", *Proc. 22nd Design Automation Conference*, pp. 461-466, 1986.

[Par88a] K. Parhi, "Algorithm and architecture design for high speed digital signal processing", *PhD Dissertation*, University of California, 1988.

[Par88b] N. Park, A.C. Parker: "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. on CAD*, Vol 7, No. 3, pp. 356-370, 1988.

[Par88c] N. Park, A.C. Parker: "Theory of Clocking for Maximum Execution Overlap of High-speed Digital Systems", *IEEE Trans. on Computers*, Vol. 37, No. 6, pp. 678-690, 1988.

[Par88d] S.K. Park, K.W. Miller: "Random Number Generators: Good Ones are Hard to Find", *Communications of the ACM*, Vol. 31, No. 10, pp. 1191-1201, 1988.

[Par89a] K.K. Parhi, D.G. Messerschmitt: "Pipeline interleaving and parallelism in recursive digital filters - I: Pipelining using scattered look-ahead and decomposition", *IEEE T-ASSP*, pp. 1099-1117, July 1989.

[Par89b] K.K. Parhi, D.G. Messerschmitt: "Pipeline interleaving and parallelism in recursive digital filters - II: Pipelined incremental block filtering", *IEEE T-ASSP*, pp. 1118-1134, July 1989.

[Pat89] D.A. Patterson, J.L. Henessy: "Computer architecture: a quantitative approach", San Mateo, Calif. : Morgan Kaufman Publishers, 1989.

[Pau89] P.G. Paulin, J.P. Knight: "Force -Directed Scheduling for the Behavioral Synthesis of ASIC", *IEEE Transaction on CAD*, Vol 8. No 6, pp. 661-679, 1989.

[Ped89] M. Pedram and B. Preas, "Accurate Prediction of Physical Design Characteristic for Random Logic", 1989 *IEEE ICCD Conf.*, Boston, pp. 100-108, 1989.

[Pet87] C. Peterson, J.R. Anderson: "A Mean Field Learning Algorithm for Neural Networks", *Complex Systems*, Vol. 1, No. 5, pp. 995-1019, 1987.

[Pet88] C. Peterson, J. R. Anderson: "Neural Networks and NP-complete Optimization Problems, A Performance Study on the Graph Bisection Problem", *Complex Systems*, Vol. 2, No. 1,1988.

[Pet89] A.D. Petford, D.J.A. Welsh: "A Randomised 3-colouring Algorithm", *Discrete Mathematics*, Vol 74, pp. 253-261, 1989.

[Pot89a] M. Potkonjak, J. Rabaey, "A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs", *26th ACM/IEEE Design Automation Conference*, June 1989.

[Pot89b] M. Potkonjak: "Partitioning of VLSI circuits", EECS 290h, Class Project, Spring 1989.

[Pot89c] M. Potkonjak: "Neural Networks", Physics 250, Class Project, Spring 1989.

[Pot90] M. Potkonjak and J. Rabaey, "Retiming for Scheduling", *VLSI Signal Processing Workshop*, pp. 23-32, San Diego, Nov. 1990.

[Pot91a] M. Potkonjak and J. Rabaey, "Optimizing the Resource Utilization Using Transformations", *Proc. IEEE ICCAD Conference*, Santa Clara, November 1991.

[Pot91b] M. Potkonjak and J. Rabaey, "Algorithms for Hierarchical Data Control Flow Graphs", to be published in the Special Issue on "Fundamental Methods in CAD" of the *International Journal on Circuit Theory and Applications*, 1991.

[Pow90] S.R. Powell, P.M. Chau: "Estimating Power Dissipation of VLSI Signal Processing Chips: The PFA Technique", in "VLSI Signal Processing IV" edited by H.S. Moscovitz, K. Yao, R. Jain, IEEE Press, pp. 250-259, 1990.

[Pre88] W.H. Press, B.P Flannery, S.A. Teukolsky, W. T. Vetterling: "Numerical Recipes in C, The Art of Scientific Computing", Cambridge University Press, Cambridge, 1988.

[Rab90] J. Rabaey, and M. Potkonjak, "Resource Driven Synthesis in the HYPER system," *ISCAS-90*, vol. 4, pp. 2592-2595, New Orleans, LA, May 1990.

[Rab91a] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak: "Fast Prototyping of Data Path Intensive Architecture", *IEEE Design and Test*, Vol. 8, No. 2, pp. 40-51, 1991.

[Rab91b] J. Rabaey: Personal Communications, July 1991.

[Ram88] V. Ramachandran: "Fast parallel algorithms for reducible flow graphs", in S.K. Tewksbury, B.W. Dickinson and S.C. Schwartz, eds: Concurrent Computations: Algorithms, Architecture, and Technology, Plenum, New York, pp. 117-138, 1988.

[Res86] M.L Resnick: SPARTA: A System Partitioning Aid, *IEEE Trans. on CAD*, Vol. CAD-5, No. 4, pp. 490-498, 1986.

[Rum86] D. E. Rumelhart, J.L. McClelland: "Parallel Distributed Processing", The MIT Press, Cambridge, 1986.

[Rus71] R.L. Russo, P.H Oden, P.K. Wolff, Sr.: A Heuristic procedure for the partitioning and mapping of computer logic graphs, *IEEE Trans. on Computers*, Vol 20., No. 12, 1455-1462, 1971.

[San87] A. Sangiovanni-Vincentelli: Automatic Layout of Integrated Circuits, in G. De Micheli, A. Sangiovanni-Vincentelli, P. Anthognetti: Design Systems for VLSI Circuits, pp. 113-197, 1987.

[Sas85] S.Sastry and A.C. Parker, "Stochastic Models for Wirability Analysis of Gate Arrays", *IEEE Transactions on CAD*, Vol.5, No 1, pp. 52-65, 1985.

[Shi89] H. Shin, N.S. Woo: "A cost based optimization technique for scheduling in data path synthesis", *IEEE Conference on Computer Design*, pp. 424-427, Cambridge, MA, October 1989.

[Shu91] C. Shung et al.,"An Integrated CAD System for Algorithmic Specific IC Design", *IEEE Journal on Computer Aided Design*, Vol. 10, No 4, pp 447-463, April 1991.

[Smi89] M. Smith, M. Johnson and M. Horowitz, "Limits on Multiple Instruction Issues", *Proc. 3d International Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, pp.290-302, May 1989.

[Sim81] B. Simons, "On Scheduling with Release Times and Deadlines", in Deterministic and Stochastic Scheduling, M. Demster et all, editors, D. Reidel, Dordrecht, pp. 75-88, 1981.

[Spr90] D.L. Springer, D.E. Thomas: "Exploiting the Special Structure of Conflict and Comparability Graphs in High-Level Synthesis", *IEEE ICCAD Conf.*, pp. 254-257, 1990.

[Sto89] L. Stock, R. van der Born: "EASY: Multiprocessor Architecture Optimisation", in G. Saucier, P.M. McLellan: Proc. of Int. Workshop on Logic and Architecture Synthesis for Silicon Compilers, May 88", North Holland, pp. 313-328, 1989.

[Sto90] T. Stoelzle et all, "A Flexible VLSI 60,000 Word Real Time Continuous Speech Recognition System", *Proc. IEEE Workshop on VLSI Signal Processing*, pp. 247-284, November 1990.

[Tag87] G.E. Tagliarini, E.E. Page: "Solving constrain satisfaction problems with neural networks", *Proc. Int. Conf. on Neural Networks*, pp. 741-748, 1987.

[Thi88] R.A. Thisted: "Elements of statistical computing", Chapman, 1988.

[Tri87] H. Trickey: "Flamel: A high-Level Hardware Compiler", *IEEE Trans. on CAD*, Vol. 6, No. 2, pp. 259-269, 1987.

[Tse86] C. Tseng, D.P. Siewiorek: "Automated synthesis of data paths in digital systems", *IEEE Trans. on CAD*, vol 5., No 3., pp. 379-395, 1986.

[Tur88] J.S. Turner: "Almost All k-Colorable Graphs Are Easy to Color", *Journal of Algorithms*, Vol. 9, No. 1, pp.63-82, 1988.

[Val83] L.G. Valiant, S. Skyum, S. Berkowitz, C. Rackoff: "Fast Parallel Computation of Polynomials Using Few Processes", *SIAM Journal on Computing*, Vol. 12, No 4, pp. 641-644, 1983.

[Ull84] J. Ullman, "Computational Aspects of VLSI", Computer Science Press, Rockville, Maryland, 1984.

[Van50] B.L. Van der Warden: "Modern Algebra", Frederick Ungar, New York, 1950.

[Vet86] M. Vetterli and A. Lichtenberg, "A Discrete Fourier-Cosine Transform Chip", *IEEE Journal on Selected Areas in Communications*, Vol. SAC-4, No. 1, pp. 49-61, Jan. 1986.

[Wai84] W.M. Waite, G.Goos: Compiler construction, New York: Springer-Verlag, 1984.

[Wal89] R.A. Walker, D.E. Thomas: "Behavioral Transformation for Algorithmic Level IC Design" *IEEE Trans. on CAD*, Vol 8. No.10, pp. 1115-1127, 1989.

[Wel84] D.J.A. Welsh, "Correlated percolation and repulsive particle systems, in: P. Tautu, "Stochastic Spatial Processes", Springer Lecture Notes 1212, pp. 300-311, 1984.