High-Level Synthesis Of Memory-Intensive Application-Specific Systems

Pravil Gupta

CENG Technical Report 94-20

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

August 1994

HIGH-LEVEL SYNTHESIS OF MEMORY-INTENSIVE APPLICATION-SPECIFIC SYSTEMS

by

Pravil Gupta

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

August 1994

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

Pravil Gupta

under the direction of his..... Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of

DOCTOR OF PHILOSOPHY

Dean of Graduate Studies

Date July 14, 1994

DISSERTATION COMMITTEE

Chairperson

Dedication

To my parents.

Acknowledgements

I take this opportunity to express my gratitude to several people who have made this thesis possible. I am grateful to my advisor Prof. Alice Parker for her constant guidance and inspiration during my dissertation work. She has been a great source of encouragement and support. Her numerous comments and suggestions made this thesis possible. I also wish to thank Profs. Melvin Breuer and Ken Goldberg for being on my dissertation and guidance committee, and Profs. Michel Dubois and Sandeep Gupta for being on my guidance committee.

I would like to thank my colleagues Chih-Tung Chen, Diogenes Silva, Shiv Prakash, Atul Ahuja, J.C. Batista-Desouza, Dong-Hyun Heo, Jen-Pin Weng, Kayhan Kucukcakar, Yung-Hua Hung for their friendship and help. I benefited greatly from interacting with them.

Other friends who made my years at USC pleasant are Rajgopal Srinivasan, Ishwar Parulakar, Deb Mukherjee, and Sridhar Narayanan.

My parents have been a continuous source of love, support and sacrifice. I dedicate this thesis to them. My brothers and sister were always encouraging. I especially thank my wife Pratibha for her patience through these years. Her love always provided me comfort.

This research was supported by Semiconductor Research Corporation (Contract No. 89-DJ-075), the Advanced Research Projects Agency (Contract No. JFBI90092) and National Science Foundation (Contract No. GER-9023979). I would like to thank these organizations for their support.

Contents

List Of Figures

List Of Tables

Abstract

The thesis addresses high-level synthesis of memory-intensive application-specific systems, with emphasis on hierarchical storage architecture design. These systems are commonplace in real-time applications where they demand high performance datapaths along with efficient storage schemes to support them. SMASH (Synthesis of Memory intensive Application-Specific Hardware) is a program which combines storage hierarchy design with datapath synthesis for a given behavioral specification with constraints on cost and performance.

SMASH includes the following major tasks: datapath synthesis, which includes operation scheduling combined with I/O accesses by the datpath from on-chip I/O buffers while looking ahead to evaluate storage architecture tradeoffs; and storage hierarchy design, which includes determining the data transfers between different levels of memory hierarchy. The synthesis techniques are based on (*i*) feasibility analysis of the input/output access during datapath scheduling using global system parameters like the memory bandwidth and I/O timing constraints, (*ii*) tradeoffs in storage structure, and (*iii*) cost-delay estimations for both functional and storage structures.

Experimental results are presented which validate our techniques and demonstrate the existence of a cost-performance trade-off in the storage architecture. In addition, a system level synthesis experiment illustrates the way SMASH can be integrated into a system-level synthesis environment. The experiment involved rapid prototyping of a JPEG image compression system, where SMASH was invoked both in the initial phase of the design flow to obtain the area-delay trade-off curves for different components of the system; and in the final phase to synthesize the selected design points.

Chapter 1

Introduction

There has been an exponential growth in microelectronics in the last couple of decades. Electronic systems which are mostly application-specific special-purpose hardware are becoming increasingly complex. Furthermore, the intense competition in the electronic industry is forcing companies to ensure a very early marketing time for these extremely complex, but reliable systems at a competitive price. Achieving these goals is the motivation behind *design automation*. Designers are relying more and more on CAD tools for assistance during the design process. CAD tools are being successfully used for physical design, logic synthesis, simulation, and other activities. However, system-level design still remains an art. Most system-level decisions are still being made by human designers. The correctness of these decisions completely depends on the knowledge and experience of the designer. Furthermore, these manual designs are so time consuming that there is virtually no exploration of the design space. Therefore, there is a growing need for system-level design tools. The research being addressed here deals with development of such system-level synthesis tools.

1.1 High-Level Synthesis

Computer-aided design is being successfully used in automatic synthesis of application specific ICs (ASICs) from algorithmic descriptions. Such automatic synthesis of register-transfer level (RTL) designs from a given algorithmic behavioral specification of a digital system, while satisfying a set of constraints and best meeting a design goal, is called *high-level synthesis* [MMC88].

In the above definition, the RTL datapath consists of a network of functional units (e.g. adder, multiplier), storage units (registers, register files), interconnection units (multiplexers, tri-state drivers), and buses. The behavioral specification reflects the mapping from inputs to outputs. Constraints specify design parameters (such as performance, area, or power consumption of the chip); a design goal could be to minimize one (or more) of these design parameters. For example, the design goal could be to minimize total power consumption of the chip while achieving a certain area and performance.

The ADAM (Advanced Design Automation) system at the University of Southern California deals with datapath and control synthesis [JKMP89, GKP85]. The inputs to ADAM are

- a behavioral specification either in VHDL or in form of a control data flow graph,
- a module library of available hardware operators (e.g., adders, multipliers) with their cost/delay characteristics, and
- a set of area/delay constraints and goals to be satisfied.

The outputs from ADAM are

- an RTL netlist consisting of components specified in the module library, and
- a description of a finite state machine as the controller for the generated RTL structure.

Though the realization of RTL structure from the behavior appears straightforward, the goal of high-level synthesis is to explore the design space and select the design that best meets the design constraints while achieving the design goals, from a number of RTL structures that can realize the given behavior. The RTL netlist is then processed by a silicon compiler to produce an IC.

Encouraged by the success in high-level synthesis, researchers are now looking into automation of the design process at the *system level*, which is one-step higher in the design process. The next section briefly describes the system-level synthesis problem and the issues at that level.

1.2 System-Level Synthesis

System-level synthesis, as a general problem, is such a complex problem that researchers are still struggling with the formulation of the problem description. The concern at this level is with system-level components and issues, and with the overall design methodology.

1.2.1 System-level components and issues

System-level components include processors (such as general-purpose microprocessors, special-purpose processors, and ASICs), memories (RAMs, ROMs, caches), bus interfaces, and I/O devices. Which components should be considered in a particular design depends heavily on the application under consideration and the designer as he/she may wish to design the system with a particular set of components. System-level issues consist of which and how many system-level components to include in the design, how to configure and interconnect them in the system, and how to control them. Again, which issues should be considered depends on the application and the designer.

1.2.2 Design methodology

The biggest challenge in system-level synthesis is deciding on a design methodology. The following three design methodologies are of potential use in a system-design environment:

1. *Top-down design flow*: in this methodology, system-level tools start with the system specification, and generate specifications and constraints for individual components. Then, the individual components are designed using high-level synthesis tools. The drawback in this approach is that since the design parameters of the individual components are not yet known, system level tools have to rely heavily on predictors.

2. *Bottom-up design flow*: here, individual components are designed first using high-level synthesis tools. Then, the whole system is designed with these components using system-level tools. The drawback in this approach is that

the design space exploration at the system level is very restricted as the system-level tools have to work with the pre-designed components.

3. *Mixed design flow*: This methodology is a combination of the above two approaches. For example, first some representative implementations of individual components are synthesized by the high-level synthesis tools to obtain the cost-performance tradeoff curve for these components (a bottom-up step) Then, based on the tradeoff curve, the exact specification and constraints for individual components are generated using the system-level tools. Finally, the individual components are synthesized (a top-down step) for these specifications and constraints using high-level synthesis tools.

As we can see in the methodologies described above, *high-level synthesis tools are at the core of system-level tools*. Therefore, it is essential that these high-level synthesis tools are able to deal with the system-level information. In the top-down approach, they should be able to handle system-related parameters specified by the system-level tools; similarly, in the bottom-up approach, they should be able to provide system-related parameters to the system-level tools; and in the mixed approach both. Even though, there has been significant progress in the field of high-level synthesis during the last decade and researchers have developed powerful techniques to synthesize designs from behavioral specifications, these techniques are constrained in their scope and focus mainly on the synthesis of datapaths alone. There has been very little research to address the synthesis of storage architectures in the designs. It is essential for high-level synthesis to become practical by addressing more issues during synthesis and broadening the scope. The Unified System Construction (USC) project at the University of Southern California is one such effort in this direction.

1.3 The USC Project

The Unified System Construction (USC) project involves the development of an integrated suite of system-level tools for synthesizing multi-chip, heterogeneous application specific systems which meet cost, performance and power constraints [PCG93]. The focus of the USC project is on real-time systems, such as entertainment and

communication technologies, but does not exclude other applications requiring specialized system design.

The major tasks that are accomplished using USC tools are

- partitioning, selection of components and packaging styles, and scheduling and allocation of components to meet cost and performance constraints.
- automatic synthesis of memory-intensive architectures, including both on and off chip memory, in conjunction with design of the processing units.
- exploration of system-level design tradeoffs by means of predictors, prior to synthesis, including power and thermal effects.

A block diagram of the system is shown in Figure 1.1. All the parts of USC project are not relevant to this thesis and will not be discussed here. This thesis addresses the tools and techniques to support automation of the design of memory-intensive architectures, including both on-chip and off-chip memory, in conjunction with design of the processing units. The techniques developed on this subject have been incorporated in a tool called SMASH (Synthesis of Memory-Intensive Application-Specific Hardware).

1.4 Memory-Intensive Systems

Memory-intensive application-specific systems are commonplace in video signal processing and real-time applications. These systems consist of four basic subsystems: datapath, controller, memory and I/O architectures. They demand high performance datapaths along with efficient storage schemes to support them. Datapaths for application-specific designs may process enormous amounts of real-time data. Such data must be stored in structures which are cost-effective and allow access to the data as required by the datapaths. With the declining cost of hardware, memories increasingly dominate the cost of digital systems. Although cost of storage per bit is very low, the total cost of memory may dominate the overall system cost due to the huge storage requirements of today's complex systems.

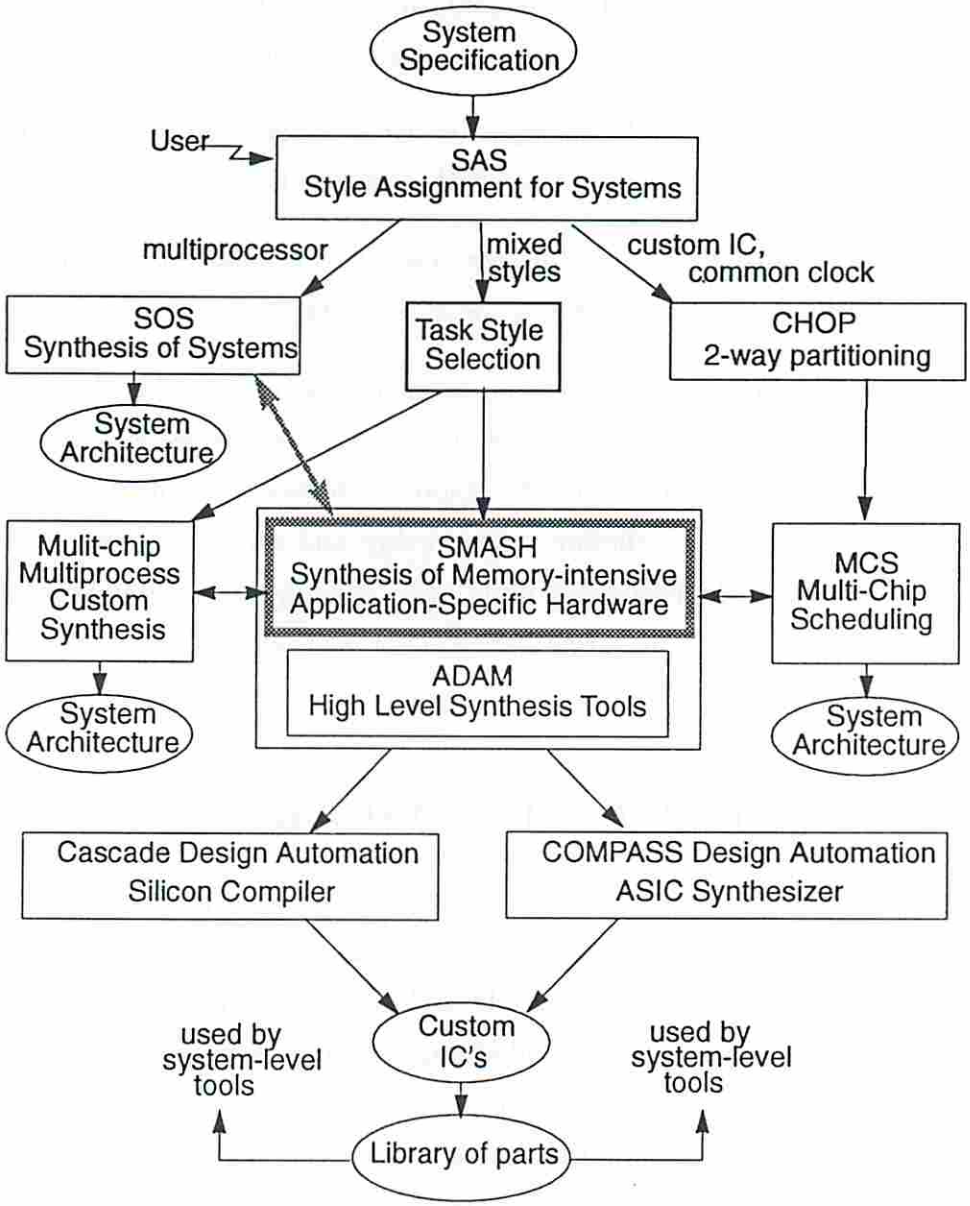


Figure 1.1: Block diagram of the Unified System Construction (USC) Project

1.5 Motivation

The storage architecture is closely connected to the datapath of the system, and isolating its synthesis from datapath synthesis may not result in an efficient solution. Datapath synthesis procedures themselves must take into account the design of the memory hierarchy which is companion to the datapath; ultimately the design of the datapaths and memory hierarchies must somehow be coordinated. Therefore, we aim for a combined datapath and storage architecture design. In addition, datapaths and memory interact with the controller and with the external world, and require interfaces to both. In order to keep the problem complexity within limits, we focus more on the issues relevant to storage architecture synthesis.

Our topic was motivated by a study of a digital video processing example, which indicated that memory design is as important and difficult as datapath design. Datapaths for application-specific designs may process enormous amounts of real-time data. Such data must be stored in storage structures which are cost-effective and allow access to the data as required by the datapaths. With the declining cost of hardware, memories increasingly dominate the cost of digital systems.

The storage architecture synthesis problem is also important in applications where the data transfer rate is very high (e.g. real-time applications such as personal communications). Here the major design issue is how and where to store the data and then how to distribute it efficiently. In such cases, depending on the processing speed requirements and cost limitations of different systems, a variety of strategies is needed to handle data.

Furthermore, our layout studies made it clear that datapath synthesis must consider physical design effects along with the design of storage architecture and other system modules in order to be more widely useful to industry [PWGH90, PGH91]. We must now automate the memory design process in order to gain both cost and performance because of the complexity of these systems.

1.6 Introduction to Memory Design

There are many ways of designing an efficient system (in terms of performance and cost) which satisfies all the system specifications, but such designs must currently

be done by skilled human designers. In this research, various design tradeoffs and issues in storage architecture synthesis have been identified and characterized, and a more general approach to storage architecture synthesis has been developed, so that the design process can be automated.

1.7 A Simple Example

We design a hierarchical storage system concurrently with the datapath and also determine the input/output data-transfer schedule between various hierarchies and datapath, as the datapath itself is scheduled. The need for such a combined synthesis step is illustrated in the following example, where a simple data flow graph is scheduled considering various memory-related issues (Figures 1.2 a, b, and c). In Figure 1.2 a, the datapath scheduling is done without considering any storage-related issues. As a result, all the inputs A, B, and C are required simultaneously in step 1 so, the module(s) storing A, B and C should have 3 read ports and these inputs must be transferred on-chip from outside in one step, demanding a bandwidth of 3 words/cycle. Furthermore, a 3-word buffer is required to store these inputs after the transfer.

In Figure 1.2 b, the scheduling is done with limited read ports. This schedule results in operators requiring at the most two inputs in any step (so, we require only 2 read ports on the storage modules(s) here), but the design still requires a bandwidth of 2-words/cycle. The buffer size required here is reduced to two as A can be overwritten with C.

Finally, Figure 1.2 c shows the scheduling done with limitations on both the read ports and the bandwidth. This schedule requires only one data transfer per cycle from the external world and two inputs for operators in any step. The storage size required here remains two.

Notice that in a larger CDFG, there may be a little or no delay in execution due to data transfer as the execution of other parts of the CDFG is overlapped with the data transfer.

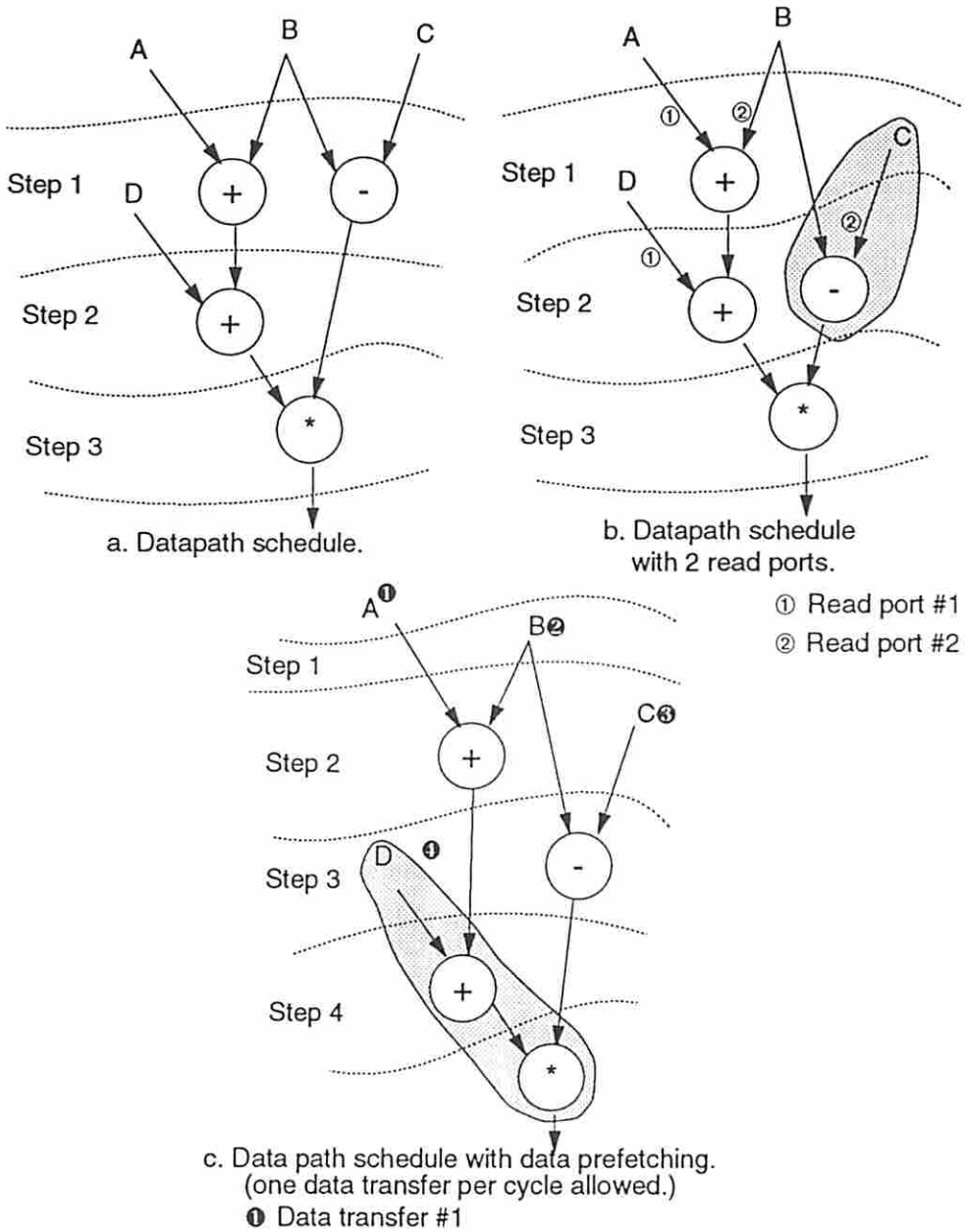


Figure 1.2: An Example Showing Scheduling with Memory Related Issues

1.8 Problem Description

The goal of this research is to develop a set of synthesis tools which will, for a given behavioral description (*i*) design the datapath, (*ii*) design the storage architecture (foreground and background memory), and (*iii*) coordinate interaction among the memory system, datapath and the external world, while satisfying all the constraints.

In addition, another important aspect of this research is the emphasis on being able to handle "real" designs. Our tools are able to synthesize designs having

- mixed control and data flow,
- multi-cycle operations,
- memories,
- I/O timing constraints,
- on-chip constants,
- arrays,
- conditional branches, and
- loops.

These capabilities of our tools allowed us to experiment with large, realistic examples. These experiments are described later in the thesis.

1.8.1 Problem Statement

The problem statement is as follows:

We are given

- the behavioral VHDL description of a memory-intensive application-specific system, which may contain inner loops and conditional branches, with the loop structure, arrays and the indexed references assumed to be already transformed and optimized;

- the module library consisting of (i) functional modules (e.g. adders) with each module characterized by its area, delay and bitwidth, and (ii) storage modules (e.g. registers, single-port/multiport register files, single-port/multiport RAMs) characterized by cost per word, number of ports, access time and storage capacity;
- area-performance constraints;
- the clock cycle, which is the duration of each control step in the datapath;
- input/output timing constraints imposed by the external world; and
- memory bandwidth constraints (the number of words that can be transferred onto the chip in one control step).

The synthesis software must produce the target system with

- a datapath consisting of operators and operation schedule,
- size and port configuration for on-chip foreground memory to store inputs, outputs and intermediate variables,
- data-transfer schedule between the datapath and on-chip memory,
- size and port configuration off-chip (or on-chip) background memory for bulk storage, and
- data-transfer schedule between the foreground and background memory.

The block diagram of the system is illustrated in Figure 1.3.

1.9 Decisions made by SMASH

The key decisions made by SMASH in designing the target system are to

- determine the number of functional modules of each type in the datapath;
- determine the operation schedule in the datapath;
- design the storage hierarchy:

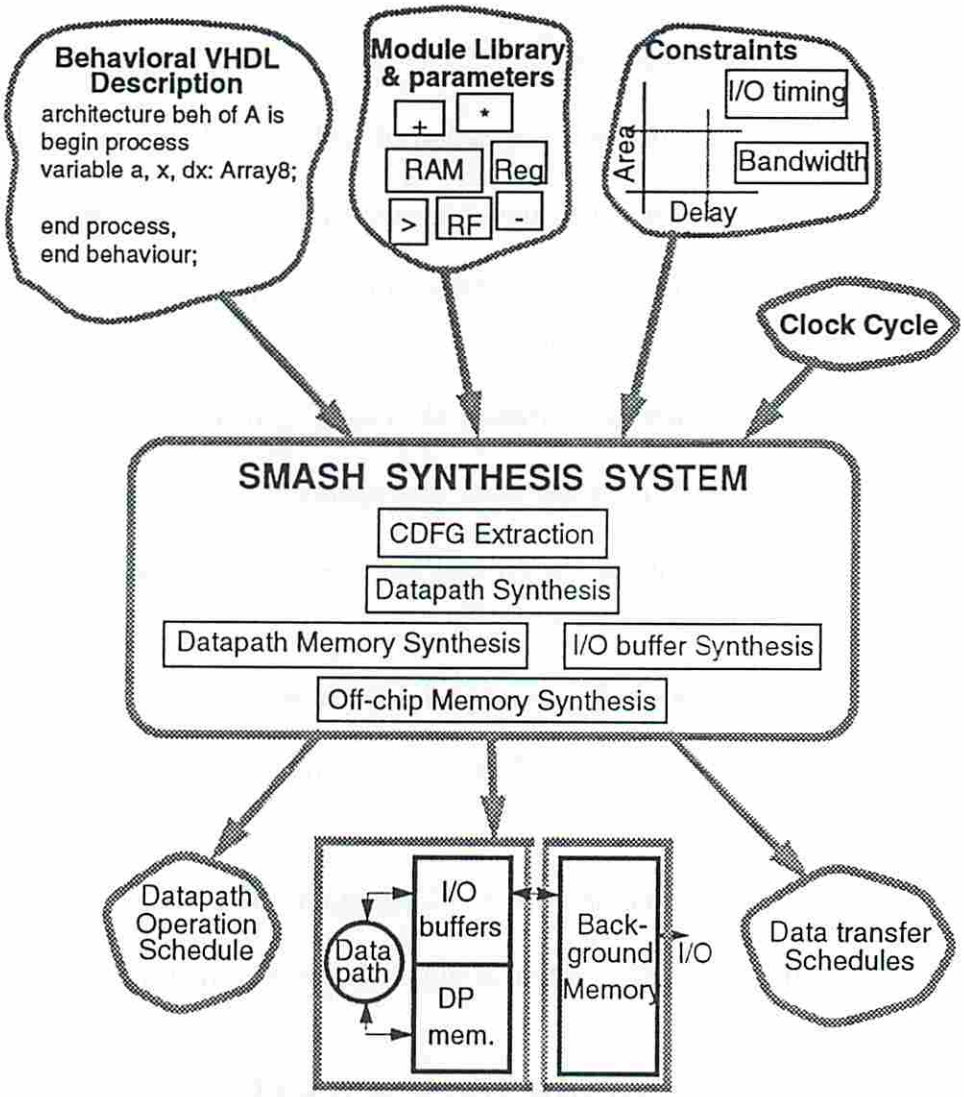


Figure 1.3: SMASH Synthesis System

  - decide the size and port configuration for foreground memory, and
  - decide the size and port configuration for background memory;

- determine the data transfer schedule between datapath and the foreground memory; and
- determine the data transfer schedule between foreground and background memory.

In the current implementation of SMASH, the number of ports on the foreground memory by SMASH, whereas the number of ports on the background memory is derived from the bandwidth between the foreground and background memory specified by the user. To vary the number of ports on the background memory the user must invoke SMASH repeatedly.

1.10 Our Approach

The overall synthesis approach consists of two major subtasks: First, operation scheduling combined with scheduling of on-chip data transfers to/from I/O buffers is performed. As a result of this scheduling, constraints are placed on the memory structure. Second, storage architecture synthesis is carried out, which includes determining the data transfers between the foreground and background memory[1]. The first step of the stepwise construction of the system takes into account the second step by looking ahead so that the second step is not overly constrained. This approach ensures that the partial design obtained in each step supports the synthesis of the next structure. Global design parameters like the bandwidth between the on-chip and off-chip memories, and the I/O timing constraints are considered when constructing the partial design in each step, tying the whole synthesis process together. The datapath scheduling imposes significant constraints on the design; therefore, all the storage structure related tradeoffs must be considered in the datapath scheduling software itself.

[1]synthesis of the storage structures is not described in this research.

Datapath design requires scheduling the data flow graph and then binding the operations to the hardware modules while taking into account the external bandwidth. In order to satisfy read/write port constraints on the storage architecture, the datapath operations and I/O buffer reads and writes by the datapath are scheduled simultaneously. The data is provided to the datapath when and only when it is required, because unavailability of the data would result in processing delays and unnecessary transfers would result in extra bandwidth and buffer-size requirements. This can be achieved by scheduling an operation in such a way that the I/O data can be prefetched into I/O buffers from the background memory before it is required.

The second step is the synthesis of the memory hierarchy. As mentioned earlier, this part of the system consists of two levels: on-chip foreground memory and off-chip background memory. Synthesis of each part involves scheduling the data transfer and creating data value bindings to the physical modules. The datapath schedule obtained in the previous step (datapath scheduling) determines the reads/writes between the datapath and I/O buffers. The objective in this step is to schedule the writes into the I/O buffers from the background memory and the writes back into the background memory from the I/O buffers such that the buffer size is minimized. While doing so the memory bandwidth constraints between the on-chip and the off-chip memory and the timing constraints on the I/O cannot be violated. To minimize the buffer size the data is transferred only if it is required and is overwritten whenever possible. The outcome of this step is a complete data transfer schedule in and out of the I/O buffers. Once the data transfer from background memory to the I/O buffers is known, we have the read schedule for the background memory. If there are timing constraints on the I/O then the background memory writes/reads of the data are already scheduled, otherwise, they are scheduled the way the data transfers for the I/O buffers were scheduled. The final step, datapath memory design, has been researched in the literature quite extensively. There are several good approaches to merge variables into larger modules like single/multi port register files, which will be described in Chapter 2.

1.11 Storage Design Tradeoffs

This section describes the three storage architecture tradeoffs included in SMASH. The total storage size, the number of read/write ports on the storage structure, and the number of clock cycles available for data transfer, can be traded off with each other as described below.

1.11.1 Storage Size *vs.* Number of Execution Cycles

The storage size *vs.* number of execution cycles tradeoff can be exploited in two situations.

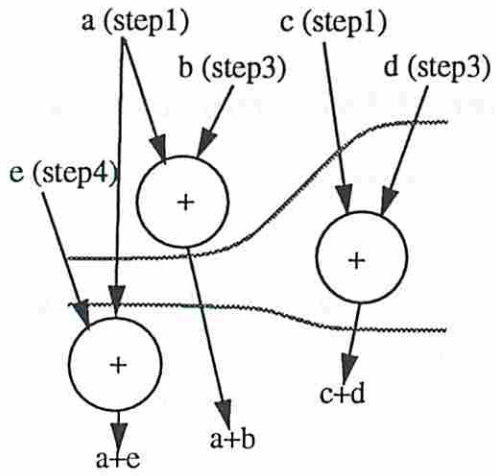
1. When data needs to be prefetched in the buffers to avoid delay in the future, it can be stored in the buffers. Otherwise, if the data cannot be prefetched into the buffers because of the storage size limitation, processing has to be delayed because the required data is unavailable in the buffers.

2. When the data is required again in the future, it can be stored in the buffers which may result in increased buffer size. Otherwise it must be fetched again, which may result in extra clock cycles in the execution.

This tradeoff can be illustrated by the example shown in Figure 1.4. The design is allowed only one adder and inputs 'a' and 'c' are available after step 1, 'b' and 'd' are available after step 3, and 'e' is available after step 4. The sequence of operations is shown in the table for both the designs. Observe that, in the first design, step 2 is utilized in prefetching 'c' for future use in step 4, whereas in the second design, step 2 is wasted due to insufficient storage space resulting in a delayed operation later on. Also, in the first design 'a' could be saved for use in step 5, whereas in the second design 'a' was fetched again, adding one more step in the total execution time. (Note that the choice of a particular storage module type is not a part of this tradeoff.)

1.11.2 Number of Ports *vs.* Number of Execution Cycles

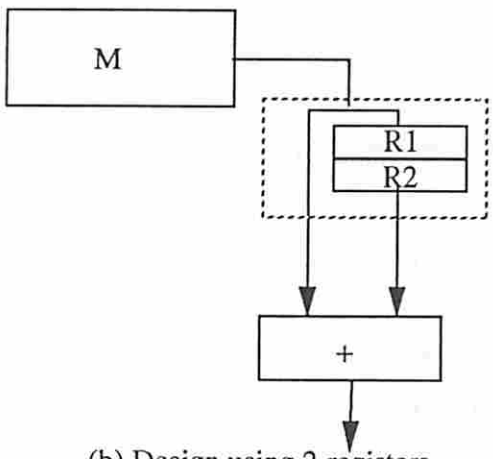
The number of ports *vs.* number of execution cycles tradeoff is a trivial tradeoff between space and time multiplexing. The required data can be transferred by



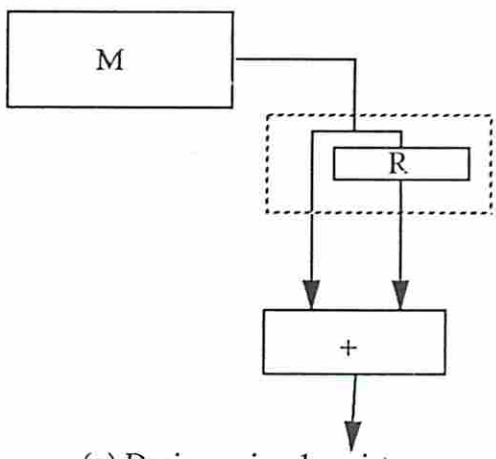
(a) A scheduled CDFG.

	Sequence b	Sequence c
Step 1	Read a from M, Store in R1	Read a from M, Store in R
Step 2	Read c from M, Store in R2	None
Step 3	Read b from M, Add a+b	Read b from M, Add a+b
Step 4	Read d from M, Add c+d	Read c from M, Store in R
Step 5	Read e from M, Add a+e	Read d from M, Add c+d
Step 6		Read a from M Store in R
Step 7		Read e from M, Add a+e

(d) Execution sequence for designs (b) and (c).

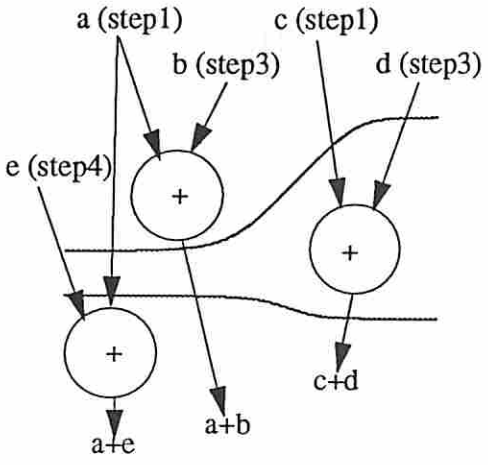


(b) Design using 2 registers.



(c) Design using 1 register.

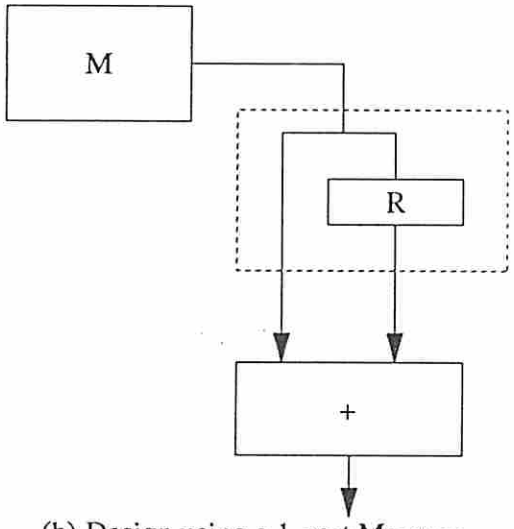
Figure 1.4: Storage Size *vs.* Number of Execution Cycles



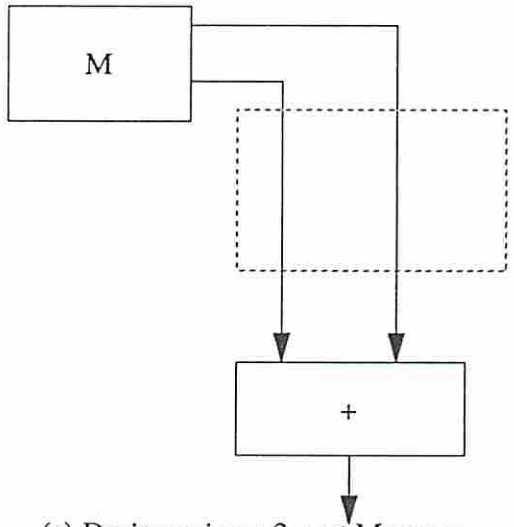
(a) A scheduled CDFG.

	Sequence b	Sequence c
Step 1	Read a from M, Store in R	None
Step 2	None	None
Step 3	Read b from M, Add a+b	Read a, b from M, Add a+b
Step 4	Read c from M, Store in R	Read c, d from M, Add c+d
Step 5	Read d from M, Add c+d	Read a, e from M, Add a+e
Step 6	Read a from M, Store in R	
Step 7	Read e from M, Add a+e	

(d) Execution sequence for designs (b) and (c).

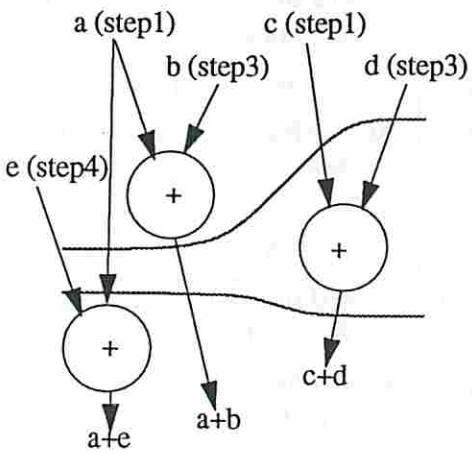


(b) Design using a 1-port Memory.



(c) Design using a 2-port Memory.

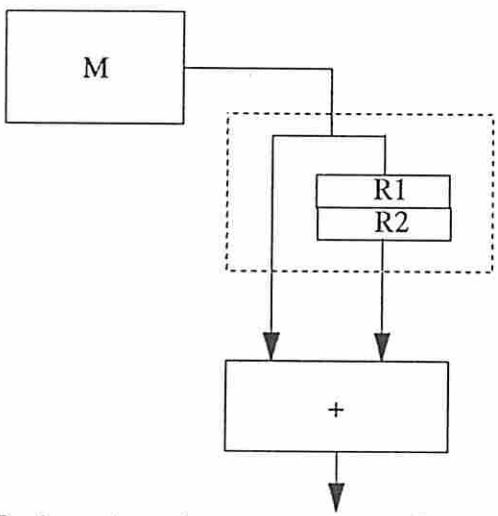
Figure 1.5: Number of Ports *vs.* Number of Execution Cycles



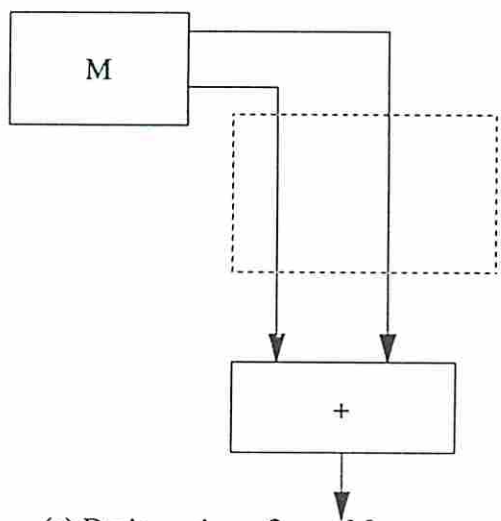
(a) A scheduled CDFG.

	Sequence b	Sequence c
Step 1	Read a from M, Store in R1	None
Step 2	Read c from M, Store in R2	None
Step 3	Read b from M, Add a+b	Read a, b from M, Add a+b
Step 4	Read d from M, Add c+d	Read c, d from M, Add c+d
Step 5	Read e from M, Add a+e	Read a, e from M, Add a+e

(d) Execution sequence for designs (b) and (c).



(b) Design using a 1-port memory and 2 registers.



(c) Design using a 2-port Memory.

Figure 1.6: Number of Ports vs. Size of Storage

transferring more words per clock cycle through more ports or by using more cycles to transfer these words.

This tradeoff is exemplified by the two implementations shown in Figure 1.5. The first design has a one-port memory and requires seven clock cycles for execution. In contrast, the second design has a two-port memory and requires only three clock cycles.

1.11.3 Number of Ports *vs.* Size of the Storage

From the above two tradeoffs one can deduce the tradeoff between the number of ports and the size of the storage. This tradeoff is very effective when data values are used again. In such a case, they can either be retrieved repeatedly whenever needed with more ports or must be saved for the future use which will increase the storage size.

In the example shown in Figure 1.6 'a' is used twice. In the second design there are two ports on the memory so 'a' could be fetched twice instead of being stored. In the first design there is only one port on the memory, so 'a' is stored for future use.

1.11.4 3-way Tradeoff

We have seen that in storage architecture design there are three parameters which vary while making the cost-performance tradeoffs:

1. number of ports,
2. size of the storage structure, and
3. execution time.

These tradeoffs must be made during the datapath scheduling step, as the datapath schedule determines the execution time, buffer size and the read/write port requirements. If these decisions are postponed until the storage synthesis task, the datapath schedule might have to be altered in order to accommodate the storage structure tradeoffs. This could result in complex backtracking and iteration.

1.11.5 A Storage Architecture Tradeoff Example

Consider the following example of a noise-cleaning algorithm. The filter can be described as shown below [Pra78]:

$$\text{if} \qquad \left[x - \frac{1}{8}\sum_{i=1}^{8} O_i \right] > \epsilon \qquad \text{then}$$

$$x = \frac{1}{8}\sum_{i=1}^{8} O_i$$

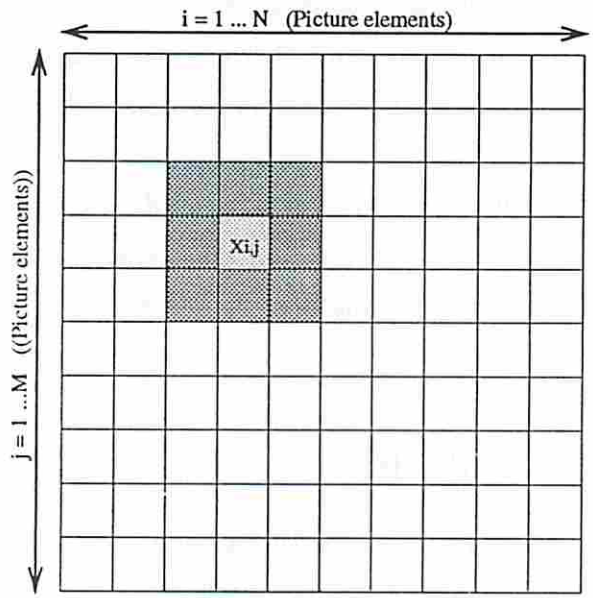
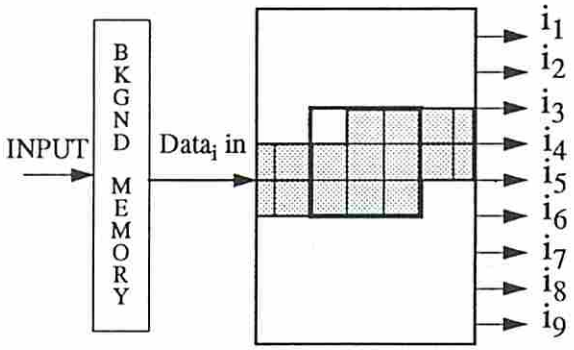


Figure 1.7: Example of a Noise Cleaning Algorithm.

The whole image is assumed to be stored in an off-chip background memory and only the required data is transferred into the on-chip input buffers. Inputs buffers are the part of the memory shown in detail in the designs in Figures 1.8, 1.9, 1.10, and 1.11. The buffers are being used to fetch the data from the background memory and make them available to the datapath for further processing. The values are required more than once (actually 9 times). Depending on the BW_{on-off} and the execution time allowed, the data may have to be stored in the buffers. That will determine the size of the buffers.

In designs 1.8, 1.9, and 1.10, it is assumed that the datapath has already been designed and requires all 9 data of the image for processing at the same time. The storage-related tradeoff in these designs is described below. In addition, design 1.11 illustrates the need for combining the datapath design with the design of storage architecture.



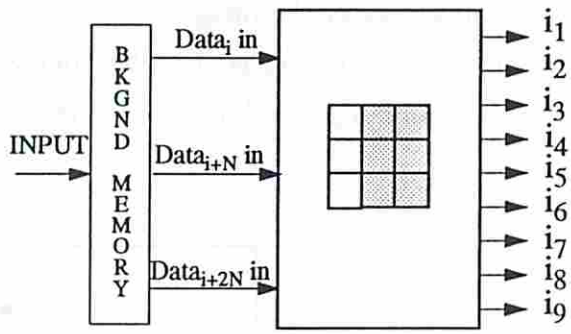
Storage size	2 Rows + 3
Number of input ports	1
Number of output ports	9
Window processing rate	every clock cycle

Figure 1.8: Design 1.

The storage tradeoffs considered in SMASH illustrated by these designs are as follows (Figure 1.12).

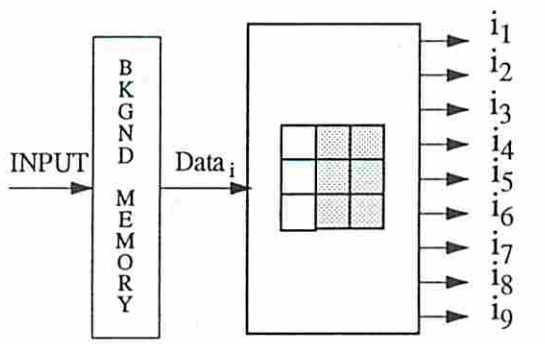
- Designs 1 and 2 show the tradeoff between the size of the storage and the number of ports,
- Designs 2 and 3 show the tradeoff between the number of ports and the execution time, and
- Designs 3 and 1 show the tradeoff between the size of the storage and the execution time required.

In Design 1 the buffer size required is $2 \times \text{rows} + 3$. Only 1 write port is required, as only 1 datum is transferred from the background memory in each cycle. 9 read



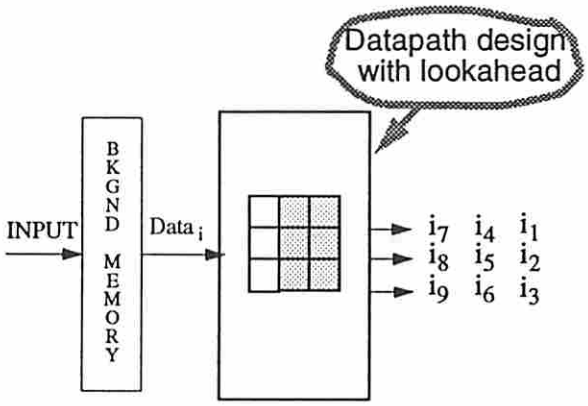
Storage size	**9**
Number of input ports	**3**
Number of output ports	9
Window processing rate	every clock cycle

Figure 1.9: Design 2.



Storage size	9
Number of input ports	1
Number of output ports	**9**
Window processing rate	every 3 clock cycles

Figure 1.10: Design 3.



Storage size	9
Number of input ports	1
Number of output ports	3
Window processing rate	every 3 clock cycles

Figure 1.11: Design 4.

ports are required to make all the 9 data points available to the datapath. In this implementation the datapath can process a window every clock cycle. The buffers in Designs 2 and 3 are of size 9 but in Design 2 there are 3 write ports whereas, in Design 3 there is only 1 write port. Both the designs provide 9 read ports. Both the designs need 3 new data from the background memory to process the next window as the data is not being stored in the buffers in these cases. Since, in Design 2, there are 3 write ports, 3 data points can be transferred from the background memory into the input buffer every clock cycle; the datapath processes a window every clock cycle. Design 3 has only 1 write port, therefore 3 clock cycles are needed to transfer the new data. This design processes a window every 3 clock cycles. Notice that in Design 3 the delay is caused by the data transfer and not the datapath.

Design 4 illustrates the necessity of looking ahead into the storage architecture parameters while designing the datapath. This design has the same performance as Design 3 but it requires a slower and cheaper datapath as well as less read ports on the buffer. The datapath in this implementation is designed with lookahead into the storage parameters BW_{on-off}, which is the number of write ports on the buffers.

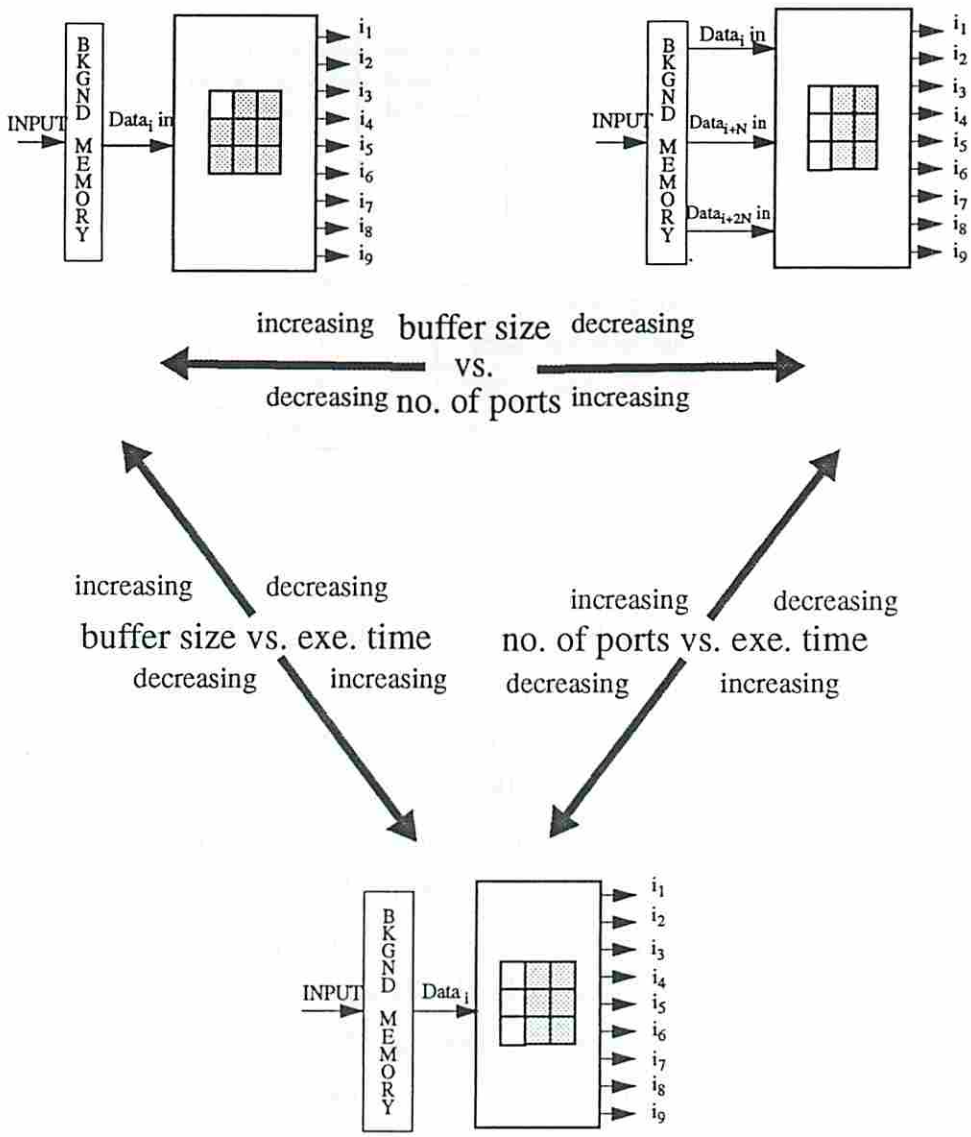


Figure 1.12: 3 Way Tradeoff in Storage Architecture

During the datapath design SMASH determined that the bottleneck in processing is the small bandwidth, BW_{on-off}; it takes 3 clock cycles to transfer the required data for processing. Therefore, a high performance datapath (as used in Design 3) cannot improve overall performance. Knowing this, a slower and cheaper datapath is designed for this implementation. The datapath in this case processes a window in 3 clock cycles and requires only 3 data per clock cycle. Furthermore, by reducing the data requirement per clock cycle, the number of read ports on the input buffers could also be reduced resulting in a cheaper storage structure. Thus a cheaper datapath as well as storage architecture was designed without degrading the overall performance. This example illustrates the need for combining the datapath design with the design of storage architecture.

1.12 SMASH as a Part of USC

High-level synthesis tools (like SMASH) are at the core of any system-level tools (like USC) and they can be used in three basic ways: (*i*) bottom-up, (*ii*) top-down, or (*iii*) mixed, in system synthesis. SMASH can be used in any of three design methodologies within USC.

- Using SMASH in bottom-up design flow: In bottom-up design flow, SMASH provides the area-delay tradeoff curve for system-level tools like SOS and Propart. Several implementations of a design with varying cost/performance parameters can be quickly synthesized using SMASH and then these parameters are provided to the system-level tools. Finally, the system-level tools select the most cost-effective design for the final system design.

- Using SMASH in top-down design flow: When used in top-down design flow, SMASH synthesizes ASIC designs to meet the cost/performance constraints determined by system-level tools such as SOS and Propart.

- Using SMASH in mixed design flow: SMASH can generate the area-delay tradeoff curve of individual components for SOS and Propart and then can produce the selected design point for the final design.

1.13 Thesis Organization

The thesis has been organized in the following manner. Chapter 2 surveys the related work. It describes existing research in high-level synthesis in general, and high-level synthesis of storage structures. Chapter 3 describes the SMASH approach to solving the storage synthesis problem. Chapter 4 discusses the techniques we have developed for various design parameter estimations. These estimations are used during the synthesis process. Chapter 5 describes the details of the synthesis techniques used in SMASH for datapath synthesis combined with I/O transfer scheduling while looking ahead in storage structure tradeoffs. Chapter 6 describes the storage structure synthesis performed in SMASH. Chapter 7 outlines the experiments we have performed to demonstrate our ideas and the results obtained. Finally, Chapter 8 presents the conclusions and future directions. Appendix A briefly describes a utility developed during the course of research. The utility was widely used in our research as well as in a variety of other experiments. Appendix B contain the VHDL descriptions of all the examples synthesized by SMASH as described in Chapter 7.

Chapter 2

Related Research

2.1 Introduction

The goal of this research is to develop a software system which performs high-level synthesis with emphasis on memory architecture synthesis. In the following sections we will briefly review the research performed in high-level synthesis in general, and then review the work done in high-level synthesis of memory architectures.

2.2 High-Level Synthesis Research

There has been a lot of research in automatic synthesis of a design structure from a behavioral specification. Some of the systems are briefly described below.

The ADAM system at USC deals with datapath synthesis, starting with a control flow/data flow graph as the behavior description of the desired system [JKMP89, GKP85]. Pipelined designs are synthesized by Sehwa [PP88] or a multi-chip scheduler, and non-pipelined designs by MAHA [PPM86] or LADS [WP91]. MAHA uses freedom-based scheduling to schedule the data flow graph. We also propose to use a very similar but modified approach for our scheduler. For module allocation, MABAL is used [KP90]. MABAL allocates functional units as they are required and registers are allocated by doing life time analysis of the intermediate values. It also trades off functional units with interconnects (mux, drivers etc.) when required.

The ELF system uses list scheduling with urgency of performing that operation before any enclosing timing constraint as the priority function [GK84]. Graph grammar productions are used for allocation of functional units and registers. After

scheduling and allocation, a greedy global partitioning/clustering algorithm is used to minimize the interconnect.

Paulin's HAL system accepts a data flow/control flow graph as the design behavior input [PK87, PK89]. It uses a powerful technique for scheduling known as force-directed scheduling. In force-directed scheduling a "force" for each operation in each possible step is used as the priority function. First both an ASAP and an ALAP schedule are generated. These schedules indicate the possible control steps for each operation. Then, for each operation for each possible time step, distribution graphs are computed. These graphs are used to calculate self force for each operation, as well as external forces, reflecting the effect of the schedule on other operations. The total force is used as the priority function in the scheduling. Allocation is done by a rule-based expert system followed by operation binding using a greedy algorithm. A clique partitioning algorithm is used to minimize the number of registers. Then multiplexers and interconnects are added. Finally, the design is improved by some local optimizations.

CMU's second CMU-DA system supports behavioral transformations [DPST81]. They use ASAP technique for the scheduling. Their datapath allocator EMUCS binds operations onto each hardware element based on the cost of each unbound data flow element.

Some other synthesis packages from CMU include DAA (Design Automation Assistant) a knowledge based expert system [KT83]; CSTEP (control step scheduler), which uses list scheduling with timing constraint evaluation as the priority function [LT89], and FACET which uses ASAP scheduling [TS83].

University of California at Berkeley's HYPER system applies optimizing compiler transformations on the data flow graph, determines the lower and the upper bounds on the number of functional units, registers, and buses in order to determine the initial number of resources [RP90, RMV+88]. Then, it schedules each control step in turn. It applies transformations like multiplexer reduction and datapath partitioning to perform module binding.

Stok's scheduler uses a variation of force-directed scheduling [Sto91]. He tries to balance the use of each functional unit. Multi-cycle operations and chaining are allowed. For datapath synthesis, operations are assigned to functional units, based

on a weighted clique partitioning algorithm. Intermediate variables are merged in register files using an edge coloring algorithm.

IMEC's CATHEDRAL-II system is designed for the high-level synthesis of DSP chips [RMV+88]. It constructs datapaths from a set of execution units and also a set of memories (register files and buffers), I/O units and controller modules. Their scheduler uses list scheduling, with priority to the operations on the longer critical path. Intermediate values are assigned to the register files based on their lifetime analysis. CATHEDRAL-2nd and CATHEDRAL-III are two successors to the CATHEDRAL-II system. Philips' PIRAMID system uses CATHEDRAL-II as a synthesis engine along with a floorplanner and a module generator.

AMICAL from INPG/TIMA, France, is targeted towards control-flow dominated systems [JPO93]. It provides an interactive environment where automatic and manual synthesis can be mixed. Starting with a pure VHDL input, AMICAL produces a full specification for existing logic and RTL synthesis tools. The target architecture allows complex, synchronous, heterogeneous, parallel application-specific architectures.

There are many more synthesis systems from different universities, as well as from industry. Unfortunately, no single universally-accepted theoretical framework has yet emerged, due to the complexity of the whole synthesis process.

2.3 Memory Architecture Research

There has been very little work done on the automatic design of memory hierarchies, with the exception of some European and Canadian activity. The bulk of research on memory hierarchy design in the United States involves theoretical and probabilistic studies for general purpose computer design where the design issues are quite different from the issue in ASICs. There has been little application-specific memory design research performed. In a general purpose computer, the memory access pattern varies from application to application; therefore, for these machines the memory design is based on probabilistic models. On the other hand, SMASH will be used for systems designed for specific applications. In our case, the memory access pattern is not only relatively fixed but also known before hand. This mostly

deterministic access characteristic helps us in being more specific, hence more efficient in our designs. This also makes it feasible to automate the design process. An example of the kind of tradeoff study related to our work is the work by Parker and Nagle which was performed a number of years ago [NP77].

Many researchers have talked about the need for fast, large memories for high-performance systems. Multi-port RAMs can provide high throughput as simultaneous access is possible. The MIMOLA design system is the first system to make tradeoffs in the use of multi-port memories [Mar79, Zim79]. The design space parameters in MIMOLA included memories and number of memory ports. The designer starts with minimizing the high-cost low-utilized resources like memory ports. He/she begins with very small number of resources in the database. The system description is processed and if more resources are demanded in a microstatement, the program attempts to resolve it by sequentialization, introduction of storage cells for intermediate results or restructuring. Unresolvable situations are reported to the designer. He can revise his declaration and process the description again.

Balakrishnan et al. presented an approach to use multi-port memories to implement single isolated registers [BMBL87, BMB+88]. This approach "packs" these registers into a homogeneous group of modules. They also consider register interconnection to operators while packing these registers in order to minimize the interconnection cost. They solve the register packing problem sequentially, i.e., by placing registers into one memory at a time. Each memory is packed by solving an integer-linear program. Leftover registers are then candidates for the next memory and so on. For a more global approach and a faster solution, a heuristic method based on a graph model of the problem was developed. After finding a feasible packing, a 3-step post processing is done to reduce the interconnection cost. The first step involves interchanging pairs of registers between memories. The second step minimizes the number of memories that require access to both input terminals of a given functional unit. And the third step minimizes the number of operator terminals that require access to multiple ports of a given memory.

Chen explored the design space for multiport memory synthesis [CS90, Che91]. The memory modules were generated in sequence, which resulted in locally optimal

solutions and in some cases failed to generate the globally optimal solution of minimizing the number of multiport memories[CS90]. Later the work was modified and extended to generate a more globally optimal solution [Che91].

Ahmad and Chen use 0-1 integer-linear programming to group intermediate variables in the datapath into a minimum number of multiport memories depending on their ports and their access pattern [AC91]. The formulation also take into account the interconnection hardware. Though their technique works well with small examples, 0-1 ILP is not suitable for bigger examples and will exhibit run-time problems.

For each design style there is an area/time tradeoff possible when we physically lay out the memory cell array. In the case of large memories, a large row count causes a very long bit line, resulting in more delay and an inefficient layout. S. Hiorofume et al. presented a compiler having a flexible port arrangement and layout [HMFK90].

Grant et al. suggested an approach to group the memory requirements of various operators such that control and communications may be optimized [GD90]. They consider single-port memory modules. To optimize the communication network between the functional units and memory, simple heuristics are used which optimize the write-bus network and read-bus network. They also optimize the controller by optimizing the control bit sequence. In an earlier study the same group studied a different aspect of memory synthesis, address generation hardware [GDF89].

Stok optimizes register files during the synthesis process by splitting read and write phases of registers, and by considering parallel storage and rewrites of values that have to be read several times [Sto89].

Recently Lippens et al. from the Philips Research Labs, in PHIDEO, implemented techniques to perform automatic memory allocation and address allocation for high speed applications [LvMvdW+91]. They synthesize memory after the design of arithmetic units and after scheduling. They model multi-dimensional periodic signals as data streams and then manipulate these streams to form a distributed memory structure. They assume a limited number of memory types available to them (1 and 2 port RAMs) and so their approach is to distribute the data among parallel memories. They do not distinguish between background and foreground memory. They perform memory synthesis independent of datapath synthesis. They do not consider conditional branching in the behavior of the system.

In IMEC's CATHEDRAL-II, efficient storage schemes and memory access techniques were implemented by De Man et al. [VBM91]. According to them, efficient storage schemes and memory access are as crucial as allocation and scheduling of datapaths in DSP ASIC design. They compile multi-dimensional data structures into distributed dual-port register files and single-port SRAMs. They also consider multi-dimensional signals and optimize the high-level memory organization using transformations. They use a polyhedral-based model for the linear, piecewise linear and data-dependent signal indexing [FBS+93]. The model is used to derive alternative control flow structures for a given data flow specification to optimize large-scale memory organization both in terms of storage locations and access order.

All the above efforts concentrated on separate aspects of memory synthesis. An overall tradeoff approach like the one we have constructed has not been reported elsewhere.

Chapter 3

Problem Approach

3.1 Introduction

We observed in the introductory chapter that every new design of a storage architecture requires a different strategy for the most efficient design in terms of cost and performance. Each strategy is unique in itself and may be difficult or impossible to automate. We also observed that storage architecture is companion to the datapath and the design of the datapaths and memory hierarchies must be coordinated. In this research, our goal is to develop a more general approach which combines the synthesis of datapaths with the synthesis of storage structures. We wish to apply the approach to memory-intensive application-specific systems in order to automate the design process while producing efficient and correct designs. Of course, this kind of general approach cannot match human designs and may suffer from larger or slower hardware as compared to human designs. Nevertheless, we wish to exploit all the advantages design automation has over manual designs like faster design time, less errors, and exploration of a larger design space.

This chapter describes the target system architecture produced by SMASH, the SMASH module library which includes functional modules as well as storage modules, the clocking scheme used in SMASH, and the overall approach to synthesize the target design implemented in SMASH.

3.2 Target System Architecture

Our target system architecture consists of a datapath and a hierarchical storage structure, as shown in Figure 3.1. Our view of the architectural structure and the role of each structure in the system is described in this section.

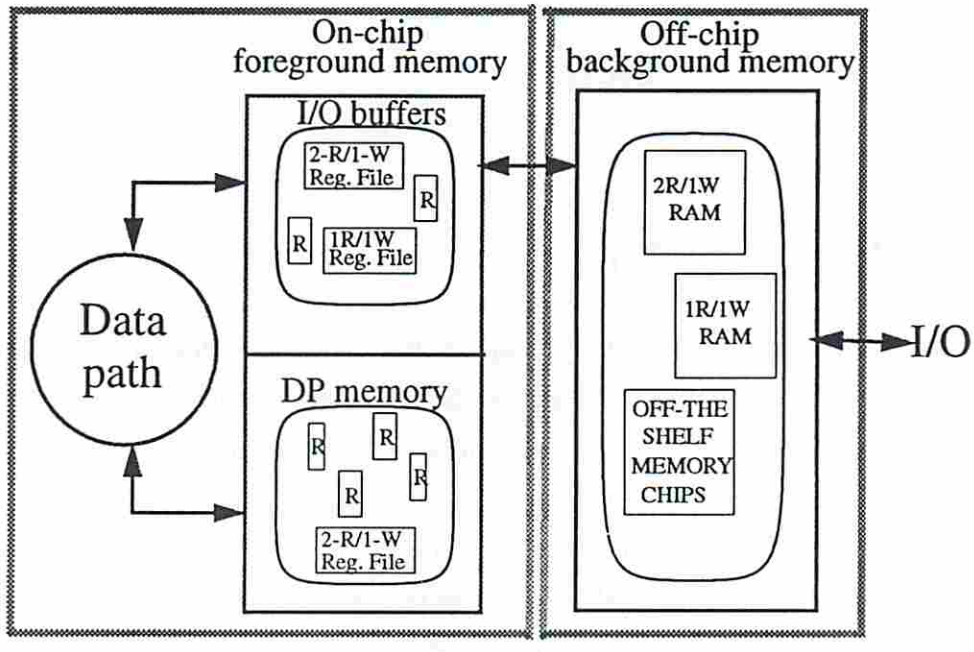


Figure 3.1: Target Architecture in SMASH

3.2.1 Datapath

In our model, the datapath consists of the functional hardware to execute the specified behavior. The functional operators used in the datapath are characterized in the user-specified module library in terms of bitwidth, number of inputs and outputs, area, and execution delay. This datapath model is the conventional model used by other researchers except for one difference that in our model, the datapath *per se* does not have any kind of storage capability. All the intermediate variables are stored in datapath memory, which is described later.

3.2.2 Storage Architecture

The overall storage structure is classified into two major sub-architectures based on their location in the storage hierarchy. These storage sub-architectures are as follows (Figure 3.1):

1. *on-chip foreground memory* to store inputs, outputs and intermediate variables, and

2. *off-chip background memory* for bulk storage.

The sub-architectures may have similar hardware structure but may vary in functionality. The overall architecture of the system may be fixed, but each sub-architecture is determined by the synthesis system and may consist of various storage modules and devices like registers, register files, and RAMs.

The cost of each sub-architecture is a function of

1. the number of read and write ports on the sub-architecture, and

2. the size of the sub-architecture.

The total cost of the storage sub-architectures can be further optimized during the actual construction of the storage structure by using the most cost effective modules from the storage library, as explained later in the thesis. The sub-architectures are briefly described below.

On-chip Foreground Memory

The on-chip foreground memory consists of I/O buffers and datapath memory. In cases where the background memory can directly interface to the datapath, the I/O buffers are made transparent by using simple wires to replace storage modules.

I/O buffers are that part of the foreground memory which temporarily stores I/O variables that are required for processing, and makes the variables available to the datapath in appropriate control steps. In each control step the required data is loaded onto the buffer ports and as the datapath reads data from the buffers, the controller adds new data to them from the off-chip memory for further processing. Similarly, the output variables are first stored in the on-chip memory as they

are produced and then are transferred to off-chip memory at an appropriate time, overlapped with the execution of the datapath. An efficient overlap between the processing and data transfer can make the data access latency transparent to the system. In addition, I/O buffers are also used to store data which will be used in future steps; if such data can not be retrieved easily in the future then it may be retained in the buffers.

The relevant parameters that are to be determined for the I/O buffers are

1. total buffer size, which is determined by the maximum number of inputs and outputs stored in the buffers in any given control step, and
2. the number of read ports (R_{buf}) and write ports (W_{buf}) accessible to the datapath, which is the maximum number of inputs and outputs accessed by the datapath in any given control step.

The user specifies the bandwidth between the buffers and the off-chip background memory BW_{on-off}. BW_{on-off} is the maximum number of inputs and outputs that can be transferred between the on-chip I/O buffers and the background memory in one control step.

Datapath memory stores the intermediate variables in the datapath. The parameters which determine this subpart are derived from the scheduled CDFG and include

1. the number of intermediate variables, and
2. the lifetimes of these variables.

Datapath memory synthesis tradeoffs have already been performed by other researchers [BMB+88, Che91, Sto89] and are not described in this thesis.

Off-chip Background Memory

The off-chip background memory is the bulk storage. All the I/O data values from/to the external world are stored in the background memory. The purpose of the off-chip memory is to provide large cheap storage space, just as in a general purpose computer. In general, off-chip memory can be *shared* or *distributed* between various

parts of the datapath or between multiple datapaths in a multiprocessor system, but in SMASH distributed memory is implemented for the following reasons:

1. Performance requirements are generally quite high for target systems. Distributed memory is faster and more efficient in access as the data values can be accessed directly by the processing elements.

2. We know (at least we can implicitly enumerate) the access pattern of the data. So, we can efficiently distribute the data for different datapaths, saving us unnecessary routing and switching.

The parameters relevant to off-chip memory are

1. the number of read and write ports (each equal to the user-specified BW_{on-off}), and

2. the number of words, which is expected to be large compared to the size of on-chip storage, and which is implicitly determined by our software as a side-effect of the data transfer scheduling.

3.2.3 Target Architecture Discussion

Although the top-level target architecture is fixed, there is a great deal of flexibility for each subpart. The number of read and write ports in each sub-architecture is variable and is decided by SMASH. Each of these sub-architectures can be implemented using a heterogeneous combination of storage modules. For example, the I/O buffers in the on-chip memory can be made up of a heterogeneous combination of registers, single-port/multi-port register files, and single-port/multi-port RAMs. Datapath memory may consists of registers and single-port/multi-port register files and may even contain RAMs if the storage requirements are huge enough. The background memory, which is bulk off-chip storage, is constructed using larger modules like RAMs. In an extreme case, a subpart can degenerate to interconnections if there is no need for any storage. Such a part will have no cost and no storage capability.

The cost model used in SMASH is in terms of chip area. The model assumes that the cost of the storage structure primarily depends on the number of required ports and the total size of the structure. The objective of the techniques used in SMASH

is to minimize the number of ports and the total size of the storage structure. The heterogeneous construction mentioned above could further minimize the chip area of each sub-architecture by using the most cost effective modules from the given library.

In the proposed model, the bandwidth between the on-chip and the off-chip memory, BW_{on-off} imposes cost constraints on the overall design because of (i) the pin constraints on the chips, and (ii) the expense of having multiport memory for off-chip bulk storage. Furthermore, the access time for the off-chip background memory will be greater because (i) it is off-chip, and (ii) it is bigger in size. Since access to the data values in the off-chip memory may be slower than the on-chip access as a result, the software schedules the transfer of the input variables from off-chip memory into the on-chip I/O buffers before they are required in order to avoid delay in the execution.

An interesting feature of this model is that the storage hierarchy can be ignored by providing higher BW_{on-off} (which corresponds to more ports on the background memory). In such a situation, SMASH can produce designs without I/O buffers as done in PHIDEO [LvMvdW+91].

This categorization of storage architecture is quite similar to the one used for general purpose computer architectures. However, their datapath and control architectures are clearly different from application-specific designs, and hence the storage architectures design problem differs also.

3.3 Module Library Characteristics

In this section we describe the specification of the module library in SMASH including the characteristics of each module that must be specified in the library. The actual library used in our experiments is described later in the thesis in Chapter 7. The module library in SMASH has two components:

1. functional modules and

2. storage modules.

3.3.1 Functional Modules

The functional modules are characterized in the conventional way in terms of

1. cost (which could be area, number of transistors, or static power dissipation),
2. execution delay,
3. number of inputs,
4. number of outputs, and
5. bitwidth of each input and output.

Examples of functional modules are adder, multiplier, and comparator.

3.3.2 Storage Modules

The storage modules are characterized in a different manner. Their specification includes

1. bitwidth per word,
2. cost of storage (area, number of transistors or static power) per word,
3. maximum storage capacity per module, and
4. number of read and write ports on the module.

A brief description of some example storage modules is given below:

- Register or latch: The register is the simplest storage element. It can store only one word at a time. Its bitwidth is predetermined.
- Single-port register file: A register file is a collection of registers with addressing hardware included in it. The size of the register file (the number of words that can be stored) is variable but must be determined prior to instantiation in the layout. The register files also have a maximum capacity limit per module. The cost of each register file is a function of its size. Their bitwidth is predetermined.

- Multi-port register file: The multi-port register file is the same as the single-port register file except that it has multiple read and write ports. Usually, there is only one write port and multiple read ports. Each read port has an address bus, an output data bus, and a read enable signal. Similarly, the write port also has an address bus, an input data bus, and a write enable signal. Simultaneous reads from the same location are possible but in the case of multiple write ports, simultaneous writes to the same location are prohibited.

- Single-port RAM: Single-port RAM modules considered in the library are to be used on-chip. Each RAM module has a data input bus, address bus, an output enable, and a write enable. The output can be standard or tristated. The bitwidth and the number of words per module are specified before silicon compilation. The cost of each RAM module is a function of its size.

- Multi-port RAM: Multi-Port RAM modules are also used on-chip. Each write port on the RAM module has an address bus, a input data bus, and a write enable signal. Similarly, each read port has an address bus, data out bus (standard or tristated), and an output enable bus. Simultaneous reads from the same address are legal but simultaneous writes to the same address are obviously not allowed. Timing is critical for the write vs. read operation only when accessing data that is being written in the current address cycle.

3.4 Clocking Scheme

We assume two-phase clocking for our target system. The data is written to the off-chip background memory (either from the external world or from the I/O buffers) in phase one ($\phi 1$) and read from the off-chip background memory (either by the external world or the I/O buffers) in phase two ($\phi 2$). Similarly, data is read from the on-chip foreground memory (either by the background memory or by the datapath) in $\phi 1$ and written to the on-chip foreground memory (from the background memory or by the datapath) in $\phi 2$. The datapath processes the data in $\phi 1$ and writes it back into the buffers in $\phi 2$. In case the design does not have I/O buffers, the datapath interacts directly with the background memory; so instead of writing the data back

into buffers, it writes it into the background memory in $\phi 2$. The scheme is shown in Figures 3.2 and 3.3 and illustrated in the following example in Figure 3.4.

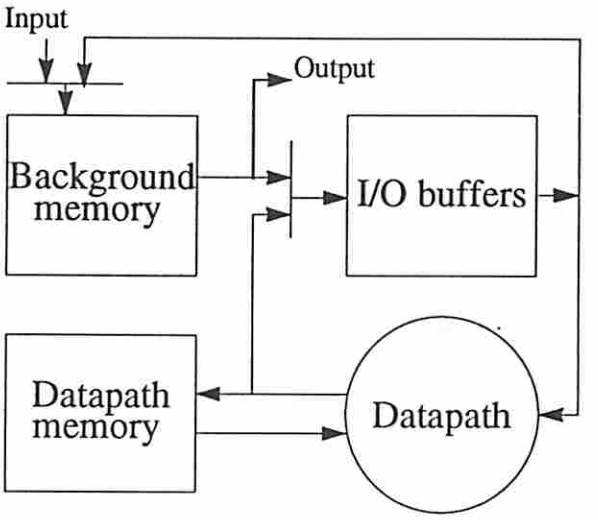


Figure 3.2: Target architecture with Communication Links

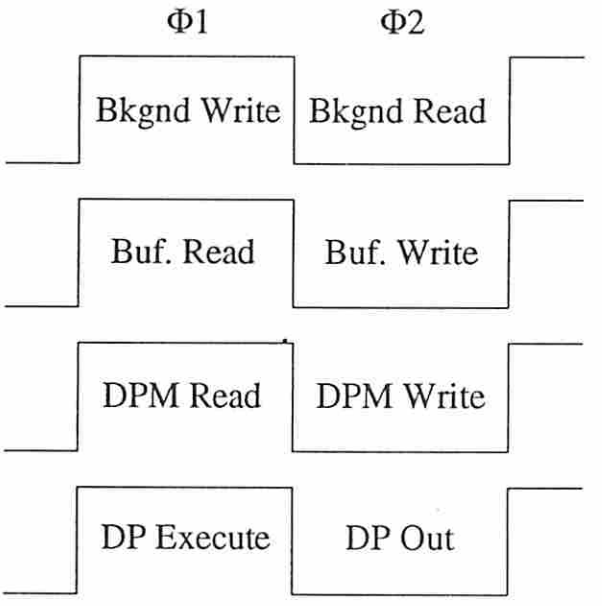


Figure 3.3: 2-phase Clocking Scheme in SMASH

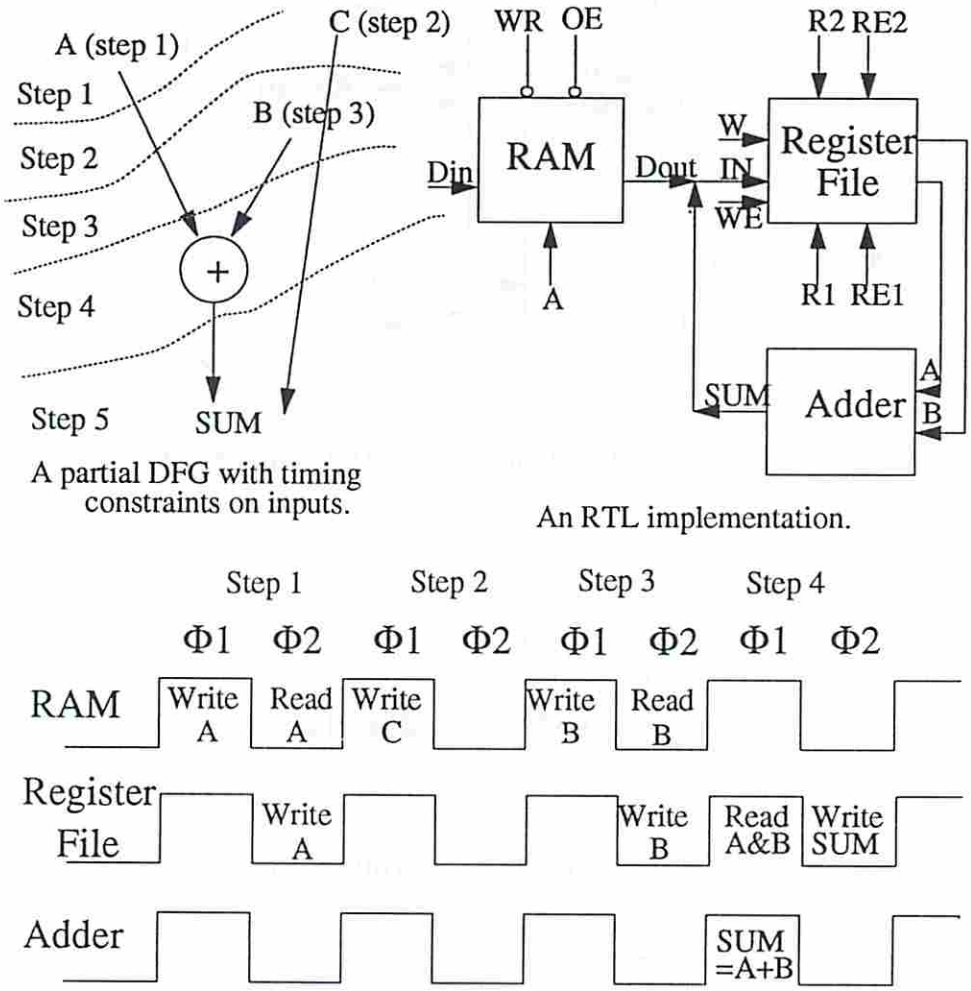


Figure 3.4: An Example Illustrating 2-phase Clocking

3.5 Overall Synthesis Approach

Our overall approach is shown in Figure 3.5. The synthesis process starts with the VHDL behavioral description of the system to be designed. This description is then compiled and translated into a control data flow graph (CDFG) represented in DDS (Design Data Structure) [KP85, KP83] format. Next, the datapath for this CDFG is synthesized, followed by synthesis of the supporting storage architecture. Finally, the RTL netlist of the synthesized design is generated, and then compiled to obtain the chip layouts. These steps are briefly described below. The two primary tasks in the synthesis process *viz.* combined datapath and I/O transfer synthesis, and storage architecture synthesis are described in detail in Chapters 5 and 6 respectively.

3.5.1 Control Data Flow Graph Extraction

The very first step in the synthesis process is to extract the control data flow graph (CDFG) of the target system from the VHDL description. This is done by generating the DDS description of the system using V2DSS [CP91] and then extracting the relevant information from the DDS description. The DDS describes the data flow in its *data flow model* and the control flow required for conditional branches and inner loops in its *timing model*. These two models are merged into a CDFG for processing by SMASH.

The CDFG specifying the target system is defined as follows:

Definition 3.5.1 *A CDFG is a directed acyclic graph $G(V, E)$ where*

- *V is a finite set of nodes. Each node $v \in V$ represents an operation O_k in the behavioral description of the system. It includes control-flow operations like* distribute *and* join*; and*
- *E is a finite set of directed edges between the nodes in the CDFG. A directed edge e_{ij} from node $v_i \in V$ to node $v_j \in V$ exists in E if (1) the data produced by v_i is consumed by v_j (data edge), or (2) the data produced by v_i controls v_j (control edge).*

Some related definitions follow. These definitions are used later in the thesis.

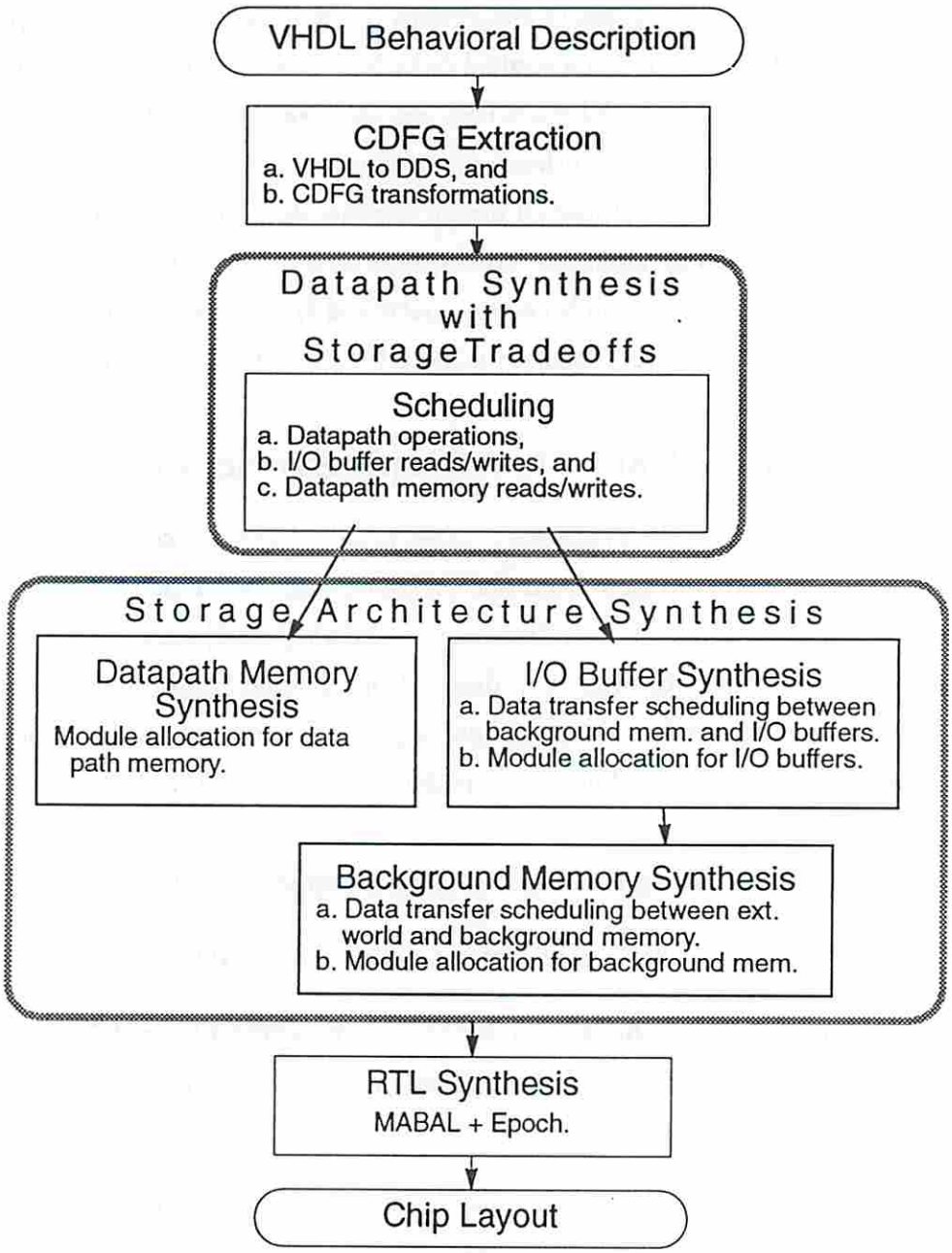


Figure 3.5: Synthesis Approach in SMASH

Definition 3.5.2 *$Pred(v) \subset V$ is the set of all immediate predecessors of node v.*

Definition 3.5.3 *$Succ(v) \subset V$ is the set of all immediate successors of node v.*

Definition 3.5.4 *$Op(v) = O_k$ implies that node v is executed using an operator of type O_k.*

Definition 3.5.5 *$V_{O_k} \subseteq V$ is a set of all nodes of operation type O_k in the CDFG i.e.*

$$V_{O_k} = \{v \mid v \in V \text{ and } Op(v) = O_k\}$$

Example: V_R is a set of all the read nodes in the CDFG.

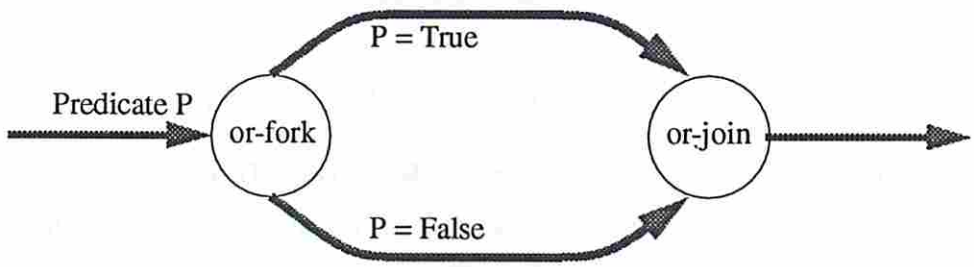
The CDFG represents a partial order $\prec$ on all the operations in the behavior of the system. The partial order defines the operational precedence constraints. For example $v_1 \prec v_2$, implies that v_1 has to be fully executed before v_2 can start, where v_1 and v_2 are any two operations in the behavioral description *i.e.* $v_1, v_2 \in V$.

Representing Conditional Branches

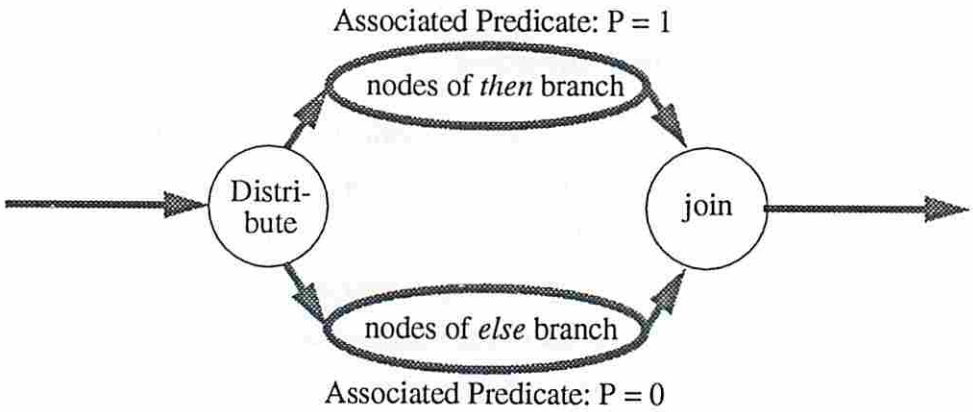
To represent the conditional branches, a predicate is attached to each node in the CDFG under which that node is executed. The predicate is extracted from the *range* information in the control and timing model of DDS. In DDS, a pair of conditional (*i.e.* exclusive) branches is represented by a pair of ranges connected at both ends by an *or-fork* and an *or-join* point in the control flow graph; a predicate is attached to each range describing the conditions under which that path is to be taken, as illustrated in Figure 3.6 a. Bindings (triples) link operations in the DFG to time ranges in the control and timing model. While constructing the CDFG for SMASH, the appropriate predicates are attached to each node as shown in Figure 3.6. Multi-way branches are converted into multiple 2-way branches as implemented in the VHDL compiler.

Representing Loops

Loop boundaries and loop bodies are identified using the special points α and ω of the timing model in DDS, as shown in Figure 3.7 a. α is the initial point of a loop, and ω is the point at which the loop iterates back to α . The α and ω have symbolic



a. A Conditional Branch in the DDS Control and Timing Model

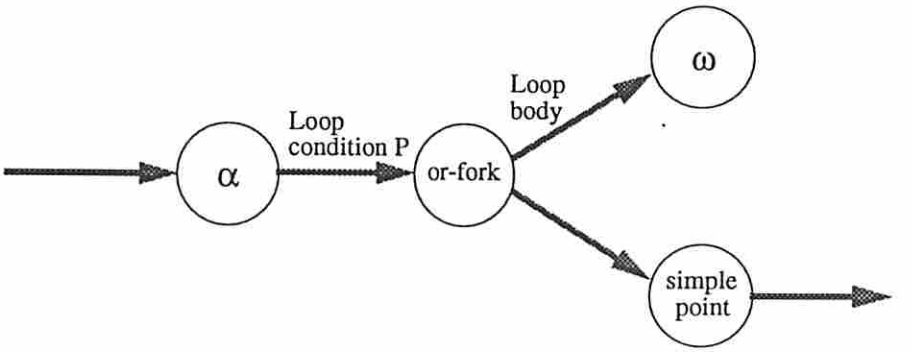


b. A Conditional Branch in the CDFG in SMASH

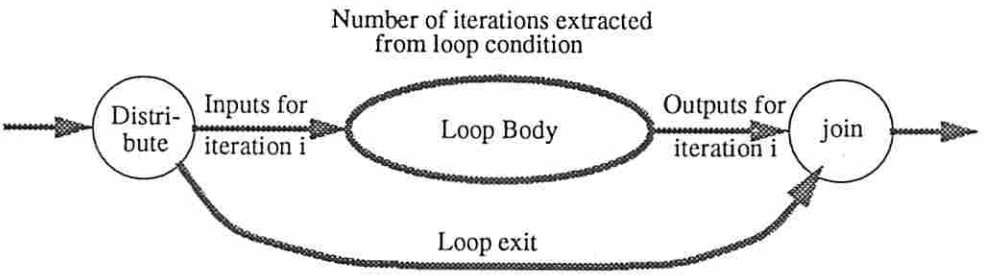
Figure 3.6: Representing Conditional Branches

subscripts which are used in distinguishing values and nodes in different iterations of the loop.

In our CDFG, the loop body is enclosed by a distribute node and a join node. One branch between the distribute-join pair contains the loop body, whereas the other branch is the exit branch, as shown in Figure 3.7 b. The number of iterations for the loop is extracted from the loop condition.



a. Representing a Loop in DDS timing model



b. Representing a Loop in CDFG in SMASH

Figure 3.7: Representing Loops

Representing Inputs and Outputs

The input and output variables (arrays or scalars) are explicitly specified in the VHDL description of the target system. Corresponding to each access of the input

and output variables, a *Read* or *Write* node is introduced in the CDFG, as described below. The set of input and output nodes are denoted by IP and OP respectively.

A data variable can be of two types distinguished by the type of addressing scheme used to access it: deterministic addressable or non-deterministic addressable.

Definition 3.5.6 *Deterministic Addressable:* the address of the data is constant or can be determined at VHDL compile time. Such data can be treated on an individual basis and can be assigned to storage locations independent of other data as long as we provide the required interconnections.

Definition 3.5.7 *Nondeterministic Addressable:* The address is a variable or is data dependent and so cannot be determined *a priori*. The address refers to a specific location in an array. In such a case the whole array is considered as one entity rather than treating each value individually. The whole array will be made accessible to the datapath as per requirements and will be mapped to one or more storage modules in the I/O buffers depending on its size and the number of variables that have to be accessed simultaneously. The required data will be accessed immediately after the address has been determined.

Every input read in the CDFG is represented by a *Read* node and every output write by a *Write* node as illustrated in Figure 3.8. The read node consists of (i) array A (which is being read), and (ii) indices $i_1, i_2, \ldots, i_n$ (the "address" of the value being read) as input edges and value $A[i_1, i_2, \ldots, i_n]$ as the output edge. Similarly, the write node has (i) array A, (ii) indices $i_1, i_2, \ldots, i_n$, and (iii) the value v to be written as input edges and the modified array A' with $A'[i_1, i_2, \ldots, i_n] = v$ as the output edge. The number of indices in both the read node and write node is variable and depends on the dimension of the array being accessed, instead of assigning just one input edge corresponding to the "address" of the referenced value. This is done to preserve the behavior of the array representation. Assigning just one edge would require precomputation of the absolute address of the value in the array, which would implicitly indicate a specific arrangement within the array. A single data value is represented as an array of one element with the index input being a constant zero.

Each input/output array (or scalar) has a *size* associated with it.

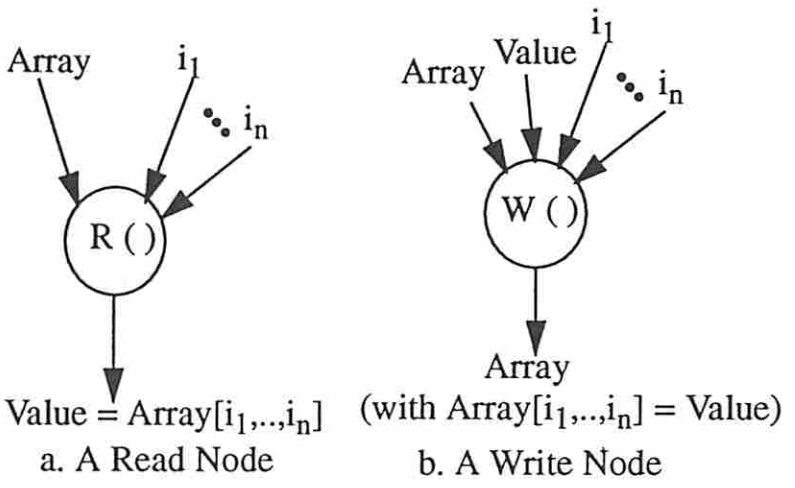


Figure 3.8: Read/Write Nodes in SMASH

Definition 3.5.8 *The size of an input or output array* $SZ(a), a \in IP \cup OP$ *(where* IP *is the set of inputs and* OP *is the set of outputs) is the number of data elements in* a*. The size of a scalar input is always 1.*

Each input/output variable may have an optional timing constraint. A timing constraint on input *ip* implies that *ip* is available for processing only after time step s. Since, the input timing information is usually available in terms of real time, we require the user to provide the clock cycle length and the program computes the corresponding step s for each input. The description format is described in Chapter 5.

This CDFG then can be transformed and optimized as suggested by many researchers [FBS+93, LvMvdW+91, WL91]. Various transformations can be applied to optimize the loop structure, arrays and indexed references. However, these transformations require extensive research in themselves and will be implemented in other USC tools.

3.5.2 Datapath Synthesis with Storage Tradeoffs

The datapath synthesis step is the first major step in the synthesis process. During this step datapath operations are scheduled combined with I/O transfer scheduling

between I/O buffers of the foreground memory and the datapath, while looking ahead in storage structure tradeoffs.

In general, datapath synthesis consists of two fundamental steps:

- Datapath scheduling: In this step, each operation in the CDFG is assigned to a control step, and
- Module allocation and binding: Here the number of functional units to be used in the design is determined and then each operation in the CDFG is mapped into a functional unit.

Datapath Scheduling

This step determines the serial/parallel tradeoffs of the design, which result in the area-performance tradeoff. For example, if 'n' add operations are scheduled in a single step during scheduling then it implies that the final design will have at least 'n' adders.

In our application this step is even more important because the scheduled CDFG also dictates the access-pattern for inputs and outputs by the datapath, which in turn determines the number of read and write ports on the buffers, and influences the buffer size. For example, if the scheduled operations access 'n' inputs in a single step then the final design must have at least 'n' ports on the input buffers so that the datapath can access these 'n' inputs in one step. In other words, the datapath scheduling step imposes constraints on the storage structure. Later, during storage synthesis, these constraints must be met by the synthesized storage structure. Furthermore, the constraints generated for the storage structure must not be very stringent, otherwise the storage synthesis software might not be able to satisfy the given cost constrains and would fail in the future. Therefore, in our approach the storage-related parameters are considered during datapath scheduling as follows:

- The scheduling technique combines data scheduling with scheduling of I/O transfers between the datapath and the I/O buffers so that the number of ports on the storage structure under synthesis is acceptable.

- The scheduling technique looks ahead into data prefetching requirements during scheduling, so that the constraints for the on-off chip bandwidth are not violated in the future.

Decisions regarding the tradeoffs in storage architecture must also be made during datapath synthesis. If these decisions were postponed until the storage synthesis step, we might have had to alter the datapath schedule in order to accommodate storage architecture tradeoffs. This could have resulted in complex backtracking and iteration. The storage architecture tradeoffs are described in detail in Chapter 5.

Many decisions made during this step have great impact on the final design. Therefore, it is necessary to evaluate the impact of these decisions on the final design prior to complete synthesis. This is achieved through estimation of design parameters for storage architecture as well as datapath. These techniques are explained later in Chapter 4.

The result of the above approach is a schedule which guarantees that no bandwidth or I/O timing constraint is violated in the next synthesis step when the complete I/O transfer schedule between the foreground and background memory is determined. This is how the whole synthesis process is tied together in SMASH.

Module Allocation and Binding

For module allocation and binding a decision was made to use existing ADAM tools. These tasks are completed using MABAL [KP90]. Unfortunately, MABAL is unable to handle loops, arrays and multicycle operations in the design specification. Therefore, the designs we synthesize using MABAL do not contain loops. Multicycle operations are taken care of by explicitly specifying module bindings to MABAL for these operations.

After datapath synthesis, the exact I/O access pattern is known, we know exactly when each input is required by the datapath and also when each output is produced by the datapath. Our next step is to construct a cost effective storage structure from the above access pattern that will support the datapath.

3.5.3 Storage Architecture Synthesis

As was described earlier, our target system consists of two levels of hierarchy in the storage architecture: (i) on-chip foreground memory (I/O buffers and datapath memory), and (ii) off-chip background memory. To reduce the complexity of the problem, these hierarchies are synthesized separately. Dividing the storage architecture also helps us decide the order in which various data requirements should be considered. Since the datapath has already been scheduled in the previous step, the data requirements of the datapath in each control step are now known deterministically. The foreground memory, which directly interacts with the datapath, can be synthesized. (Within foreground memory, the order between I/O buffer synthesis and datapath memory synthesis is arbitrary.) Following the synthesis of foreground memory, data writes in the background memory can be scheduled and its synthesis can be completed.

Like datapath synthesis, I/O buffers and off-chip memory are also synthesized in two basic steps:

- data transfer scheduling, where the read and write times of each word are determined; and
- module allocation, where a physical location is assigned to each word.

3.5.3.1 Data Transfer Scheduling

There is a finite interval from the production of a data value to its consumption. The inputs required by the datapath can be transferred to the buffers from the background memory and stored locally in the buffers in any time step before they are consumed by the datapath. Similarly, the outputs produced by the datapath can be stored locally in the buffers and transferred to the background memory in any time step before they are required elsewhere. Both storing a value in the buffers and transferring it into or from buffers require resources: a memory location to store a value and a port to access it. In fact, the size and port requirements of the storage structure depend on the data transfer schedule.

The data transfer scheduling step determines a definite schedule for the transfer of all the data values to and from the storage sub-architecture being designed which

have not been previously scheduled. This enables us to know the exact storage, port and interconnect requirements and also will help us in constructing the address generator.

Before formulating the problem mathematically, we define the following terms in this context. For any data value the following time points are very important (they are given with respect to the storage module, m_1, being constructed):

Definition 3.5.9 *The Birth time, $\mathcal{T}_b^{m_1}(d)$, is defined as the time point when the data d is made available for storage in module m_1.*

Definition 3.5.10 *The Write time, $\mathcal{T}_w^{m_1}(d)$, is the time when d is written into the storage module m_1.*

For example, $\mathcal{T}_w^{buf}(d)$ represent the time point when d is written into the I/O buffers from either the background memory or the datapath. Note that the data d may be written multiple times into storage module m_1, in such a case $\mathcal{T}_w^{m_1}(d)$ will have multiple values associated with it corresponding to each write time.

Definition 3.5.11 *The Read time, $\mathcal{T}_r^{m_1}(d)$, is the time when d is read from the storage module m_1. Note that the data d may be read multiple times from m_1, in such a case $\mathcal{T}_w^{m_1}(d)$ will have multiple values associated with it corresponding to each read time.*

Definition 3.5.12 *The Death time, $\mathcal{T}_d^{m_1}(d)$, is when d dies or is no longer available.*

These definitions are illustrated in Figure 3.9.

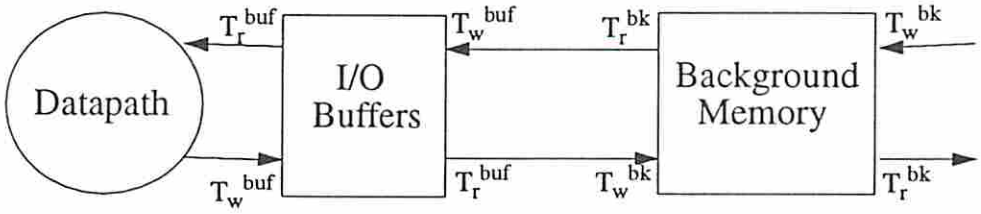


Figure 3.9: Read and Write Timing for Data Values

When the data is being transferred in real-time from the outside world to the background memory, the birth time and the write time are going to be the same, as

the data is written in to the background memory as soon as it is available. When the data is being transferred from the background memory to an I/O buffer, the birth time is the time when it is written into the background memory from the external world, and write time $\mathcal{T}_w^{buf}$ is the time when it is actually written into the I/O buffer from the background memory. It can be written into the buffer any time after its birth time (or after it is written in the background memory from the external world).

For some data values the write time is fixed. For example, in our case, the output write time into the buffers by the datapath is fixed once the datapath scheduling is done. For some data values the read time is fixed; for example, after the datapath scheduling, the input read times by the datapath from the I/O buffers are fixed.

The birth time and the death time for all the values are either specified *a priori* by the external timing constraints or assigned the execution start time and the execution end time respectively.

Some inherent constraints, called the *timing constraints*, for these time points are given below. These constraints are with respect to the storage structure (I/O buffer or off-chip memory) being considered:

- While constructing the I/O buffers, the write time of a data value is the time when the data value is written into the I/O buffers from the background memory (for the inputs) or by the datapath (for the outputs). Similarly, the read time of a data value is the time when the value is read from the buffer by the datapath (for the inputs) or the background memory (for the outputs).

- While constructing the off-chip background memory, the write time of data value is the time when the data value is written into the background memory from the external world (for the inputs) or from the I/O buffers (for the outputs). The read time is the time when the data value is read from the background memory by the I/O buffers (for the inputs) or the external world (for the outputs).

  The timing constraints are as follows:

1. $\mathcal{T}_b^{m_1}(d) \leq \mathcal{T}_w^{m_1}(d)$. The data value cannot be written into the storage module m_1 before it is made available to m_1.

2. $\mathcal{T}_w^{m_1}(d) \leq \mathcal{T}_r^{m_1}(d)$. The write and read times are the times when the data is written into and read from m_1 respectively. The equality in the constraint depends on the situation. The data value can be written and read in the same step only with a two-phase clocking scheme. Strictly speaking, a write must be done before a read.

3. $\mathcal{T}_r^{m_1}(d) \leq \mathcal{T}_d^{m_1}(d)$. The value cannot be read from m_1 after it is not available.

As mentioned earlier $\mathcal{T}_b^{m_1}(d)$ and $\mathcal{T}_d^{m_1}(d)$ are specified prior to scheduling unless a value is created or used by an operation not yet scheduled. However, $\mathcal{T}_w^{m_1}(d)$ and/or $\mathcal{T}_r^{m_1}(d)$ may not be specified at that time. The goal in the data transfer scheduling step is to determine $\mathcal{T}_w^{m_1}(d)$ and $\mathcal{T}_r^{m_1}(d)$.

The port constraints that must be satisfied are: $\mathcal{P}_w^{m_1}$ is the number of write ports on storage structure m_1, $\mathcal{P}_{rw}^{m_1}$ number of read-write ports on m_1, and $\mathcal{P}_r^{m_1}$ is the number of read ports on m_1. $\mathcal{N}_w^{m_1}(s)$ is the number of all the data values being written in m_1 in step s, i.e.

$$\mathcal{N}_w^{m_1}(s) = |\{d\}| \ \ \forall d \text{ s.t. } \mathcal{T}_w^{m_1}(d) = s$$

and $\mathcal{N}_r^{m_1}(s)$ is the number of data values being read from m_1 in step s,

$$\mathcal{N}_r^{m_1}(s) = |\{d\}| \ \ \forall d \text{ s.t. } \mathcal{T}_r^{m_1}(d) = s$$

Then, assuming a **two phase clocking** scheme with the reads and writes being performed in alternate clock phases, so that we can use $\mathcal{P}_{rw}^{m_1}$ for reads in one phase and for writes in the other phase, the following three inequalities hold:

$$\begin{aligned}
\mathcal{P}_w^{m_1} + \mathcal{P}_{rw}^{m_1} &\geq \mathcal{N}_w^{m_1}(s) && \forall s \\
\mathcal{P}_r^{m_1} + \mathcal{P}_{rw}^{m_1} &\geq \mathcal{N}_r^{m_1}(s) && \forall s \\
\mathcal{P}_w^{m_1} + \mathcal{P}_{rw}^{m_1} + \mathcal{P}_r^{m_1} &\geq \mathcal{N}_w^{m_1}(s) + \mathcal{N}_r^{m_1}(s) && \forall s
\end{aligned}$$

Given the read and write times of all the data values stored in a module, a lower bound on the size of that module is given by

$$\mathcal{SM}_{m_1} = \max_{\forall s}\{|\{d\}| \text{ s.t. } \mathcal{T}_w^{m_1}(d) \leq s \leq \mathcal{T}_r^{m_1}(d), \forall d\}$$

Note that this is a lower bound because we may read the same value into the storage module more than once in order to reuse the memory locations. Nevertheless, this relationship indicates that the size of a storage module depends on the read and write timings of the data values stored in that module.

Next, we present our objective in this step. Given the number of ports $\mathcal{P}_r^{m_1}$, and $\mathcal{P}_w^{m_1}$ that can be used to read and write in one step[1], the birth time $\mathcal{T}_b^{m_1}(d)$ and the death time $\mathcal{T}_d^{m_1}(d)$ of the data d, our goal is to determine a transfer schedule, read time $\mathcal{T}_r^{m_1}(d)$ and write time $\mathcal{T}_w^{m_1}(d)$ for all d, such that the size of the memory $\mathcal{SM}_{m_1}$, is minimized, and the port and timing constraints are met.

The details of the data transfer algorithm and its implementation in SMASH are described in Chapter 6.

3.5.3.2 Module Allocation

After every data transfer has been scheduled and various requirements at each step are known we should allocate each value to a physical storage location. The objective here is to optimize the total storage area, meeting all the requirements. This step is not covered in this dissertation but will be addressed in the future.

3.5.4 RTL Synthesis

The next and final step in the overall synthesis process is the register-transfer level (RTL) synthesis. This includes mapping the scheduled CDFG onto an RTL netlist and generating the layout of the netlist. We used existing tools (commercial and ADAM) to assist us in this step instead of developing our own tools.

3.5.4.1 MABAL and Epoch

In this pathway, MABAL [KP90] is used for module allocation, binding and completing the RTL netlist, and Epoch (a commercial silicon compiler from Cascade Design Automation Corporation) [Cas93] is used for layout generation. MABAL accepts the datapath schedule along with other relevant information such as the CDFG and module library, and generates the RTL netlist of the design. This netlist is then

[1] $\mathcal{P}_{rw}^{m_1}$ is assumed to be zero in the current implementation of SMASH.

translated into Epoch's netlist format using the MABAL2Epoch netlist translator. Finally, the layout of the design is generated using Epoch.

SMASH was quickly interfaced with these tools as all the required interfaces for these tools were already in existence. Several designs were generated using this pathway. These designs are presented in Chapter 7. As mentioned above, MABAL is does not handle loops, arrays and multicycle operations in the design specification. Therefore, the layouts of designs with loops cannot be generated currently. However, designs with multicycle operations can be handled by explicitly specifying module bindings to MABAL for these operations. We are looking into another commercial RTL synthesis tool called DPSYN from COMPASS Design Automation. This option is planned for future research and is described in Chapter 8.

3.6 Summary

In this chapter we have outlined our approach to solving the high-level synthesis problem for memory-intensive systems. We presented the target architecture produced by SMASH, which consists of a datapath and a two-level memory hierarchy. Next, we described the characteristics of the module library used by SMASH. The module library includes functional modules as well as storage modules. We also briefly described the clocking scheme assumed in SMASH. Finally, we outlined all the major steps of the overall synthesis process. These steps are

1. CDFG extraction,
2. datapath synthesis,
3. storage architecture synthesis, and
4. RTL synthesis.

The next chapter describes the estimation techniques used in the datapath synthesis step. The details of datapath synthesis step and the storage architecture synthesis step are presented in Chapters 5 and 6 respectively.

Chapter 4

Estimation Techniques

4.1 Introduction

As discussed earlier, high-level synthesis is known to contain NP-complete problems [SJ94]. Searching the design space for an optimal solution is a complex and time consuming process. Therefore, most synthesis algorithms are based on heuristics, searching for near-optimal solutions. The use of heuristics makes it necessary to evaluate the decisions made using the heuristics during the synthesis process. Lower and upper bounds on the cost can help us in doing that. Specifically, these bounds can be used in

1. guiding the user (or other software tools) in design space search by quickly providing useful design parameters without synthesizing the design,
2. evaluating the impact of high-level decisions on the final design at an early design stage without going through the complete synthesis process,
3. speeding up the design space search during synthesis, and
4. evaluating the quality of a design produced by the heuristic.

A lower and upper bound are computed at the beginning of the synthesis process. Later, as the synthesis proceeds, these bounds are updated and analyzed to evaluate the impact of the decisions made on the final designs. If the estimated cost indicates a violation of the cost constraints in the future then that decision is discarded.

The total cost of the system consists of various components *viz.* functional cost, storage cost, interconnection cost, and controller cost. There has been a lot of

research in estimating some of the components like functional cost and controller cost but not much research on estimating the storage cost. In this research we have developed techniques to estimate the storage cost. Storage cost estimation combined with functional cost estimation which is based on existing techniques has been incorporated in SMASH.

In this chapter we will describe the methodology to estimate the storage and functional cost of the final implementation from the partial operation schedule. The overall estimation step is divided into two parts: (i) storage cost estimation, and (ii) functional cost estimation.

We begin with some definitions which will be used in this chapter.

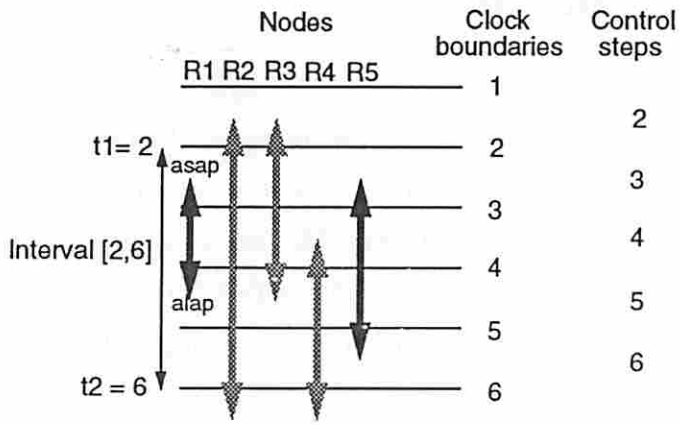


Figure 4.1: ASAP and ALAP Times for Various Read Nodes

Definition 4.1.13 $[t1, t2]$ *is defined as an interval between clock boundaries* t_1 *and* t_2, *where* $0 \leq t_1 < t_2 \leq MaxSteps$ *and* $MaxSteps$ *is the number of control steps required in executing the CDFG. Equivalently,* $[t1, t2]$ *is the sequence of control steps* $\{s_1 + 1, \ldots, s_2 - 1, s_2\}$.

For example, in Figure 4.1 $[2, 6]$ is the interval between clock boundaries $t_1 = 2$ and $t_2 = 6$ and includes control steps 3, 4, 5, and 6.

Observation 4.1.1 *A node* v *representing operation* O_k *with delay* D_{O_k} must be *scheduled in the interval* $[t_1, t_2]$, *if* $t_1 < ASAP(v) + D_{O_k} \leq ALAP(v) \leq t_2$. $ASAP(v)$ *is the As Soon As Possible time for* v, D_{O_k} *is the delay of operator* O_k, *and* $ALAP(v)$ *is the As Late As Possible time for* v.

In the example (Figure 4.1), R1 and R5 must be scheduled in the intervals $[2,5]$ and $[2,6]$ respectively. Note that R1 must also be included when considering the interval $[2,6]$.

Definition 4.1.14 *$L_{O_k,t_1,t_2} \subseteq V_{O_k}$ is defined as a set of nodes such that the node $v \in V_{O_k}$ must be scheduled in the interval $[t_1,t_2]$. Using observation 4.1.1,*

$$L_{O_k,t_1,t_2} = \{v \mid v \in V_{O_k} \text{ and } t_1 < ASAP(v) + D_{O_k} \leq ALAP(v) \leq t_2\}$$

In the example (Figure 4.1), $L_{R,2,6} = \{R1, R5\}$.

Observation 4.1.2 *A node v representing operation O_k with delay D_{O_k} may be scheduled in the interval $[t_1,t_2]$, if*

$$\begin{aligned} ASAP(v) \leq\ & t_1 & \leq ALAP(v) + D_{O_k},\ or \\ ASAP(v) \leq\ & t_2 & \leq ALAP(v) + D_{O_k},\ or \\ t_1 \leq\ & ASAP(v) & \leq ALAP(v) + D_{O_k} \leq t_2 \end{aligned}$$

Example: In Figure 4.1, all of the 5 read nodes R1, R2, ..., R5 may be scheduled between clock boundaries 2 and 6.

Definition 4.1.15 *$U_{O_k,t_1,t_2} \subseteq V_{O_k}$ is defined as a set of nodes such that the node $v \in V_{O_k}$ may be scheduled in the interval $[t_1,t_2]$. Using observation 4.1.2,*

$$U_{O_k,t_1,t_2} = \{v \mid v \in V_{O_k} \text{ and any of the 3 conditions given in observation 4.1.2 is satisfied.}\}$$

In the example (Figure 4.1), $U_{R,2,6} = \{R1, R2, R3, R4, R5\}$.

4.2 Storage Cost Estimation

Estimated cost of the storage (buffers) is used in the design process to avoid violating the area constraint in the final design. The storage cost of a design is the cost of constructing the storage structure such that it has the number of ports and size

required by the final implementation. The construction must be done using modules from the given storage library.

During the datapath synthesis, estimated storage cost is included in the total estimated area. This enables us to avoid datapaths at an early stage, which may violate the area constraints in the future. Without any knowledge of the storage cost, a datapath that requires unacceptably large storage in the final implementation might be selected at this point but eventually would have to be rejected.

The storage cost estimation consists of the following three steps:

1. determining a lower bound on the number of read and write ports on the buffers.

2. determining a lower bound on the total size of all the buffers, and

3. meeting these requirements with the lowest-cost storage modules from the library.

4.2.1 Lower Bound on Read (Write) Ports on Buffers

A lower bound on the number of read (R) ports on the buffers, $N_{LB}(R)$, is the minimum number of read ports that are needed on the buffers in the final design. Actually, $N_{LB}(R)$ is nothing but the maximum number of inputs that must be accessed in a step. Since, in our approach each read operation (input access) is represented by a read node, $N_{LB}(R)$ can be determined the same way that the lower bound for functional operators is determined. In fact, our techniques are based on the techniques developed to obtain the lower bound for functional operators [SJ94, JMP88, JPP87, Kuc91].

Basically, we count the number of reads that must be scheduled only during a specified interval, using the ASAP and ALAP analysis of the read nodes as the actual scheduling has not been done yet. We assume that these reads will be uniformly distributed over that interval because any other distribution will result in greater number of reads in at least one control step than the average. By performing this analysis over all the possible intervals and then determining the maximum number of reads that must be performed in any interval, we determine the desired lower bound.

Theorem 4.2.1 *A lower bound on the number of read (R) ports on the buffers, $N_{LB}(R)$, required in executing the CDFG in MaxSteps is*

$$N_{LB}(R) = \max_{\forall[t_1,t_2]} \left\{ \left\lceil \frac{1}{(t_2 - t_1)} |L_{R,t_1,t_2}| \right\rceil \right\}$$

where $t1 = 0 \ldots (MaxSteps - 1)$,

$t_2 = (t_1 + 1), (t_1 + 2), \ldots, MaxSteps$, *and*

L_{R,t_1,t_2} *is as per Definition 4.1.14.*

Proof: Consider an interval $[t_1, t_2]$ and the set of read nodes V_R. A lower bound on the number of read nodes $n(R, t1, t2)$ that must be scheduled in the interval $[t_1, t_2]$ is at least equal to the cardinality of the set of read nodes $v \in V_R$ such that v must be scheduled in $[t_1, t_2]$. Using Observation 4.1.1,

$$\begin{aligned} n(R, t1, t2) &= |\{v \mid v \in V_R \text{ and } t_1 < ASAP(v) \leq ALAP(v) \leq t_2\}| \\ &= |L_{R,t_1,t_2}| \end{aligned}$$

Our objective is to determine the minimum number of read ports $n_{lb}(R, t1, t2)$ that are required to access these inputs (i.e. perform the read operation) in the interval $[t_1, t_2]$, therefore we must assume uniform distribution of these nodes over the interval $[t_1, t_2]$. A uniform distribution results in an average number of input reads in a control step, whereas any other distribution will result in more number of input reads than the average in at least one control step. Therefore, the maximum of input reads in a single control step for the interval $[t_1, t_2]$ (which is the lower bound on the number of read ports required for that interval) is minimum when the distribution is uniform. With that assumption we get

$$\begin{aligned} n_{lb}(R, t1, t2) &= \left\lceil \frac{n(R, t1, t2)}{(t_2 - t_1)} \right\rceil \\ &= \left\lceil \frac{|L_{R,t_1,t_2}|}{(t_2 - t_1)} \right\rceil \end{aligned} \tag{4.1}$$

The maximum taken over all possible intervals $[t_1, t_2]$ gives the minimum number of read ports required in the final design:

$$\begin{aligned} N_{LB}(R) &= \max\{n_{lb}(R, t1, t2)\} \\ &\quad \text{where } t1 = 0 \ldots (MaxSteps - 1) \text{ and} \\ &\quad t_2 = (t_1 + 1) \ldots MaxSteps \end{aligned} \tag{4.2}$$

From Equations 4.1 and 4.2 we get the desired lower bound. □

A lower bound on the number of write ports on the buffers is determined in a similar manner.

Theorem 4.2.2 *A lower bound on the number of write ports on the buffers, $N_{LB}(W)$, required in executing the CDFG in $MaxSteps$ is*

$$\begin{aligned} N_{LB}(W) &= \max\left\{\left\lceil \frac{1}{(t_2 - t_1)} |L_{W,t_1,t_2}| \right\rceil\right\} \\ &\quad \textit{where } t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps, \textit{ and} \\ &\quad L_{W,t_1,t_2} \textit{ is as per definition 4.1.14.} \end{aligned}$$

Proof: Similar to the above proof.

4.2.1.1 Computational Complexity

Computing $n(R, t_1, t_2)$ for each interval $[t_1, t_2]$ is done by processing all the elements of the set V_R, which requires $O(|V_R|)$ computations. This analysis is done for all the possible intervals *i.e.* it is done $O(MaxSteps^2)$ times. Therefore, the overall complexity of determining the lower bound on the read ports is $O(MaxSteps^2.|V_R|)$. Similarly, the complexity of determining the lower bound on the write ports is $O(MaxSteps^2 \cdot |V_W|)$.

4.2.2 Buffer Size Estimation

The next step in estimating the cost of the storage structure is to estimate the total size of all the buffers. The problem is as follows: given a CDFG and I/O bandwidth

(BW_{on-off}) on the chip, the problem is to determine a lower bound on the total size of all the I/O buffers $BufSize_{LB}$.

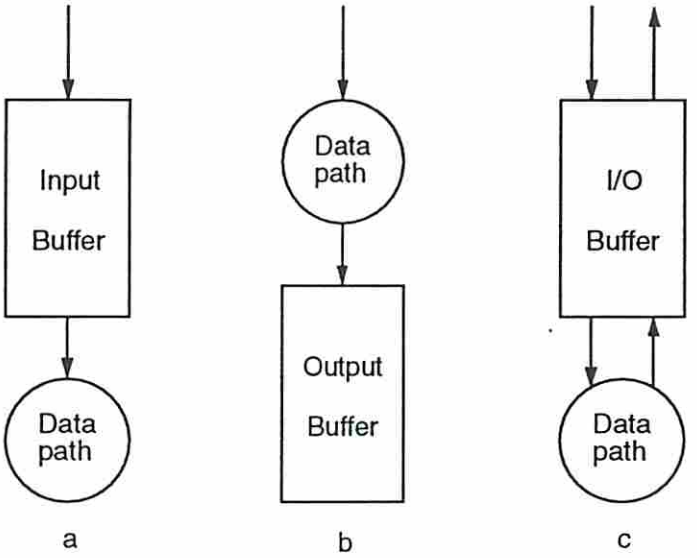


Figure 4.2: Buffer Configurations

Though in our model the buffers store both inputs and outputs, we have developed a lower-bound estimation theory for the following three possible configurations of the buffers:

1. input only buffer (Figure 4.2 a),
2. output only buffer (Figure 4.2 b), and
3. input as well as output buffer (Figure 4.2 c).

A lower bound on the total size of the I/O buffers is the smallest buffer size which is able to support the I/O transfer performed concurrently with the datapath execution, using the given bandwidth BW_{on-off}. In other words, it is the smallest I/O buffer required to store

1. all the inputs that are prefetched before they are required by the datapath, and/or
2. all the outputs produced by the datapath before they are transferred back to the off-chip memory.

Before presenting the estimation techniques we define some terms.

Definition 4.2.16 *$L_{I,t_1,t_2} \subseteq I$ is defined as a set of inputs such that the input $i \in L_{I,t_1,t_2}$* must be *accessed in the interval $[t_1, t_2]$. Using Observation 4.1.1,*

$$L_{I,t_1,t_2} = \{i \mid i \in Pred(v) \text{ and } v \in L_{R,t_1,t_2}\}$$
$$\text{where } L_{R,t_1,t_2} = \{v \mid v \in V_R \text{ and } t_1 < ASAP(v) \leq ALAP(v) \leq t_2\}$$

Definition 4.2.17 *$U_{I,t_1,t_2} \subseteq I$ is defined as a set of inputs such that the input i* may be *accessed in the interval $[t_1, t_2]$. Using Observation 4.1.2,*

$$U_{I,t_1,t_2} = \{i \mid i \in Pred(v) \text{ and } v \in U_{R,t_1,t_2}\}$$
$$\text{where } U_{R,t_1,t_2} = \{v \mid v \in V_R \text{and}$$
$$\text{any of the 3 conditions given in}$$
$$\text{Observation 4.1.2 is satisfied.}\}$$

Definition 4.2.18 *$L_{O,t_1,t_2} \subseteq O$ is defined as a set of outputs such that the output $o \in L_{O,t_1,t_2}$* must be *produced in the interval $[t_1, t_2]$. Using Observation 4.1.1,*

$$L_{O,t_1,t_2} = \{o \mid o \in Succ(v) \text{ and } v \in L_{W,t_1,t_2}\}$$
$$\text{where } L_{W,t_1,t_2} = \{v \mid v \in V_W \text{ and } t_1 < ASAP(v) \leq ALAP(v) \leq t_2\}$$

Definition 4.2.19 *$U_{O,t_1,t_2} \subseteq O$ is defined as a set of outputs such that the output o* may be *produced in the interval $[t_1, t_2]$. Using Observation 4.1.2,*

$$U_{O,t_1,t_2} = \{o \mid o \in Succ(v) \text{ and } v \in U_{W,t_1,t_2}\}$$
$$\text{where } U_{W,t_1,t_2} = \{v \mid v \in V_W \text{and}$$
$$\text{any of the 3 conditions given in}$$
$$\text{Observation 4.1.2 is satisfied.}\}$$

Definition 4.2.20 *$I_{acc}(t_1, t_2)$ is the number of inputs that* must be *accessed during $[t_1, t_2]$ (Figure 4.3). $O_{prod}(t_1, t_2)$ is the number of outputs that* must be *produced during $[t_1, t_2]$ (Figure 4.4).*

Definition 4.2.21 *$I_{st}(t_1, t_2)$ is the number of inputs that are prefetched and stored in the buffers before t_1 to support the processing in $[t_1, t_2]$ (Figure 4.3). Similarly, $O_{st}(t_1, t_2)$ is the number of outputs that are stored in the buffers after t_2 (Figure 4.4).*

Definition 4.2.22 *$I_{trans}(t_1, t_2)$ is the number of inputs and $O_{trans}(t_1, t_2)$ is the number of outputs that are transferred during $[t_1, t_2]$ as illustrated in Figures 4.3 and 4.4 respectively.*

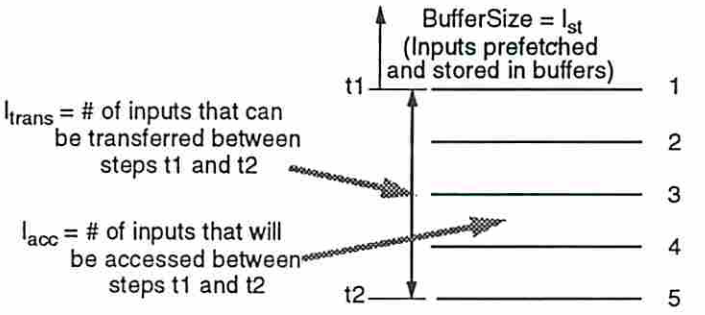


Figure 4.3: Lower Bound Estimation for Input Buffers

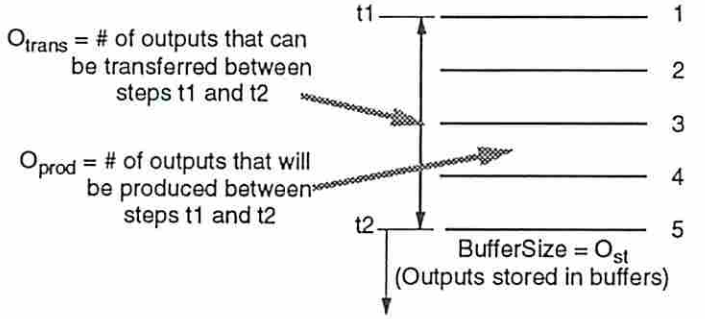


Figure 4.4: Lower Bound Estimation for Output Buffers

We start with estimation of the lower bounds for the first two cases in order to understand the idea behind the theory, then develop the estimates for the third configuration, although the third configuration subsumes the first and second configurations, and is the one which is actually used in SMASH.

4.2.3 Lower Bound on the Size of the Input Only Buffer

When the input buffers are separate from the output buffers, a lower bound for the total size of all the input buffers can be computed using the following theorem.

Theorem 4.2.3 *A lower bound on the size of the input buffers, $BufSize_{LB}(I)$, required in executing the CDFG in $MaxSteps$ is given by*

$$BufSize_{LB}(I) = \max\{(|L_{I,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1)), 0\} \quad \forall \text{ intervals } [t_1, t_2]$$
$$\text{where } t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps, \text{ and}$$
$$BW_{on-off} \text{ is the bandwidth between on and off chip memory.}$$

Proof: Consider an interval $[t_1, t_2]$. The size of the input buffer to support the processing in $[t_1, t_2]$ is the number of inputs that should have been prefetched and stored into the buffers prior to this interval, *i.e.* $I_{st}(t_1, t_2)$. Now, $I_{st}(t_1, t_2)$ is the difference between $I_{acc}(t_1, t_2)$ and $I_{trans}(t_1, t_2)$ (assuming that the inputs transferred during $[t_1, t_2]$ are not stored in the buffer, and therefore do not contribute to the buffer size), *i.e.*

$$I_{st}(t_1, t_2) = I_{acc}(t_1, t_2) - I_{trans}(t_1, t_2)$$

$I_{acc}(t_1, t_2)$ is the cardinality of the set L_{I,t_1,t_2}, *i.e.*

$$I_{acc}(t_1, t_2) = |L_{I,t_1,t_2}|$$

$I_{trans}(t_1, t_2)$ is the number of inputs that can be transferred during $[t_1, t_2]$ using a bandwidth of BW_{on-off} which can be determined by considering the following two cases:

- Case(i): $I_{acc}(t_1, t_2) \geq BW_{on-off} \cdot (t_2 - t_1)$. In this case, we assume the maximum utilization of the bandwidth because we are interested in the lower bound, *i.e.*

$$I_{trans}(t_1, t_2) = BW_{on-off} \cdot (t_2 - t_1)$$

- case(ii): $I_{acc}(t_1, t_2) < BW_{on-off} \cdot (t_2 - t_1)$. In this case, all the accessed inputs can be transferred in the interval $[t_1, t_2]$, *i.e.*

$$I_{trans}(t_1, t_2) = I_{acc}(t_1, t_2)$$

All the remaining inputs should have been prefetched and stored into the buffers prior to this interval. Therefore, $BS_{lb}(I, t_1, t_2)$, the minimum size of the buffer required to support the processing in $[t_1, t_2]$ is

$$\begin{aligned}
&BS_{lb}(I, t_1, t_2) \\
&= I_{st}(t_1, t_2) \\
&= I_{acc}(t_1, t_2) - I_{trans}(t_1, t_2) \\
&= \begin{cases} I_{acc}(t_1, t_2) - BW_{on-off} \cdot (t_2 - t_1) & \text{if } I_{acc}(t_1, t_2) \geq BW_{on-off} \cdot (t_2 - t_1) \\ 0 & \text{otherwise} \end{cases} \\
&= \max\{(I_{acc} - BW_{on-off} \cdot (t_2 - t_1)), 0\} \\
&= \max\{(|L_{I,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1)), 0\} \qquad (4.3)
\end{aligned}$$

The maximum taken over all the possible intervals $[t_1, t_2]$ gives the minimum size of the inputs buffer required in the final design.

$$\begin{aligned}
BufSize_{LB}(I) &= \max\{BS_{lb}(I, t1, t2)\} \qquad (4.4) \\
&\text{where } t1 = 0 \ldots (MaxSteps - 1) \text{ and } t_2 = (t_1 + 1) \ldots MaxSteps
\end{aligned}$$

From 4.3 and 4.4 we get the desired lower bound. □

Theorem 4.2.4 *A lower bound on the size of the output buffers, $BufSize_{LB}(O)$, required in executing the CDFG in $MaxSteps$ is given by*

$$\begin{aligned}
BufSize_{LB}(O) &= \max\{(|L_{O,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1)), 0\} \\
&\textit{where } t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps, \textit{ and} \\
&BW_{on-off} \textit{ is the bandwidth between on and off chip memory}
\end{aligned}$$

Proof: Similar to the above proof.

4.2.3.1 Computational Complexity

The computational complexity of determining the lower bound of the input-only buffer is $O(MaxSteps^2 \cdot |V_R|)$: Computing $BS_{lb}(I, t_1, t_2)$ in Equation 4.3 requires linear processing of the read nodes in CDFG, which can be done in $|V_R|$ steps. This analysis must be done for all the possible intervals $[t_1, t_2]$ i.e. $O(MaxSteps^2)$ times. Therefore, the overall complexity of this analysis is $O(MaxSteps^2 \cdot |V_W|)$.

4.2.4 Lower bound on the Size of I/O Buffer

Under the assumption that the same buffers are used to store inputs as well as outputs, the buffer size is estimated using the theorem given later in this section. Prior to presenting the theorem, we present the following lemma which will be used in the proof of the theorem.

Lemma 4.2.1 *$F = max\{A-x, B+x\}$ is minimum when $A-x = B+x$ or $x = \frac{A-B}{2}$*

Proof: Proof follows by considering the following three cases:

$$\begin{aligned}
\text{case(i) } A - x > B + x \text{ or } x < \frac{A-B}{2} &\Rightarrow F > \frac{A+B}{2} \\
\text{case(ii) } A - x < B + x \text{ or } x > \frac{A-B}{2} &\Rightarrow F > \frac{A+B}{2} \\
\text{case(iii) } A - x = B + x \text{ or } x = \frac{A-B}{2} &\Rightarrow F = \frac{A+B}{2}
\end{aligned}$$

Clearly, F is minimum in case(iii). □

Now we present the theorem to determine a lower bound on the total size of all the I/O buffers.

Theorem 4.2.5 *A lower bound on the size of the I/O buffers, $BufSize_{LB}(IO)$, required in executing the CDFG in $MaxSteps$ is given by:*

$$\begin{aligned}
BufSize_{LB}(IO) = \max\{&(|L_{I,t_1,t_2}| - I_{trans}(t_1, t_2)), \\
&(|L_{O,t_1,t_2}| - (BW_{on-off} \cdot (t_2 - t_1)) - I_{trans}(t_1, t_2))\}
\end{aligned}$$

where

$$t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps,$$

BW_{on-off} *is the bandwidth between on and off chip memory, and*

$$I_{trans}(t_1, t_2) =$$
$$\begin{cases} 0 & \text{if } |L_{O,t_1,t_2}| > BW_{on-off} \cdot (t_2 - t_1) + |L_{I,t_1,t_2}| \\ BW_{on-off} \cdot (t_2 - t_1) & \text{if } |L_{I,t_1,t_2}| > BW_{on-off} \cdot (t_2 - t_1) + |L_{O,t_1,t_2}| \\ \frac{BW_{on-off} \cdot (t_2 - t_1) + |L_{I,t_1,t_2}| - |L_{O,t_1,t_2}|}{2} & \text{otherwise} \end{cases}.$$

Proof: For a given interval $[t_1, t_2]$, a lower bound on the buffer size $BS_{lb}(IO, t1, t2)$, is the maximum of (i) $I_{st}(t_1, t_2)$, the number of inputs required to be stored before t_1, and (ii) $O_{st}(t_1, t_2)$, the number of outputs required to be stored after t_2.

$$BS_{lb}(IO, t1, t2) = \max\{I_{st}(t_1, t_2), O_{st}(t_1, t_2)\}$$

We also know that

$$\begin{aligned} I_{st}(t_1, t_2) &= I_{acc}(t_1, t_2) - I_{trans}(t_1, t_2) \\ O_{st}(t_1, t_2) &= O_{prod}(t_1, t_2) - O_{trans}(t_1, t_2) \end{aligned}$$

$I_{acc}(t_1, t_2)$ is the cardinality of the set L_{I,t_1,t_2} and $O_{prod}(t_1, t_2)$ is the cardinality of the set L_{O,t_1,t_2} *i.e.*

$$I_{acc}(t_1, t_2) = |L_{I,t_1,t_2}| \tag{4.5}$$
$$O_{prod}(t_1, t_2) = |L_{O,t_1,t_2}| \tag{4.6}$$

Furthermore, the total number of inputs/outputs that can be transferred from/to the buffers is limited by the bandwidth and the duration of the interval:

$$I_{trans}(t_1, t_2) + O_{trans}(t_1, t_2) \leq BW_{on-off} \cdot (t_2 - t_1)$$

Since we are interested in the lower bound, we consider the case when

$$I_{trans}(t_1, t_2) + O_{trans}(t_1, t_2) = BW_{on-off} \cdot (t_2 - t_1)$$

Therefore

$$\begin{aligned} BS_{lb}(IO, t_1, t_2) &= \max\{I_{st}(t_1, t_2), O_{st}(t_1, t_2)\} \\ &= \max\{(I_{acc}(t_1, t_2) - I_{trans}(t_1, t_2)), (O_{prod}(t_1, t_2) - O_{trans}(t_1, t_2))\} \\ &= \max\{(|L_{I,t_1,t_2}| - I_{trans}(t_1, t_2)), \\ &\quad (|L_{O,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1) + I_{trans}(t_1, t_2))\} \end{aligned} \tag{4.7}$$

$I_{trans}(t_1, t_2)$ is yet to be determined. Since we are interested in the lower bound, the desired $I_{trans}(t_1, t_2)$ is the one which minimizes $BufSize_{lb}(IO, t_1, t_2)$. We know that $BufSize_{lb}(IO, t_1, t_2)$ is minimum when the following condition is true, from Lemma 4.2.1.

$$\begin{aligned} I_{acc}(t_1, t_2) - I_{trans}(t_1, t_2) &= O_{prod}(t_1, t_2) - BW_{on-off} \cdot (t_2 - t_1) + I_{trans}(t_1, t_2) \\ \text{or, } I_{trans}(t_1, t_2) &= \frac{BW_{on-off} \cdot (t_2 - t_1) + I_{acc}(t_1, t_2) - O_{prod}(t_1, t_2)}{2} \end{aligned} \tag{4.8}$$

Furthermore, since

$$0 \leq I_{trans}(t_1, t_2) \leq BW_{on-off} \cdot (t_2 - t_1)$$

$I_{trans}(t_1, t_2)$ must be modified as follows:

- Case(i): $O_{prod}(t_1, t_2) > BW_{on-off} \cdot (t_2 - t_1) + I_{acc}(t_1, t_2)$ In this case we get $I_{trans}(t_1, t_2) < 0$ in Equation 4.8. This implies that $I_{trans}(t_1, t_2)$ should be equal to zero. In other words, in this situation the number of outputs produced is so high that in order to minimize the buffer size, we should not transfer the inputs into the buffer and only transfer the outputs out of the buffer.

- Case(ii): $I_{acc}(t_1, t_2) > BW_{on-off} \cdot (t_2 - t_1) + O_{prod}(t_1, t_2)$. In this case $I_{trans}(t_1, t_2) > BW_{on-off} \cdot (t_2 - t_1)$ in Equation 4.8. Since $I_{trans}(t_1, t_2) \leq BW_{on-off} \cdot (t_2 - t_1)$, therefore $I_{trans}(t_1, t_2)$ is equal to $BW_{on-off} \cdot (t_2 - t_1)$.

- Case(iii): Otherwise, we use Equation 4.8.

In summary

$$I_{trans}(t_1, t_2) = \begin{cases} 0 & \text{if } O_{prod}(t_1, t_2) > BW_{on-off} \cdot (t_2 - t_1) + I_{acc}(t_1, t_2) \\ BW_{on-off} \cdot (t_2 - t_1) & \text{if } I_{acc}(t_1, t_2) > BW_{on-off} \cdot (t_2 - t_1) + O_{prod}(t_1, t_2) \\ \frac{BW_{on-off} \cdot (t_2 - t_1) + I_{acc}(t_1, t_2) - O_{prod}(t_1, t_2)}{2} & \text{otherwise} \end{cases} \tag{4.9}$$

From Equations 4.7 and 4.9 we get the desired result. □

4.2.5 Computational Complexity

The computational complexity of determining the lower bound I/O buffer size is $O(MaxSteps^2 \cdot \max\{|V_R|, |V_W|\})$. Computing $BS_{lb}(IO, t_1, t_2)$ in Equation 4.3 requires linear processing of the read nodes and write nodes in CDFG which can be done in $O(\{|V_R|) + O(\{|V_W|)$ or $O(\max\{|V_R|, |V_W|\})$ steps. This analysis is done for all the possible intervals $[t_1, t_2]$ which takes $O(MaxSteps^2)$ steps. Therefore, the overall complexity of this analysis is $O(MaxSteps^2 \cdot \max\{|V_R|, |V_W|\})$.

4.2.6 Storage Structure Construction

The third and final step in the storage cost estimation for the I/O buffers is to implement a storage structure of size S with R read ports and W write ports with the modules from the storage library. In SMASH, this is done by implementing the storage architecture with three types of storage modules:

1. registers,
2. register files, and
3. on-chip RAMs.

The cost of implementing the storage on these three modules is computed as described below. The minimum cost is used as the estimated storage structure cost. Note that the cost obtained here is only an estimate, and not a lower bound.

4.2.6.1 Implementing Storage Structure with Registers

A storage structure of size S with R read ports and W write ports using registers can be constructed as shown in Figure 4.5. The details of such a construction are as follows:

Number of registers needed (M)	= S
Size of input multiplexers	= W to 1
Number of input multiplexers	= S
Size of output multiplexers	= S to 1
Number of output multiplexers	= R

The total cost of this implementation is

$$Cost = S \times C_{reg} + S \cdot (2^{\lceil \log W \rceil} - 1) \cdot C_{mux} + R \times (2^{\lceil \log S \rceil} - 1) \cdot C_{mux}$$

Note that the n-to-1 multiplexer is constructed using $(2^{\lceil \log n \rceil} - 1)$ 2-to-1 multiplexers of cost C_{mux} each.

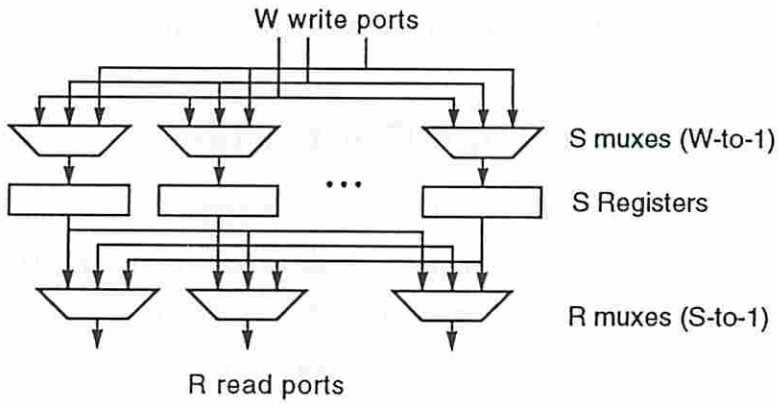


Figure 4.5: Constructing Storage using Registers

4.2.6.2 Implementing Storage Structure with Register Files

A storage structure of size S with R read ports and W write ports using register files with r-read ports and w-write ports can be constructed as shown in Figure 4.6.

The register files are allowed to have a variable size up to a maximum capacity of s_{max}. We connect each of the r read ports of a register file to R/r multiplexers only, because each read port of the register file can access all the data values of the register file. Thus, each data value is available at all the R read ports of the overall storage structure. Note that in this configuration certain access patterns are prohibited, nevertheless this configuration suffices for our needs. In this implementation

Total number of register files N	$= \lceil \frac{S}{s_{max}} \rceil$
Number of register files of size s_{max}	$= \lfloor \frac{S}{s_{max}} \rfloor$
Last reg. file size for remaining words	$= S \bmod s_{max}$
Size of input multiplexers	$= \lceil \frac{W}{w} \rceil$ to 1
Number of input multiplexers	$= w \times N$
Size of output multiplexers	$=$ N to 1
Number of output multiplexers	$=$ R

The total cost of this implementation is

$$\begin{aligned} Cost &= \left\lfloor \frac{S}{s_{max}} \right\rfloor \cdot C_{regfile}(s_{max}) + C_{regfile}(S \bmod s_{max}) \\ &\quad + w.N.(2^{\lceil \log \lceil \frac{W}{w} \rceil \rceil} - 1) \cdot C_{mux} + R \cdot (2^{\lceil \log N \rceil} - 1) \cdot C_{mux} \end{aligned}$$

4.2.6.3 Implementing Storage Structure with On-chip RAMs

A storage structure of size S with R read ports and W write ports using on-chip RAMs with r-read ports and w-write ports can be obtained in the same manner as done for register files. The RAM modules are allowed to have a variable size up to a maximum capacity of s_{max}. We connect each of the r read ports of a RAM module to R/r multiplexers only, because each read port of the RAM module can access all the data values of the RAM. Thus, each data value is available at all the R read ports of the overall storage structure. Note that in this configuration certain access-patterns are prohibited, nevertheless this configuration suffices for our needs. In this implementation

Total number of RAMs N	$= \lceil \frac{S}{s_{max}} \rceil$
Number of RAMs of size s_{max}	$= \lfloor \frac{S}{s_{max}} \rfloor$

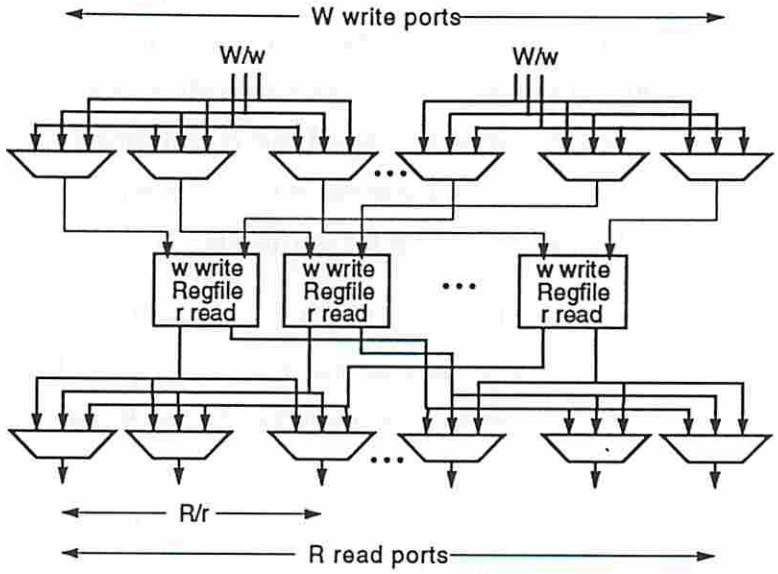


Figure 4.6: Constructing Storage using Register Files

Last RAM size for remaining words	$= S \bmod s_{max}$
Size of input multiplexers	$= \lceil \frac{W}{w} \rceil$ to 1
Number of input multiplexers	$= w \times N$
Size of output multiplexers	= N to 1
Number of output multiplexers	= R

The total cost of this implementation is

$$\begin{aligned} Cost &= \left\lfloor \frac{S}{s_{max}} \right\rfloor \cdot C_{ram}(s_{max}) + C_{ram}(S \bmod s_{max}) \\ &\quad + w \cdot N \cdot (2^{\lceil \log \lceil \frac{W}{w} \rceil \rceil} - 1) \cdot C_{mux} + R \cdot (2^{\lceil \log N \rceil} - 1) \cdot C_{mux} \end{aligned}$$

Note that the cost of a RAM module C_{ram} is a function of its size.

4.3 Functional Cost Estimation

To estimate the functional cost, first, the lower bound on the number of operators is determined. Then the minimum functional cost is estimated.

The lower bound of the number of functional operators is basically the maximum number of operators required simultaneously in a step. Since, the actual scheduling has not been performed yet, we count the number of operations that can be scheduled only during a specified interval (from the ASAP and ALAP analysis of the nodes) and assume that these operations will be uniformly distributed over that interval. A uniform distribution results in an average number of operations in a control step, whereas any other distribution will result in more number of operations than the average in at least one control step. Therefore, the maximum of operations performed in a single control step for the interval $[t_1, t_2]$ (which is the lower bound on the number of functional operators required for that interval) is minimum when the distribution is uniform. By performing this analysis over all the possible intervals and then determining the maximum number of operators that might be required in any interval, we determine the desired lower bound.

Once the lower bounds on the number of operators required for each type of operator are known, the functional cost can be estimated.

4.3.1 Lower Bound on Functional Modules

Theorem 4.3.6 *A lower bound on the number of operators of type O_k, $N_{LB}(O_k)$ required in execution of the CDFG in $MaxSteps$ is*

$$N_{LB}(O_k) = \max\left\{\left\lceil \frac{1}{(t_2 - t_1)} |L_{O_k,t_1,t_2}| \right\rceil\right\}$$

where $t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps$, and L_{O_k,t_1,t_2} is as per definition 4.1.14.

Proof: Same as proof of Theorem 4.2.1 except that the operator type here is O_k instead of R.

4.3.2 Lower Bound on the Total Functional Cost

The minimum functional cost is

$$Cost_{functional} = \sum_{\forall O_K} N_{LB}(O_k) \cdot Cost(O_k)$$

$Cost(O_k)$ is the cost of operator type O_k.

4.4 Upper Bounds on the Design Parameters

The upper bounds on various parameters in storage and functional architecture of the design are developed for theoretical reasons. Though we have not used these bounds in SMASH, they are developed for future inclusion in BEST, a behavioral area-delay estimator [Kuc91]. BEST requires both the lower and upper bounds for a tight estimation of various design parameters.

In this section, the following upper bounds are determined:

1. upper bound on read and write ports on buffers,
2. upper bound on the buffer size, and
3. upper bound on the number of functional modules of each type.

4.4.1 Upper Bound on Read (Write) Ports on Buffers

Theorem 4.4.7 *An upper bound on the number of read ports on the buffers, $N_{UB}(R)$, required in executing the CDFG in $MaxSteps$ is*

$$\begin{aligned} N_{UB}(R) &= \max\{|U_{R,t_1,t_1+1}|\} \\ &\quad \textit{where } t1 = 0 \ldots (MaxSteps - 1) \textit{ and} \\ &\quad U_{R,t_1,t_1+1} \textit{ is as per definition 4.1.15.} \end{aligned}$$

Proof: An upper bound on the number of read ports on the buffers, $N_{UB}(R)$ is nothing but the maximum number of inputs that may be accessed (i.e. the maximum number of read nodes that may be scheduled) in a step or interval $[t_1, t_1 + 1]$. Using Observation 4.1.2 and Definition 4.1.15, the maximum number of read nodes that may be scheduled in this step is $|U_{R,t_1,t_1+1}|$. The maximum taken over all the steps gives the desired upper bound. □

Theorem 4.4.8 *An upper bound on the number of write ports on the buffers, $N_{UB}(W)$, required in executing the CDFG in $MaxSteps$ is*

$$\begin{aligned} N_{UB}(W) &= \max\{|U_{W,t_1,t_1+1}|\} \\ &\quad \textit{where } t1 = 0 \ldots (MaxSteps - 1) \textit{ and} \\ &\quad U_{W,t_1,t_1+1} \textit{ is as per definition 4.1.15.} \end{aligned}$$

Proof: Same as above.

4.4.1.1 Computational Complexity

The computational complexity of determining the upper bound on the number of read ports on the buffers is $O(|V_R| \cdot MaxSteps)$. Determining U_{R,t_1,t_1+1} takes $O(V_R)$ steps for each t_1 and this must be done for all the control steps *i.e.* for $t_1 = 1 \ldots MaxSteps$. Therefore the overall complexity of this computation is $O(|V_R| \cdot MaxSteps)$.

4.4.2 Upper Bound on the Size of Buffers

The following theorem gives the upper bound on the total size of all the I/O buffers (when the buffers store both the inputs and outputs). This theorem is then extended to get the upper bound on the total size of all the input-only buffers and also that of output-only buffers.

Theorem 4.4.9 *An upper bound on the size of I/O buffers, $BufSize_{UB}(IO)$ required in executing the CDFG in $MaxSteps$ is the sum of input and output data sizes i.e.*

$$BufSize_{UB}(IO) = |IP| + |OP|$$

where IP and OP are the input and output data sets.

Proof: In a given interval $[t_1, t_2]$, the maximum number of inputs that may require storage is nothing but the number of inputs being accessed $I_{acc}(t_1, t_2)$ because in the worst case all the inputs may require storage. Similarly, the maximum number

of outputs that may requires storage is $O_{prod}(t_1, t_2)$. Therefore, the maximum I/O buffer size $BS_{ub}(IO, t1, t2)$ required for processing in the interval $[t_1, t_2]$ is

$$BS_{ub}(IO, t1, t2) = I_{acc}(t_1, t_2) + O_{prod}(t_1, t_2)$$

Now consider the interval $[0, MaxSteps]$,

$$\begin{aligned} BS_{ub}(IO, 0, MaxSteps) &= I_{acc}(0, MaxSteps) + O_{prod}(0, MaxSteps) \\ &= |IP| + |OP| \end{aligned}$$

As, $I_{acc}(0, MaxSteps)$ is the total number of inputs that may be accessed in the interval $[0, MaxSteps]$ and $O_{prod}(0, MaxSteps)$ is the number of outputs that may be produced during $[0, MaxSteps]$. □

Corollary 4.4.1 *An upper bound on the size of input-only buffer, $BufSize_{UB}(I)$ required in executing the CDFG in $MaxSteps$ is*

$$BufSize_{UB}(I) = |IP|$$

where IP is the input data set.

Proof: Similar to the above proof.

Corollary 4.4.2 *An upper bound on the size of output-only buffer, $BufSize_{UB}(O)$ required in executing the CDFG in $MaxSteps$ is*

$$BufSize_{UB}(O) = |OP|$$

where OP is the output data set.

Proof: Similar to the above proof.

Unfortunately, these upper bounds are very loose and are of little use. If we assume that the inputs and outputs are transferred such that they are not stored in the buffers, we get a better estimate of the maximum buffer size that may be required by the final design. (Because of the assumption the result remains an estimate of the maximum buffer size and not an upper bound.) The following theorem estimates

the maximum size of all the I/O buffers. The result is then extended to estimate the maximum size of all the input-only buffers as well as that of output-only buffers.

Theorem 4.4.10 *The maximum size of I/O buffers,* $BufSize_{max}(IO)$ *required in executing the CDFG in* $MaxSteps$ *can be estimated by:*

$$BufSize_{max}(IO) = \max\{BS_{max}(IO, t_1, t_2)\}$$

where

$$t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps,$$

BW_{on-off} *is the bandwidth between on and off chip memory and*

$$BS_{max}(IO, t_1, t_2)$$
$$= \begin{cases} |U_{I,t_1,t_2}| & \text{if } |U_{I,t_1,t_2}| \geq |U_{O,t_1,t_2}| \geq BW_{on-off} \cdot (t_2 - t_1) \\ |U_{O,t_1,t_2}| & \text{if } |U_{O,t_1,t_2}| \geq |U_{I,t_1,t_2}| \geq BW_{on-off} \cdot (t_2 - t_1) \\ 0 & \text{if } |U_{I,t_1,t_2}| + |U_{O,t_1,t_2}| \leq BW_{on-off} \cdot (t_2 - t_1) \\ |U_{I,t_1,t_2}| + |U_{O,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1) & \text{otherwise} \end{cases}$$

Proof: For a given interval $[t_1, t_2]$, if the number of inputs that may be accessed is $I_{acc}(t_1, t_2)$, the number of inputs that are transferred is $I_{trans}(t_1, t_2)$, the number of outputs that may be produced is $O_{prod}(t_1, t_2)$, and the number of outputs that are transferred is $O_{trans}(t_1, t_2)$; then the maximum size of all the I/O buffers, $BS_{max}(IO, t1, t2)$ is the maximum of (i) the number of inputs required to be stored before t_1 ($I_{st}(t_1, t_2)$), and (ii) the number of outputs required to be stored after t_2 ($O_{st}(t_1, t_2)$). The assumption here is that $I_{trans}(t_1, t_2)$ and $O_{trans}(t_1, t_2)$, the inputs and outputs transferred during the interval $[t_1, t_2]$, are scheduled such that they are not stored in the buffers. (Because of this assumption the result obtained here is an estimate of the maximum buffer size and not an upper bound.) Therefore

$$BS_{max}(IO, t1, t2) = \max\{I_{st}(t_1, t_2), O_{st}(t_1, t_2)\}$$

where

$$\begin{aligned} I_{st}(t_1,t_2) &= I_{acc}(t_1,t_2) - I_{trans}(t_1,t_2) \\ O_{st}(t_1,t_2) &= O_{prod}(t_1,t_2) - O_{trans}(t_1,t_2) \end{aligned}$$

Furthermore, the total number of inputs/outputs that can be transferred from/to the buffers is limited by the bandwidth and the duration of the interval, and assuming maximum utilization of the bandwidth:

$$I_{trans}(t_1,t_2) + O_{trans}(t_1,t_2) = BW_{on-off} \cdot (t_2 - t_1)$$

Therefore,

$$\begin{aligned} BS_{max}(IO,t_1,t_2) &= \max\{I_{st}(t_1,t_2), O_{st}(t_1,t_2)\} \\ &= \max\{(I_{acc}(t_1,t_2) - I_{trans}(t_1,t_2)), \\ &\quad (O_{prod}(t_1,t_2) - O_{trans}(t_1,t_2))\} \\ &= \max\{(I_{acc}(t_1,t_2) - I_{trans}(t_1,t_2)), \\ &\quad (O_{prod}(t_1,t_2) - BW_{on-off} \cdot (t_2 - t_1) + I_{trans}(t_1,t_2))\} \end{aligned} \quad (4.10)$$

Where $I_{trans}(t_1,t_2)$ is yet to be determined. Since, we are interested in the upper bound, the desired $I_{trans}(t_1,t_2)$ is the one which results in maximum $BufSize_{max}(IO,t_1,t_2)$. This is determined by considering the following two cases:
Case(i): $I_{acc}(t_1,t_2) \geq O_{prod}(t_1,t_2)$
Clearly, $BS_{max}(IO,t_1,t_2)$ will be maximum when $I_{trans}(t_1,t_2)$ is minimum. Basically, this will be the case when inputs are transferred into the buffers only after transferring the outputs from the buffers. Assuming maximum utilization of the bandwidth (*i.e. the maximum number of inputs and outputs are transferred.)*, we get the following three sub cases:

1. $O_{prod}(t_1,t_2) \geq BW_{on-off} \cdot (t_2 - t_1)$. In this case The whole bandwidth is utilized in transferring the outputs from the buffers. None of the inputs are transferred in the buffers *i.e.*

$$I_{trans}(t_1,t_2) = 0$$

2. $I_{acc}(t_1, t_2) + O_{prod}(t_1, t_2) \leq BW_{on-off} \cdot (t_2 - t_1)$. In this case all the inputs can be transferred in the buffers and outputs out of the buffers in $[t_1, t_2]$. Therefore

$$I_{trans}(t_1, t_2) = I_{acc}(t_1, t_2)$$

3. Otherwise, first the outputs are transferred out of the buffers and then the remaining bandwidth is utilized in transferring the inputs in the buffers *i.e.*

$$I_{trans}(t_1, t_2) = BW_{on-off} \cdot (t_2 - t_1) - O_{prod}(t_1, t_2)$$

In summary,

$$I_{trans}(t_1, t_2) = \begin{cases} 0 & \text{if } O_{prod}(t_1, t_2) \geq BW_{on-off} \cdot (t_2 - t_1) \\ I_{acc}(t_1, t_2) & \text{if } I_{acc}(t_1, t_2) + O_{prod}(t_1, t_2) \leq BW_{on-off} \cdot (t_2 - t_1) \\ BW_{on-off} \cdot (t_2 - t_1) - O_{prod}(t_1, t_2) & \text{otherwise} \end{cases}$$

Correspondingly, we get

$$BS_{max}(IO, t_1, t_2) = \begin{cases} I_{acc}(t_1, t_2) & \text{if } O_{prod}(t_1, t_2) \geq BW_{on-off} \cdot (t_2 - t_1) \\ 0 & \text{if } I_{acc}(t_1, t_2) + O_{prod}(t_1, t_2) \leq BW_{on-off} \cdot (t_2 - t_1) \\ I_{acc}(t_1, t_2) + O_{prod}(t_1, t_2) - BW_{on-off} \cdot (t_2 - t_1) & \text{otherwise} \end{cases}$$

Case(ii): $I_{acc}(t_1, t_2) < O_{prod}(t_1, t_2)$
Similar to case (i), here we get

$$BS_{max}(IO, t_1, t_2) = \begin{cases} O_{prod}(t_1, t_2) & \text{if } I_{acc}(t_1, t_2) \geq BW_{on-off} \cdot (t_2 - t_1) \\ 0 & \text{if } I_{acc}(t_1, t_2) + O_{prod}(t_1, t_2) \leq BW_{on-off} \cdot (t_2 - t_1) \\ I_{acc}(t_1, t_2) + O_{prod}(t_1, t_2) - BW_{on-off} \cdot (t_2 - t_1) & \text{otherwise} \end{cases}$$

By combining the two cases,

$$BS_{max}(IO, t_1, t_2)$$

$$= \begin{cases} I_{acc}(t_1,t_2) & \text{if } I_{acc}(t_1,t_2) \geq O_{prod}(t_1,t_2) \geq BW_{on-off} \cdot (t_2 - t_1) \\ O_{prod}(t_1,t_2) & \text{if } O_{prod}(t_1,t_2) \geq I_{acc}(t_1,t_2) \geq BW_{on-off} \cdot (t_2 - t_1) \\ 0 & \text{if } I_{acc}(t_1,t_2) + O_{prod}(t_1,t_2) \leq BW_{on-off} \cdot (t_2 - t_1) \\ I_{acc}(t_1,t_2) + O_{prod}(t_1,t_2) - BW_{on-off} \cdot (t_2 - t_1) & \text{otherwise} \end{cases} \tag{4.11}$$

(4.12)

$I_{acc}(t_1,t_2)$ and $O_{prod}(t_1,t_2)$ can be determined using Definitions 4.2.17 and 4.2.19, and Observation 4.1.2.

$$I_{acc}(t_1,t_2) = |U_{I,t_1,t_2}| \quad . \tag{4.13}$$

$$O_{prod}(t_1,t_2) = |U_{O,t_1,t_2}| \tag{4.14}$$

From Equations 4.11, 4.13 and 4.14 we get the desired result. □

Corollary 4.4.3 *The maximum size of all the input-only buffers, $BufSize_{max}(I)$ required in executing the CDFG in $MaxSteps$ can be estimated by:*

$$BufSize_{max}(I) = \max\{(|U_{I,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1)), 0\}$$

where $t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps$, and BW_{on-off} is the bandwidth between on and off chip memory.

Proof: The proof follows from the proof of the above theorem (Theorem 4.4.10) as it is a special case of the above theorem. In this case, the outputs are not considered, therefore $O_{prod}(t_1,t_2)$ is made 0 in Equation 4.11 *i.e.*

$$\begin{aligned} BS_{max}(I,t_1,t_2) &= \begin{cases} 0 & \text{if } I_{acc}(t_1,t_2) \leq BW_{on-off} \cdot (t_2 - t_1) \\ I_{acc}(t_1,t_2) - BW_{on-off} \cdot (t_2 - t_1) & \text{otherwise} \end{cases} \\ &= \max\{(I_{acc} - BW_{on-off} \cdot (t_2 - t_1)), 0\} \\ &= \max\{(|U_{I,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1)), 0\} \end{aligned}$$

The maximum taken over all the possible intervals $[t_1, t_2]$ gives the desired bound. □

Corollary 4.4.4 *The maximum size of all the output-only buffers,* $BufSize_{max}(O)$ *required in executing the CDFG in* $MaxSteps$ *can be estimated by:*

$$BufSize_{max}(O) = \max\{(|U_{O,t_1,t_2}| - BW_{on-off} \cdot (t_2 - t_1)), 0\}$$

where $t1 = 0 \ldots (MaxSteps - 1), t_2 = (t_1 + 1) \ldots MaxSteps$, *and* BW_{on-off} *is the bandwidth between on and off chip memory.*

Proof: Similar to the above proof.

4.4.2.1 Computational Complexity

The computational complexity of estimating the maximum size of all the I/O buffers is $O(MaxSteps^2 \cdot \max\{|V_R|, |V_W|\})$. Computing $BS_{max}(IO, t_1, t_2)$ requires $O(\max\{|V_R|, |V_W|\})$ steps. This analysis is done $O(MaxSteps^2)$ times. Therefore, the overall complexity of is $O(MaxSteps^2 \cdot \max\{|V_R|, |V_W|\})$. Similarly, the complexity of determining the upper bound of the total size of input-only buffers is $O(MaxSteps^2 \cdot \{|V_R|\})$ and that of the total size of all the output-only buffers is $O(MaxSteps^2 \cdot \{|V_W|\})$.

4.4.3 Upper Bound Cost of Storage Structure

An upper bound on the cost of storage is determined by implementing the I/O buffers of size $BufSize_{UB}(IO)$ with $N_{UB}(R)$ read ports and $N_{UB}(W)$ write ports on registers, register files and on-chip RAMs as described in Section 4.2.6.1. The minimum of the three implementations is considered to be the upper bound cost of the storage structure.

4.4.4 Upper Bound on the Number of Functional Modules

Theorem 4.4.11 *An upper bound on the number of functional modules of type* O_k, $N_{UB}(O_k)$, *required in executing the CDFG in* $MaxSteps$ *is*

$$N_{UB}(O_k) = \max\{|U_{O_k,t_1,t_1+1}|\}$$

where $t1 = 0 \ldots (MaxSteps - 1)$ *and*

U_{O_k,t_1,t_1+1} is as per definition 4.1.15.

Proof: Same as proof of Theorem 4.4.7 except that the operator type here is O_k instead of R.

4.4.5 Upper Bound on the Total Functional Cost

The maximum functional cost is

$$Cost_{functional} = \sum_{\forall O_K} N_{UB}(O_k) \cdot Cost(O_k)$$

$Cost(O_k)$ is the cost of operator type O_k.

4.5 Summary

In this chapter we developed techniques to estimate the storage cost and the functional cost. The storage cost estimation consist of the following three steps:

1. determining lower bound on the number of read and write ports on the buffers. We also presented the required theoretical basis to prove these bounds,
2. determining lower bound on the total size of all the buffers. Again, we provided theoretical basis for this bound through a proof, and
3. implementing these requirements on lowest cost storage modules from the library.

We also developed theory to determine the upper bounds for

1. the number of read and write ports on the buffers,
2. the total size of all the buffers, and
3. the number of functional modules in the design.

Chapter 5

Synthesis with Storage Tradeoffs Lookahead

5.1 Introduction

In this chapter we address datapath synthesis with lookahead in storage structure tradeoffs. During this synthesis step, datapath operations are scheduled combined with the scheduling of data transfers between I/O buffers and the datapath while looking ahead to evaluate storage structure tradeoffs. The general scheduling problem in architectural synthesis is presented first, then datapath scheduling in SMASH is described in detail. Datapath scheduling is a crucial step as many of the architectural tradeoffs, including storage-related tradeoffs, are made during this step. In this chapter, the details of the scheduling algorithm used in SMASH are presented. The chapter contains a description of how SMASH considers storage-related parameters, makes tradeoffs in storage architecture, and uses storage and functional cost estimations during datapath scheduling. Also included are the details of the techniques used in SMASH for handling various issues such as conditional branches, loops, operators with varying execution delays, and constraints on bandwidth and I/O timing.

5.2 Datapath Scheduling Problem

The scheduling problem in architectural synthesis is generally formulated in the following two ways:

1. *Scheduling under hardware constraints*: For a given choice of hardware modules with prespecified properties and total cost, sequence the operations in the CDFG such that the schedule length is minimized.

2. *Scheduling under timing constraints*: For given time constraints, minimize the number of hardware modules needed for execution of the CDFG.

In synchronous systems the basic correctness constraints are that every hardware unit can be used only once during a control step. Combinational logic cannot have feedback, buses cannot carry more than one value, registers can be loaded only once and read/write ports of storage modules can be accessed only once.

5.2.1 Problem Definition

The general *control data flow graph scheduling problem* in architectural synthesis is associating with each node $v_i \in V$, a control step $\mathcal{S}(v_i)$ such that

$$v_i \prec v_j \;\Rightarrow\; \mathcal{S}(v_i) \leq \mathcal{S}(v_j) \tag{5.1}$$

using the functional modules from the library, while satisfying the input timing constraints.

Also provided is a *library* of the functional modules such that there is one (and only one) type of functional operator in the module library to execute a functional operation in the CDFG. It is assumed that operator types have been preselected by another ADAM tool like SLIMOS or by the user. Each operator of type k has a cost $Cost_{O_k}$ (which could be in terms of area, number of transistors or static power) and delay D_{O_k} (which is time taken for execution) associated with it.

5.2.2 Various Approaches

Much research has been done in datapath scheduling. Four major types of scheduling techniques exist in the literature [Sto91]:

- *Transformational scheduling* algorithms: These algorithms apply transformations to improve an existing schedule [BCM+88, Pen86]. The algorithm may start from either a fully serial schedule which is then parallelized by applying

transformations or a maximally parallel schedule to which serializing transformations are applied.

- *Integer-linear programming* techniques: These techniques formulate the problem as an ILP problem. Here again, researchers have formulated the scheduling problem under timing constraints [LHL89, PK90] and also under hardware constraints [HHL90]. These techniques have high time complexity.

- *Neural network scheduling* algorithms: These algorithms use self-organizing rules to solve the problem [HP90]. The cited approach schedules under timing constraints. This technique is also quite slow but is extremely suitable for parallel machines.

- *Iterative scheduling* algorithms: These algorithms schedule one operation at a time. Generally, either of the following two types of schemes is adopted:

  1. The maximum number of operations is scheduled in a step and then the next step is considered. Examples are as soon as possible (ASAP) scheduling, as late as possible (ALAP) scheduling and list scheduling. In the ASAP scheduling, an operation is scheduled in the earliest time step. ASAP scheduling results in the shortest execution time but may require extra hardware. Similarly, in the ALAP scheduling, an operation is scheduled in the latest time step by starting from the terminal nodes in the data flow graph and determining the latest time an operation can be scheduled. In list scheduling, some criterion is used to delay certain operations. Different researchers have experimented with different criteria in their schedulers. ELF uses *urgency* [GBK85], whereas Slicer uses *mobility* as the criterion to delay the operation [PG87]. Sehwa uses a combination of list schedulers but can switch to exhaustive search to find a final solution when the search space has been sufficiently pruned [PP88].

  2. Each operation is chosen by some priority function and is scheduled in the best possible step. Examples include critical path [PPM86] and distribution-based schedulers such as force-directed scheduling [PK87].

5.3 Datapath Scheduling in SMASH

The problem being addressed here is as follows: given the behavior of an application-specific system in the form of a CDFG (obtained from the VHDL description), the module library, area/performance constraints, the external bandwidth constraints, and an optional I/O timing constraint; the scheduling algorithm must schedule the operations in the CDFG, including I/O reads and writes, while satisfying all the constraints. It also determines the number of functional modules and the number of read and write ports on the I/O buffers required by the schedule.

Figure 5.1 shows the top-level view of the inputs and outputs of this step in SMASH.

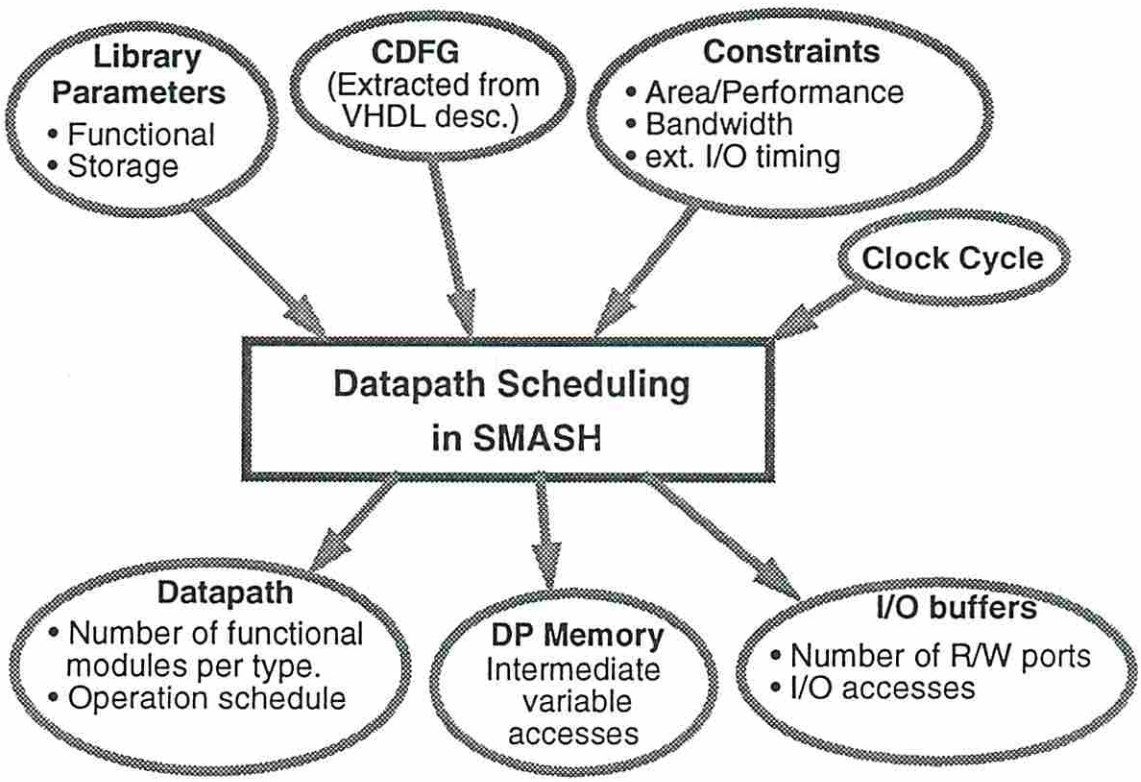


Figure 5.1: Datapath Scheduling in SMASH

Besides combining the datapath scheduling with I/O access scheduling during this step, (*i*) the storage architecture parameters are considered so that the schedule is guaranteed to satisfy the constraints during the storage synthesis, (*ii*) tradeoffs

in the storage architecture are considered so that the schedule does not require alterations during storage synthesis, and (*iii*) estimation techniques for storage and functional cost are used to evaluate the impact of high-level decisions on the final design so that potentially inferior designs are discarded early in the design process.

SMASH first preprocesses the CDFG, then schedules the processed CDFG, and finally, analyzes the schedule to verify the correctness and quality of the schedule. These three stages are described below.

5.4 Preprocessing the CDFG

SMASH preprocesses the CDFG before performing scheduling. The tasks performed during preprocessing are described below.

Inserting R/W Nodes in the CDFG

To schedules I/O accesses, SMASH inserts a read/write (R/W) node in the CDFG corresponding to each I/O access from the buffers. These R/W nodes have already been described in Chapter 3.

ASAP and ALAP Analysis of the CDFG

The ASAP and the ALAP analyses of the CDFG are performed in this step. The $ASAP(v)$ and $ALAP(v)$ times for a node $v \in V$ are defined as follows.

Definition 5.4.23 *$ASAP(v)$ is the earliest possible time for executing the node v in the CDFG.*

Definition 5.4.24 *$ALAP(v)$ is the latest possible time for executing the node v so that the CDFG is executed in $MaxSteps$.*

The basic procedure to determine $ASAP(v)$ times for all the nodes in the CDFG is outlined in Figure 5.2, and is a depth-first recursion. The ALAP procedure is similar to the ASAP procedure.

```
Procedure ASAP(node,step)
/* Should be called with node = ROOT and step = 1 */

  1. if (ASAP(node) < step) then ASAP(node) = step;

  2. if (OP(node) == OUTPORT) return;
     /* end of recursion.  */

  3. else, for all the outgoing edges of node

     (a) next_node = head of the outgoing edge.

     (b) next_step = step + delay of OP(v);

     (c) ASAP(next_node,next_step);

  4. return;
```

Figure 5.2: ASAP Analysis of the CDFG

Determination of Lower Bounds on Design Parameters

SMASH determines the lower bounds on the following design parameters based on the ASAP and ALAP schedules of the nodes:

1. number of read and write ports on the I/O buffers,
2. number of functional modules of each type, and
3. size of the I/O buffers.

These lower bounds are used as the initial resource constraints in the scheduling algorithm as described in the next section. In addition, SMASH estimates the storage and functional costs based on these lower bounds. The cost is in terms of chip area. This information is provided to the user to help him/her in setting the design constraints.

Specification of the Design Goal

Besides specifying the design constraints, the user can also specify the design goal. By design goal we mean whether priority should be given to (i) *performance optimization*, or (ii) *area optimization* during scheduling while satisfying the area or performance constraints [PPM86].

- Performance optimization: when performance optimization is the design goal, whenever the scheduler is unable to schedule a node in a control step due to resource limitations, SMASH adds an extra resource. However, if the addition of the extra resource causes an area constraint violation, then SMASH extends the execution time by inserting an extra step. When both the options result in a constraint violation, SMASH prompts an error and aborts the scheduling.

- Area optimization: when area optimization is the design goal, SMASH inserts an extra step whenever the scheduler is unable to schedule a node. However, if inserting the extra step causes a performance constraint violation then an extra resource is added. Here also, when both the options cause constraint violation, the scheduling is aborted.

Specification of the design goal facilitates the user in obtaining a design which is near-optimal for the desired parameter (area or performance).

5.5 Scheduling the CDFG

5.5.1 Assumptions

The scheduling software assumes two-phase clocking as described in Chapter 3. The datapath reads and processes the data in the first phase and writes it back into memory in the second phase. In case the datapath interacts directly with the off-chip memory, it writes the data into the off-chip memory in the second phase. SMASH allows loops with fixed iterations in the VHDL description. While estimating the storage cost, only the I/O buffer cost is included. Estimates of the cost of datapath memory have already been researched [Kuc91] and will be included in future. The basic scheduling algorithm is outlined in Figure 5.3.

```
Procedure ListScheduling

1. candidate_list = MostUrgentNodes();
   /* sort nodes in CDFG based on their freedom.  */

2. while (candidate_list != NULL) {

   (a) node = GetBestNode(candidate_list);
       /* get the node with the least freedom from candidate list.
       */

   (b) step = GetBestStep(node);
       /* determine the best step for scheduling node.  */

   (c) if node cannot be scheduled in any step due to lack of
       resources then increase

        i. resources when optimizing performance is the design
           goal, or

       ii. execution time when optimizing area is the design goal.

   (d) ScheduleNodeInStep(node,step);
       /* schedule node in step.  */

   (e) DeleteFromCandidateList(node);
       /* remove node from candidate list.  */

   } /* while */
```

Figure 5.3: Scheduling Algorithm in SMASH

5.5.2 Overview of the Scheduling Algorithm in SMASH

The following are the salient features of the scheduling algorithm used in SMASH.

Scheduling I/O accesses with datapath operations

SMASH considers I/O accesses from/to the on-chip buffers as R/W operations and schedules them concurrently with the datapath operations. SMASH performs this concurrent scheduling by treating the R/W nodes (which are inserted during the preprocessing of the CDFG) as functional operators performing *data transfers*. Whenever an operation involving I/O is scheduled, the corresponding R/W node is also scheduled, implying an I/O buffer access.

Scheduling Algorithm - Freedom Based List Scheduling

SMASH uses freedom-based list scheduling with freedom of the nodes as inverse of the urgency factor [PPM86], but any other scheduling technique which performs scheduling under hardware and performance constraints could be used. The freedom of a node v is given by:

$$Freedom(v) = ALAP(v) - ASAP(v)$$

The higher the freedom, the lower the urgency for scheduling.

Distribution graphs of the R/W operations and the datapath operations are used to assign them probabilistically to the most suitable control step, as done in force-directed scheduling [PK89]. Distribution probability of an operation type O_k in step s is given by

$$DistProbability(O_k, s) = \sum_{\forall v \in V_{O_k}} \frac{1}{ALAP(v) - ASAP(v) + 1}$$
$$\text{where, } ASAP(v) \leq s \text{ and } ALAP(v) \geq s$$

```
Procedure GetBestStep(node)

 1. min_dist = BIGNUMBER;

 2. best_step = NONE;

 3. for step = ASAP(node) to ALAP(node) {

    (a) op_dist = read_dist = write_dist = 0.0;

    (b) Check if there are sufficient resources (including read
        and write ports) for node to be scheduled in step.

    (c) If yes then,

          i. Compute distribution probability of node's operator.
         ii. If node requires inputs then compute distribution
             probability of READ operator.
        iii. If node produces outputs then compute distribution
             probability of WRITE operator.
         iv. Compute the total of the above three distribution
             probabilities.

 4. best_step = step with sufficient resources and minimum total
    distribution probability.

 5. return(best_step);
```

Figure 5.4: Selecting the Most Suitable Step in SMASH

Port Constraints on the Buffers

The read and write port constraints are used to avoid high cost for the buffers. The scheduler ensures the availability of a functional operator as well as R/W operations during scheduling (Figure 5.3) by checking whether the I/O buffers are able to provide the required number of read and write ports in each step, *i.e.*

$$\begin{aligned} R_{req}(s) &\leq R_{buf} \qquad \forall s, \text{ and} \\ W_{req}(s) &\leq W_{buf} \qquad \forall s \end{aligned}$$

Bandwidth and I/O Timing Constraints

To avoid violation of bandwidth and I/O timing constraints imposed by the user, the scheduler ensures that the inputs can be prefetched into the buffers from the background memory, and outputs can be transferred back to the background memory from the buffers before they are required elsewhere:

$$\begin{aligned} I(s) &\leq BW_{on-off} \times s \qquad \forall s \\ O(s) &\leq BW_{on-off} \times (MaxSteps - s) \qquad \forall s \end{aligned}$$

where $I(s)$ is the number of data variables required for all the operations scheduled before or in control step s. And $O(s)$ is the number of data variables produced by all the operations scheduled after or in control step s. The timing constraints on inputs and outputs are specified in a file in the following format:

```
data-name data-type timing-constraint
```

The `data-name` is the name of the data variable, the `data-type` is the type of the data variable, and the `timing-constraint` is the timing constraint on the data variable. For an input variable, the `data-type` is `INPUT`, and the `timing-constraint` is the time when the input is made available for processing. Similarly for an output variable, the `data-type` is `OUTPUT`, and the `timing-constraint` is the time by when the output must have been produced.

Storage Architecture Tradeoffs

The scheduler considers storage architecture trade-offs. During scheduling, SMASH takes the storage architecture features into account and ensures that in the following synthesis steps when it is actually doing the storage synthesis, the datapath schedule supports the selected storage architecture.

The following three types of tradeoffs possible in the storage architecture are considered in SMASH:

1. storage size *vs.* number of execution cycles,
2. number of ports *vs.* number of execution cycles, and
3. number of ports *vs.* storage size.

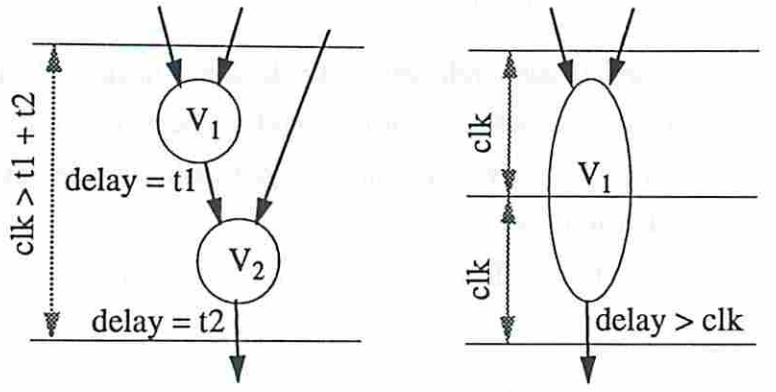
To deal with this 3-way tradeoff, SMASH iterates on the number of ports and, for each choice of number of ports, it trades off between the size and the execution time. This can be done because the number of ports does not vary too much in practical designs. Finally, the most cost-effective design (in terms of chip area) is chosen from these designs. The outer design loop minimizes the number of ports on the I/O buffers, and the inner design loop minimizes the chip area by minimizing the size of the buffers.

Chained and Multicycle Operations

To improve the total execution time of the CDFG and allow operators with varying execution delays, the nodes in the CDFG are *chained*, or *distributed* over multiple cycles appropriately. In operations chaining, if the total delay of multiple dependent nodes is less than the clock cycle then they are scheduled in the same control step, as illustrated in Figure 5.5 a. Alternately, if the delay of the operation is longer than the clock cycle, then it is executed in multiple control steps, as illustrated in Figure 5.5 b.

Handling Conditional Branches

The scheduler allows operator sharing among mutually-exclusive nodes of conditional branches. A predicate (branching condition) bitmap $PD(v)$ is associated with each



a. Operation Chaining b. Multi-cycle Operation

Figure 5.5: Operators with Varying Delays

node v. Also associated with each v is a predicate value map $PV(v)$. $PV(v)$ represents the predicate values (True or False) of all the predicates associated with v.

A one at the i^{th} bit in $PD(v)$ implies that predicate i (P_i) controls v whereas a zero at the i^{th} bit in $PD(v)$ implies that P_i does not control v. When P_i controls v, a one at the i^{th} bit in $PV(v)$ implies that v is executed when P_i is *True*, similarly, a zero at the i^{th} bit in $PD(v)$ implies that v is executed when P_i is *False*. The $PD(v)$ and $PV(v)$ are used in determining the mutual exclusion among the nodes efficiently.

The procedure to determine the mutual-exclusion between two nodes v_i and v_j is described in Figure 5.6.

```
Procedure MutualExclusion(v_i, v_j)
  1. if PD(v_i) == PD(v_j) then
       (a) if PV(v_i) == PV(v_j) then return(NO);
       (b) else return(YES);
  2. else return(NO);
```

Figure 5.6: Determining Mutual Exclusion between Nodes v_i and v_j

Handling Loops

The loops are *folded* in order to achieve higher performance as shown in Figure 5.7. Associated with every loop ℓ we have the following two parameters:

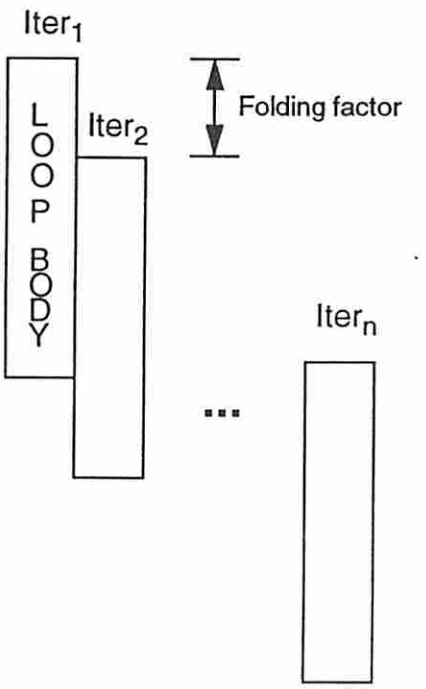


Figure 5.7: Loop Folding in SMASH

1. The *folding factor* f_ℓ, which is the delay (in clock cycles) between two consecutive iterations of the loop. f_ℓ is based on the inter-data dependency of the nodes within the loop body and is determined by the scheduler. A variation in the folding factor also results in tradeoffs in storage structure as well as datapath. This kind of tradeoff is illustrated by an experiment described later in the thesis in Section 7.7.2.

2. The *number of iterations* $Iter_\ell$ which is the number of times the loop body is executed. SMASH assumes that $Iter_\ell$ is finite and is specified in the VHDL description for every loop.

A node $v \in V$ belonging to loop ℓ, when scheduled in step s, is also executed in steps $s + f_\ell, s + 2f_\ell, \ldots, s + (Iter_\ell - 1) \cdot f_\ell$; *i.e.* v will be scheduled in steps

$$s + i \cdot f_\ell \text{ for } i = 0, \ldots, (Iter_\ell - 1)$$

In case of nested loops, if v belongs to loops $\ell_n, \ell_{n-1}, \ldots, \ell_1$, where ℓ_n is the outermost loop and ℓ_1 is the innermost, then v will be scheduled in steps

$$\begin{aligned} (((s + i_1 \cdot f_{\ell_1}) + i_2 \cdot f_{\ell_2}) \quad \ldots \quad & i_n \cdot f_{\ell_n}) \\ \text{for} \qquad & i_1 = 0, \ldots, (Iter_{\ell_1} - 1), \\ & i_2 = 0, \ldots, (Iter_{\ell_2} - 1), \\ & \quad \vdots \\ & i_n = 0, \ldots, (Iter_{\ell_n} - 1), \end{aligned}$$

I/O for Conditional Branches

To handle the I/O requirements of the operations inside **conditional branches** efficiently, the scheduler attempts to schedule the *test* node as early as possible. As a result, the interval between the test node and operation node, requiring I/O, is large enough to transfer the data into I/O buffers from off-chip memory. The data transfer must be done concurrently with the transfer of other data values being used in other parts of the CDFG. When the data transfer is scheduled, the software may or may not be able to make all the required transfers in time. This gives rise to the following two scenarios (Figure 5.8).

1. Dynamic selection of the data to be transferred: if SMASH is able to transfer the data between the test execution and the operation execution then the data which satisfies the predicates can be chosen dynamically during execution. This strategy avoids unnecessary transfers and is effective when the data (array) sizes are huge.

2. Worst case analysis: in case there is not enough time (or bandwidth) to transfer the data, after the test is executed all the data values are scheduled to be

transferred to the buffers irrespective of the outcome of the test, so that the appropriate data value is available during the execution. This results in extra data transfers but avoids delay in execution. Such situations may arise in real-time systems demanding very high performance.

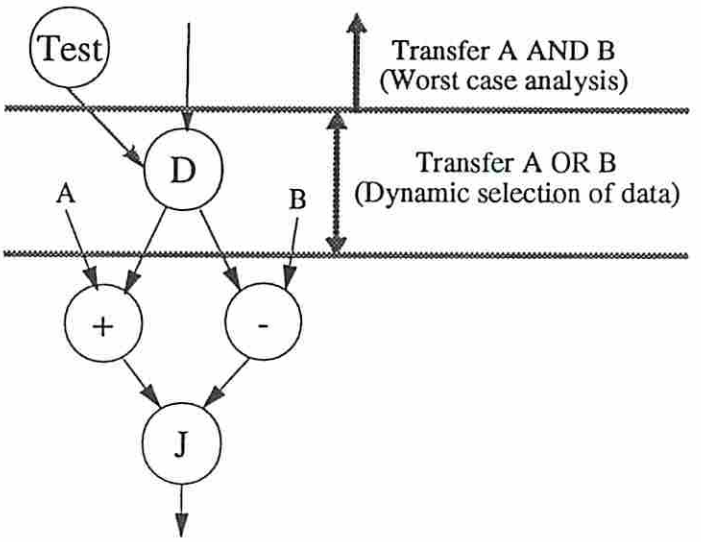


Figure 5.8: Data Transfer into I/O buffers for Conditional Branches

Array Accesses

Our approach to array accesses is very similar to our handling of conditional branches (Figure 5.9). Here, the node computing the array index is scheduled as early as possible. Again, during the actual data-transfer scheduling there could be the following two scenarios.

1. Dynamic selection of the data to be transferred: transfer only the referenced data of the array.

2. Worst case analysis: transfer the whole array. This strategy is suitable for high-performance real-time systems.

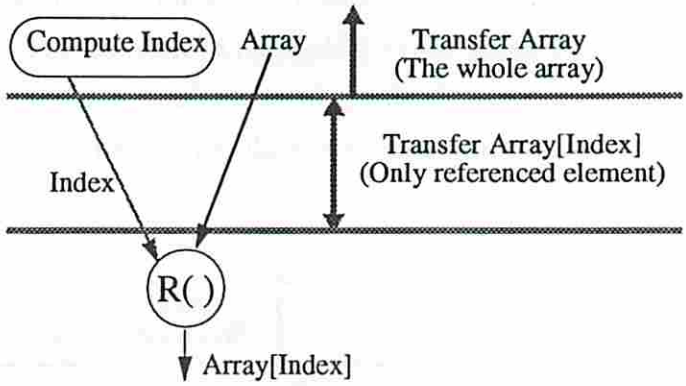


Figure 5.9: Array Accesses .

Cost Estimation

The scheduler uses storage cost and functional cost estimations described in 4. The cost of the storage and functional modules are estimated based on the partial schedule. These estimated costs are used in

- verifying whether the area constraints will be satisfied in the final designs, and
- evaluating the decision regarding scheduling an operation in a step, whether scheduling the operation in that step would result in more resources in the future.

5.5.3 Discussion on the Scheduling Algorithm

To summarize, the scheduling approach in SMASH uses freedom-based list scheduling. It is enhanced by using distribution graphs to select the best step whenever there is a choice. It combines scheduling of I/O accesses with datapath scheduling by modeling them as transfer operations. The implementation allows operation chaining and also multicycle operators. It can handle conditional branches and loops in an efficient manner. All the storage-related constraints (which include port constraints on the buffers, on-off chip bandwidth constraints and I/O timing constraints) are also considered.

As a result of this approach, the schedule obtained guarantees that no bandwidth or I/O timing constraint is violated in the next synthesis step when the complete

I/O transfer schedule between the on-chip and off-chip memory is determined. In addition, this process provides the I/O requirement schedule which is the starting point for the second step.

5.6 Analysis of the Schedule

SMASH verifies the schedule by checking data dependencies as defined in Equation 5.1 for all the nodes in the CDFG. It also outputs the percentage utilization of each type of module in each step, which helps us determine the quality of the schedule.

5.7 Summary

In this chapter we presented the details of the scheduling algorithm used in SMASH. We described the three stages of the scheduler: preprocessing of the CDFG, scheduling of the CDFG, and analysis of the schedule. The core of the scheduler is the scheduling stage. We presented the details of the techniques used in SMASH to combine scheduling of I/O accesses with datapath scheduling, to handle conditional branches and loops, to include the bandwidth and I/O timing constraints during scheduling, and to incorporate storage architecture tradeoffs during datapath synthesis.

The schedule obtained using the approach presented here guarantees that no bandwidth or I/O timing constraint is violated in the next synthesis step when the complete I/O transfer schedule between the on-chip and off-chip memory is determined. In addition, this step provides the I/O requirement schedule which is the starting point for the second step and described in the next chapter.

Chapter 6

Storage Synthesis

6.1 Introduction

After performing combined datapath and I/O access scheduling (as described in Chapter 5), our next step in the synthesis process is the design of the storage structure. At this stage in the synthesis process, we have determined the following design parameters:

- complete schedule for the datapath,
- I/O access schedule as required by the datapath,
- number of read and write ports on the I/O buffers (implicitly determined by the I/O access schedule),
- the time interval during which the inputs and outputs are available, and
- BW_{on-off}, the bandwidth between the on-chip and off-chip memory.

Also note that the datapath schedule obtained in the previous synthesis step (datapath scheduling) guarantees that no bandwidth or I/O timing constraint will be violated during the storage synthesis step.

Our objective in this step is to construct the storage structure which consists of the following three sub-architectures, as illustrated in Figure 3.5:

1. on-chip I/O buffers,
2. on-chip datapath memory, and

3. off-chip background memory.

As was described in Chapter 3, the construction of each sub-architecture consists of two basic tasks:

1. data-transfer scheduling, where the read and write times of each word are determined, and

2. module allocation, where a physical location is assigned to each word.

In this chapter we describe the design of the I/O buffers and the background memory. The design of the datapath memory has not been addressed in this thesis as it has been researched extensively by other researchers [BMB+88, Che91, Sto89]. For the design of I/O buffers and background memory, we have developed techniques to perform the data-transfer scheduling. These techniques are described below and have been implemented in SMASH. The module allocation step requires further research and is not addressed in this dissertation.

6.2 Data Transfer Scheduling for I/O Buffers

The data-transfer scheduling step consists of scheduling the data transfers between the off-chip background memory and I/O buffers with the objective of minimizing the buffer size, as shown in Figure 6.1. The data-transfer schedule must satisfy the data requirements as determined in the datapath scheduling step (Chapter 5).

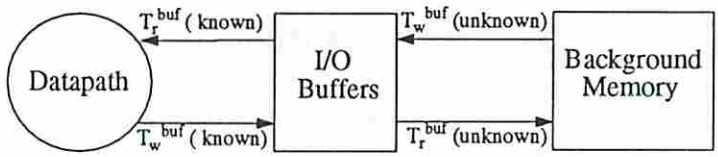


Figure 6.1: Data Transfer Timing for I/O Buffers

The problem is described as follows: The software is given as inputs

- the number of read and write ports on the I/O buffers,

- the time interval during which the inputs and outputs are available, birth time $\mathcal{T}_b^{buf}(d)$ and death time $\mathcal{T}_d^{buf}(d)$ for all the data values d, and
- the data requirement by the datapath, data read time from the buffers $\mathcal{T}_r^{buf}(d)$ for the inputs and data write time in the buffers $\mathcal{T}_w^{buf}(d)$ for the outputs.

The following parameters are to be determined:

- the data transfer schedule to and from off-chip background memory, which consists of
  - data write time to the buffers from the background memory $\mathcal{T}_w^{buf}(d_i)$ for the inputs, and
  - data read time from the buffers to the background memory $\mathcal{T}_r^{buf}(d_o)$ for outputs,

  such that the size of the I/O buffers is minimized, and the buffers provide the data to the datapath when they are required and store the outputs when they are produced; and
- the required buffer size.

Figure 6.2 shows the top-level view of the inputs and outputs of this step in SMASH.

6.2.1 Our Approach

The data transfer scheduling problem is analogous to the operation scheduling problem in datapath synthesis in the following two ways:

- Equivalent to total time required for CDFG execution (the schedule length) in datapath scheduling, we have the total time required for data transfer in this problem.
- Similarly, equivalent to the number of resources in datapath scheduling, we have the number of ports, that determines the bandwidth, or the number of words that can be accessed per cycle in this problem. In this case, the ports are the resources executing the *data transfer* operation.

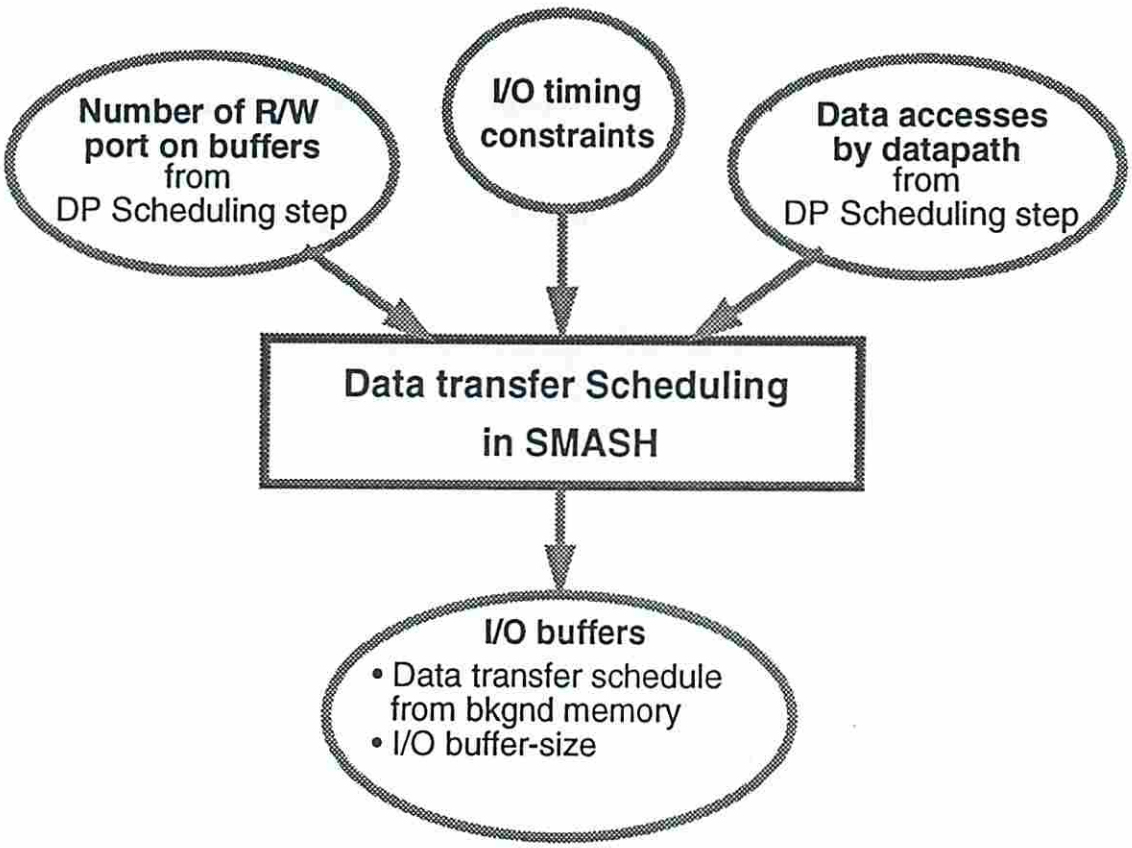


Figure 6.2: Data Transfer Scheduling in SMASH

However, the objective function in this problem differs from that in the datapath scheduling problem. In datapath scheduling, the objective is to minimize either the execution time or the number of resources, whereas in this problem, both the data transfer time and the number of resources (read/write ports) have already been decided in the previous step. The objective here is to minimize the size of the buffers. This aspect of our problem makes it different from the traditional datapath scheduling problem and more complex. However, we have used existing scheduling techniques with a redefinition of the objective function.

Before presenting our algorithm, we describe how the read and write times of the data values influence the buffer size.

Definition 6.2.25 *The set of inputs $Buf_i(s)$ present in the buffers in step s is*

$$Buf_i(s) = \{d_i | d_i \in IP \text{ and } \mathcal{T}_r^{buf}(d_i) \geq s\} - \{d_i | d_i \in IP \text{ and } \mathcal{T}_w^{buf}(d_i) > s\}$$

where $\mathcal{T}_r^{buf}(d_i)$ is the time when d_i is read by the datapath from the I/O buffers, and $\mathcal{T}_w^{buf}(d_i)$ is the time when d_i is written into the I/O buffers from the background memory.

Definition 6.2.26 *The set of outputs $Buf_o(s)$ present in the buffers in step s is*

$$Buf_o(s) = \{d_o | d_o \in OP \text{ and } \mathcal{T}_w^{buf}(d_o) \leq s\} - \{d_o | d_o \in OP \text{ and } \mathcal{T}_r^{buf}(d_o) < s\}$$

where $\mathcal{T}_w^{buf}(d_o)$ is the time when d_o is written by the datapath into the I/O buffers, and $\mathcal{T}_r^{buf}(d_o)$ is the time when d_i is transferred into the background memory from the I/O buffers.

$Buf_i(s)$ is the set of inputs which are read from the buffers by the datapath in steps $s, s+1, s+2, \ldots, MaxSteps$ but are not written into the buffers after step s, therefore these inputs must be written into the buffers in or before step s and will be present in the buffers in step s. Similarly, $Buf_o(s)$ is the set of outputs which are written by the datapath in steps $s-1, s-2, \ldots, 1$ but are not read by the background memory in these steps. Therefore, they will be present in the buffers in step s.

Definition 6.2.27 *The size of the I/O buffers is given by:*

$$\begin{aligned} BufSize &= \max_{\forall s}\{|Buf(s)|\} \\ &= \max_{\forall s}\{|Buf_i(s) + Buf_o(s)|\} \end{aligned}$$

The size of the buffers is determined by the maximum number of data values in the buffer in a control step, which implicitly depends on the data transfer timings of the I/O.

6.2.2 The Algorithm

The data-transfer scheduling algorithm used in SMASH is outlined in Figure 6.3. The salient features of this algorithm are discussed below.

Scheduling Technique

SMASH uses *list scheduling* to schedule data transfers. A list of all the input and output values to be scheduled is maintained in each step and the most urgent data transfers are scheduled based on their *urgency factor*. The *urgency factor* ($U(d, s)$), associated with each data value, determines the necessity of transferring d in step s. The following theorem provides the basis for computing the urgency factor.

Theorem 6.2.12 *The contribution of an input, read by the datapath from the buffer, to the buffer size is minimum when the input is transferred to the buffers as late as possible.*

Proof: Consider the input d_i which is accessed by the datapath from the buffers in control step $\mathcal{T}_r^{buf}(d_i)$ and is transferred into the buffers from the background memory in step $s = \mathcal{T}_w^{buf}(d_i)$.

We know that the size of the buffers is given by

$$\begin{aligned} BufSize &= \max_{\forall s}\{|Buf(s)|\} \\ &= \max\{|Buf(1)|, |Buf(2)|, \ldots, |Buf(MaxSteps)|\} \end{aligned}$$

Clearly, by Definition 6.2.26, the set $Buf(s-1)$ of data values will not include d_i as $s - 1 < \mathcal{T}_w^{buf}(d_i)$. However, if d_i is transferred into the buffers in control

Algorithm **DataTransferScheduling**

1. Read I/O timing constraints, and the read ports R_{buf} and write ports W_{buf} on the buffers.
2. Read I/O access pattern as required by the datapath
   (a) For $step = 1$ to $MaxSteps$, get Reads[$step$] and Writes[$step$]
3. Schedule data transfers for inputs.
   (a) $list$ = NULL;
   (b) For $step = MaxSteps$ down to 1
      i. Add all the inputs being read by the datapath in $step$ to $list$.
      ii. Compute the urgency factor for each input in $list$.
      iii. $avail\_ports$ = W_{buf} - |Writes[$step$]|;
      /* The number of writes that can be done in $step$.*/
      iv. Select $avail\_ports$ number of inputs from the $list$ to be written in the buffer in $step$ based on their urgency factor.
      v. Remove the scheduled inputs from $list$.
4. Schedule data transfers for OUTPUTS.
   (a) $list$ = NULL;
   (b) For $step = 1$ to $MaxSteps$
      i. Add all the outputs being written by the datapath in $step$ to $list$.
      ii. Compute the urgency factor for each output in $list$.
      iii. $avail\_ports$ = R_{buf} - |Reads[$step$]|;
      /* The number of reads that can be done in $step$.*/
      iv. Select $avail\_ports$ number of outputs from the $list$ to be read from the buffer in $step$ based on their urgency factor.
      v. Remove the scheduled outputs from $list$.
5. return;

Figure 6.3: Data Transfer Scheduling for I/O Buffers in SMASH

step $\mathcal{T}_w^{buf}(d_i) - 1$ instead of $\mathcal{T}_w^{buf}(d_i)$, the new set $Buf'(s-1)$ will include d_i as $s' = \mathcal{T}_w^{buf}(d_i)$, *i.e.*

$$Buf'(s-1) = Buf(s-1) \cup \{d_i\}$$

which implies

$$|Buf'(s-1)| = |Buf(s-1)| + 1$$

Hence, the new buffer size $BufSize'$ will be

$$\begin{aligned} BufSize' &= \max\{|Buf'(1)|, \ldots, |Buf'(s-1)|, \ldots, |Buf'(MaxSteps)|\} \\ &= \max\{|Buf(1)|, \ldots, |Buf(s-1)+1|, \ldots, |Buf(MaxSteps)|\} \\ &\geq BufSize \end{aligned}$$

Therefore, if the data d_i is transferred into the buffers prior to $\mathcal{T}_w^{buf}(d_i)$ it may increase the buffer size. □

Theorem 6.2.13 *The contribution of an output, written by the datapath into the buffer, to the buffer size is minimum when the output is transferred out of the buffers as soon as possible.*

Proof: Similar to the above proof. □

Ideally, the inputs should be transferred into the buffers in the same step as they are required, and the outputs should be transferred out of the buffers in the same step as they are produced. Unfortunately, this is not possible as the bandwidth available to perform the data transfer is limited and there may not be enough bandwidth available for the required data transfer in each step. Therefore, these transfers should be scheduled such that the bandwidth constraints are met while their presence in the buffers is minimized (to minimize the buffer size).

Based on the above theorem, our heuristic attempts to transfer the input data into the buffers from the background memory as late as possible, and transfer the output data from the buffers into the background memory as soon as possible. Unless it is necessary to transfer the input data value into the buffers, its transfer is postponed; similarly, the outputs are transferred out of the buffers as soon as possible.

Computing the Urgency Factor

The urgency factor ($U(d_i, s)$) for an input d_i in step s depends on $s - \mathcal{T}_b^{buf}(d_i)$. This is the number of steps available to transfer d_i into the buffers from the background memory before d_i is required by the datapath. The smaller this value is, the more urgent it is to transfer d_i into the buffers. If $s = \mathcal{T}_b^{buf}(d_i)$, then d_i must be transferred in step s. Furthermore, if d_i is accessed again by the datapath in a step s', such that $s' > s$ then its priority is lowered for the second transfer from the background memory as it can be stored in the buffers. As described during the discussion of storage tradeoffs, multiple transfers of data require higher BW_{on-off}. Since, BW_{on-off} is user specified, the inputs that are accessed again are retransferred only when there is sufficient bandwidth available for multiple transfers. SMASH still allows multiple transfers for these inputs because storing them may increase the size of the buffers.

Similarly, the urgency factor ($U(d_o, s)$) for an output d_o in step s depends on $\mathcal{T}_d^{buf}(d_o) - s$. This is the number of steps available to transfer d_o from the buffers into the background memory before it is required elsewhere. The smaller this value is, the more urgent it is to transfer d_o from the buffers. If $s = \mathcal{T}_d^{buf}(d_o)$, then d_o must be transferred in step s. (Note that we do not consider multiple writes of an output because if an output is produced multiple times, only the value which is produced last is valid.)

Updating the List of Data Values to be Scheduled

As mentioned above, for every control step a list of data values is maintained. Only these data values are considered for scheduling in that control step. While scheduling the input data transfers, the candidate list is determined as follows. In this case, the processing is done backwards from $MaxSteps$ to the first step. Assume that steps $MaxSteps, MaxSteps - 1, \ldots, s + 2, s + 1$ have been scheduled and now we are at step s. The candidate list (the input buffer set $Buf_i(s)$) contains the values that are

- read by the datapath from the buffers in step s, which is `Reads[s]` in our implementation, and

- read by the datapath from the buffers in steps $s+1, s+2, \ldots, MaxSteps$ but not written into the buffers from the background memory in these steps, which is the candidate list remaining after scheduling the data transfers in the previous iteration (corresponding to control step $s+1$).

These are the inputs that need to be transferred into the buffers from the background memory before s. In other words, this is the list of inputs which is considered for scheduling in step s.

For the outputs, the candidate list contains the outputs that are

- produced in step s by the datapath, which is `Writes[`s`]` in our implementation, and

- produced in steps $1, 2, \ldots, s-1$ but not transferred into the background memory in these steps, which is the candidate list remaining after scheduling the data transfers in the previous iteration corresponding to control step $s-1$.

Note that in this case the processing is done forwards from step 1 to $MaxSteps$.

6.2.3 Selecting the Data Values for Transfer

After creating the candidate list the algorithm selectively schedules the most urgent values in each step. Prior to scheduling any input transfer from (or output transfer to) the background memory from the buffers we have to take into account the output writes (or input reads) by the datapath into (or from) the buffers because the ports on the buffers are used by background memory as well as the datapath. This is done by first determining all the reads and writes by the datapath (`Reads[`*step*`]` and `Writes[`*step*`]` in Figure 6.3) from the datapath schedule, then adjusting the number of available ports (*avail_ports*) on the buffers in each step. Finally, the algorithm schedules *avail_ports* number of data values for transfer between the background memory and the buffers from the candidate list based on their urgency factor. This process is repeated for all the steps.

6.2.4 Discussion on the Algorithm

In summary, the salient features of the algorithm are that it

- minimizes the size of the I/O buffers, while meeting the port and timing constraints;
- transfers the inputs into the buffers from the background memory as late as possible and transfers the outputs to the background memory from the buffers as soon as possible, since keeping a value in the I/O buffers will contribute to the buffer size;
- checks if the value which is required again by the datapath, can be refetched, and
  - if yes, then the algorithm overwrites the value in order to save a memory location,
  - else, the algorithm stores the value in the buffers for later use.

The scheduling is guaranteed to succeed because the datapath scheduling step ensures that no bandwidth or I/O timing constraint will be violated in this step.

6.3 Background Memory Synthesis

Synthesis of the background memory is done in the same way as I/O buffer synthesis. First data transfers are scheduled and then module allocation is performed. As mentioned earlier, the module allocation step is not addressed in this thesis.

The background memory interacts with the on-chip buffers and the external I/O. After scheduling the data transfers between the buffers and the background memory, the read schedule from the background memory for all the input values and the write schedule into the background memory for the output values are known.

The remaining unknown timings for the data values include

- data write time to the background memory from the external world $\mathcal{T}_w^{bk}(d_i)$ for all the inputs d_i, and
- data read time from the background memory to the external world $\mathcal{T}_r^{bk}(d_o)$ for all the outputs d_o.

These timings are either fixed by the external I/O constraints or can be determined in the same way as the I/O buffers, as described in Section 6.2.

6.4 Summary

In this chapter we have described our approach for scheduling data transfers between the I/O buffers and the background memory. The data transfer scheduling is the first step towards synthesizing the storage structure.

SMASH schedules the data transfers between the I/O buffers and the background memory given the number of read and write ports on the buffers, the data accesses by the datapath, and the time interval during which the inputs and outputs are available, while minimizing the storage size.

The algorithm used in SMASH is also presented. The algorithm is based on list scheduling with a modified objective function. The scheduling technique is based on the urgency factor associated with each data value. The approach is extended to schedule the data transfers between the background memory and the external world if they are not already constrained by the user specification.

Chapter 7

Experimental Results

7.1 Introduction

This chapter describes the overall experimental process and the results. In the first section we briefly describe two layout experiments done prior to memory-synthesis research. The remaining chapter describes the experiments done using SMASH.

The experiments done prior to memory-synthesis research not only motivated us to address the memory synthesis problem but also showed us some area-time tradeoffs possible in the storage architecture of a design.

The experiment done using SMASH demonstrated the effectiveness of our techniques during high-level synthesis of such "real" designs. As mentioned in the introduction (Chapter 1), we developed our techniques and synthesis tools keeping "real" designs in mind. An integral part of this research was to demonstrate the effectiveness of SMASH during synthesis of such "real" designs.

7.2 Experiments Prior to Memory Research

As mentioned in the introductory chapter, our research was also motivated by our layout studies of automatic synthesis of an AR filter through the ADAM system [PGH91]. In a separate experiment we manually replaced the input latches with a RAM and compared the areas of the two implementations. These experiments are described below.

7.2.1 Experiment1 : Layout Studies of an AR Filter

In this experiment, our goal was to study the effects of wiring area and delay and unused area on final chip characteristics. We also wanted to demonstrate the existence of the cost-performance tradeoff curve in the actual layouts based on automatically synthesized designs, using both pipelined and non-pipelined design styles. This was the first published tradeoff study of its kind [PGH91].

The example chosen for this study was the AR lattice filter element, a design with a clear cost-performance tradeoff curve at the register-transfer level. The data-flow graph for this AR filter is shown in Figure 7.1.

For this experiment, we produced 6 non-pipelined and 6 pipelined RTL designs. We used MAHA [PPM86] to generate the schedule for non-pipelined designs and Sehwa [PP88] for pipelined designs. MABAL completed the RTL designs, which were then translated to Cascade Design Automation's Chipcrafter format through our netlist translation and expansion program. For non-pipelined designs the cost was measured as total area of the bounding box and performance as the delay through the active area of the chip. For pipelined designs, cost was measured as the area of the bounding box, and performance as the delay between the initiations of new data into the pipeline. We ran Chipcrafter with the OKI 1.2 micron, twin-well, double-layer metal CMOS ruleset and achieved layouts ranging from 20,000 to 30,000 transistors.

For each design, we measured individual contributions to final chip area. We then assessed whether these layouts still fit our cost-speed tradeoff curve.

Non-Pipelined Results

These designs varied in parallelism. A cost-performance tradeoff curve for non-pipelined datapaths is shown in Figure 7.2. This curve shows the register-transfer design points, and the physical parameters when layout is considered. The register-transfer design points included raw cell area and raw cell delays. Although the tradeoff curve is not a smooth convex surface when physical factors are taken into account, there are no faster designs which are cheaper, or slower designs which are also larger. The most parallel non-pipelined layout is shown in Figure 7.3.

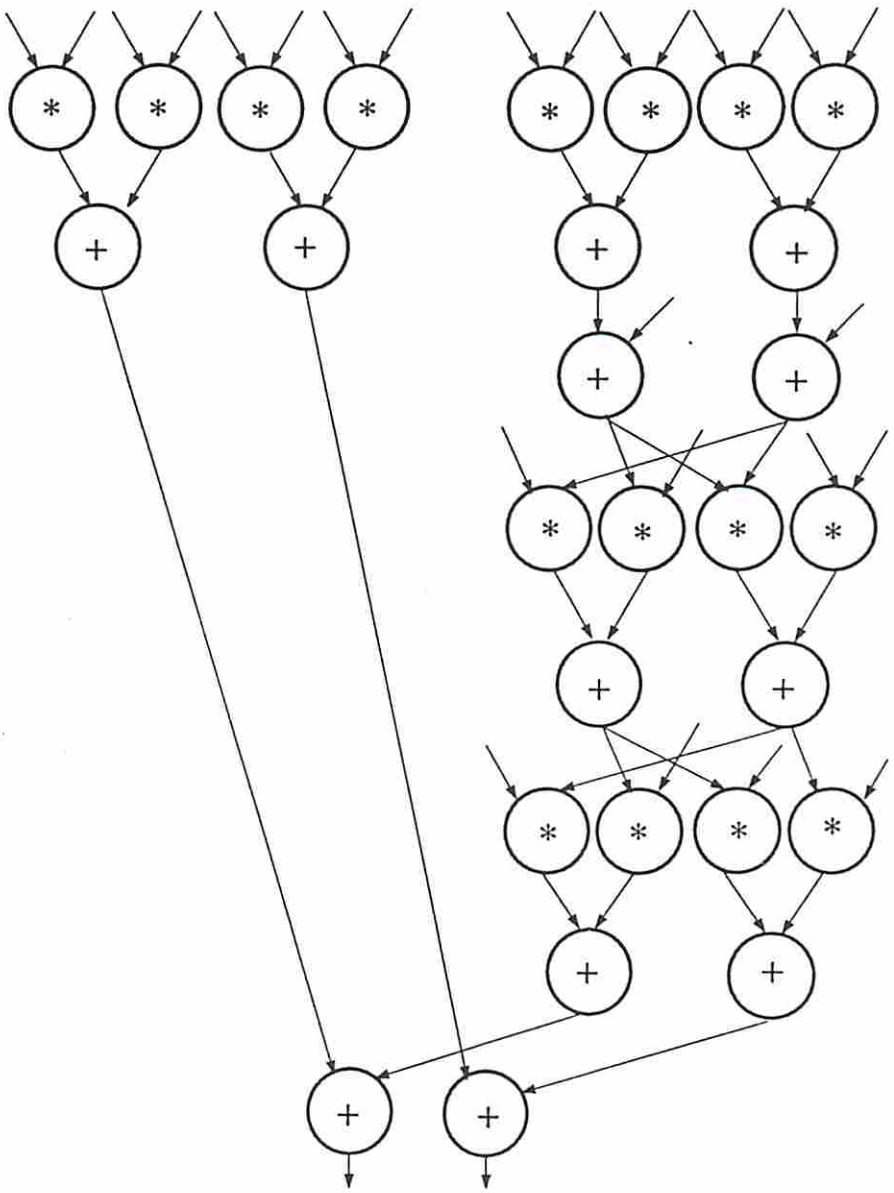


Figure 7.1: The AR Filter Dataflow Graph

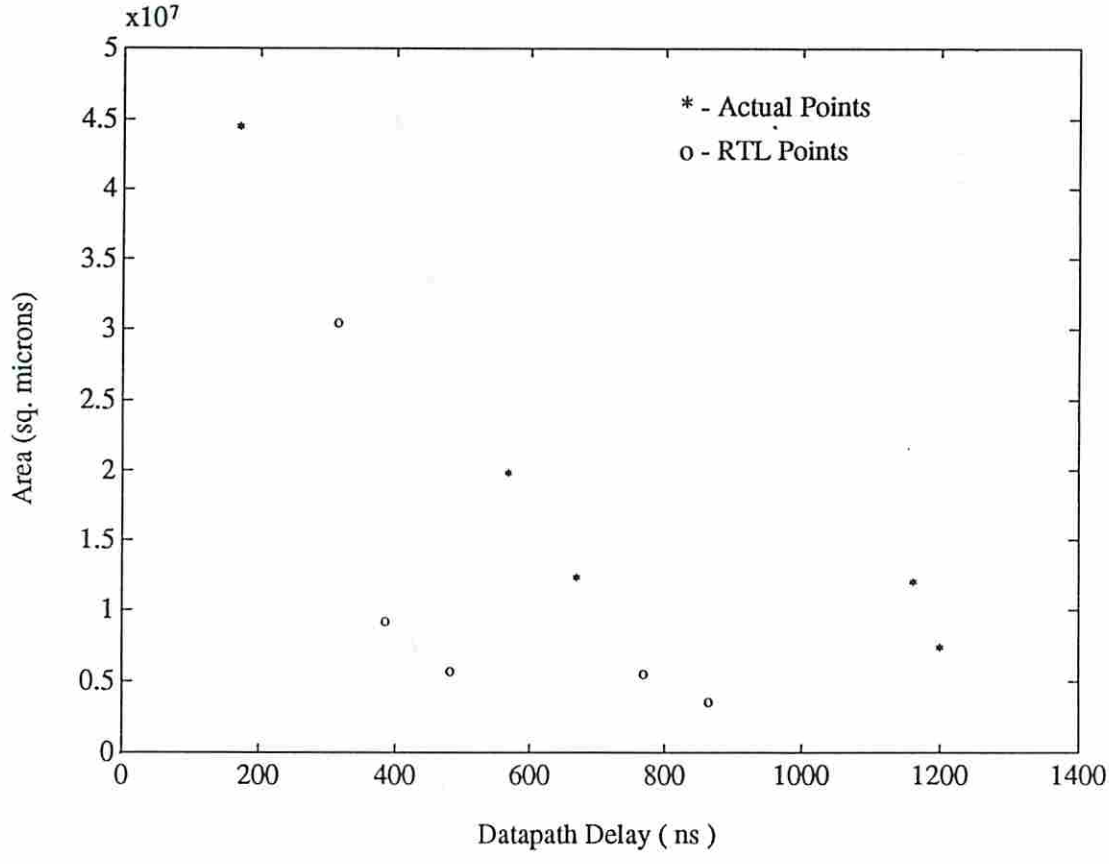


Figure 7.2: Cost-performance Tradeoff Curve for a 16-bit Non-Pipelined AR Filter Datapath Element

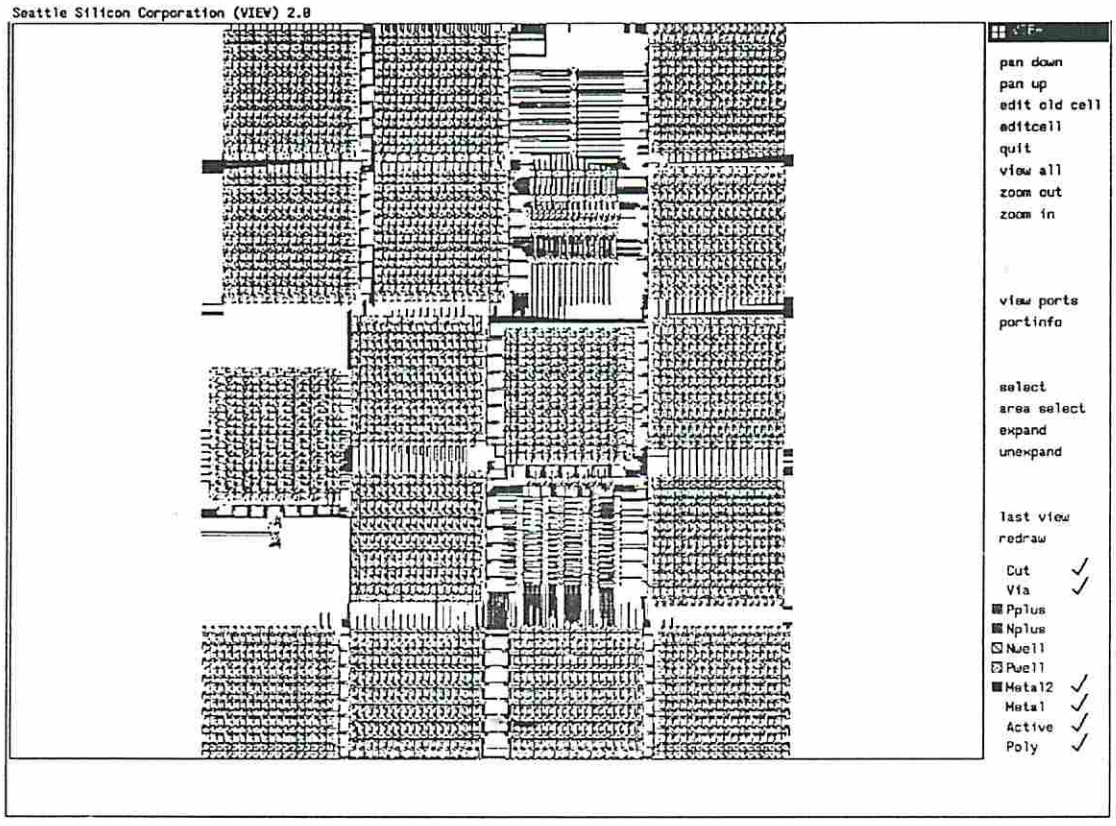


Figure 7.3: Layout of the Most Parallel Non-Pipelined Design

Design No.	No. of Control steps	Active Area $\times 10^6 \mu m^2$	Total Delay ns	Datapath Delay ns
1.	1	44.5	171	171
2.	4	23.4	NA	NA
3.	8	19.9	836	568
4.	10	12.4	1036	670
5.	16	12.0	1693	1161
6.	18	7.4	1781	1200

Table 7.1: Summarized Area - Delay Statistics of the Non-Pipelined Designs

Pipelined Results

The tradeoff curve in Figure 7.4 is for pipelined designs. The curve shows the register-transfer level design points and the actual points considering the layout. In these designs the basic clock period remains almost the same and therefore the delay depends on the initiation interval of the circuit.

Design Number	No. of Control steps	Active Area $\times 10^6 \mu m^2$	Initiation Interval ns
1.	1	64.1	63
2.	4	20.6	265
3.	6	18.5	410
4.	8	12.9	529
5.	12	11.0	830
6.	16	10.0	1185

Table 7.2: Summarized Area - Delay Statistics of the Pipelined Designs

Discussion

All these layouts described above did not include the input latches. When these input latches are included (as most of the traditional synthesis systems do) the chip area becomes very high. Furthermore, without proper I/O storage and management,

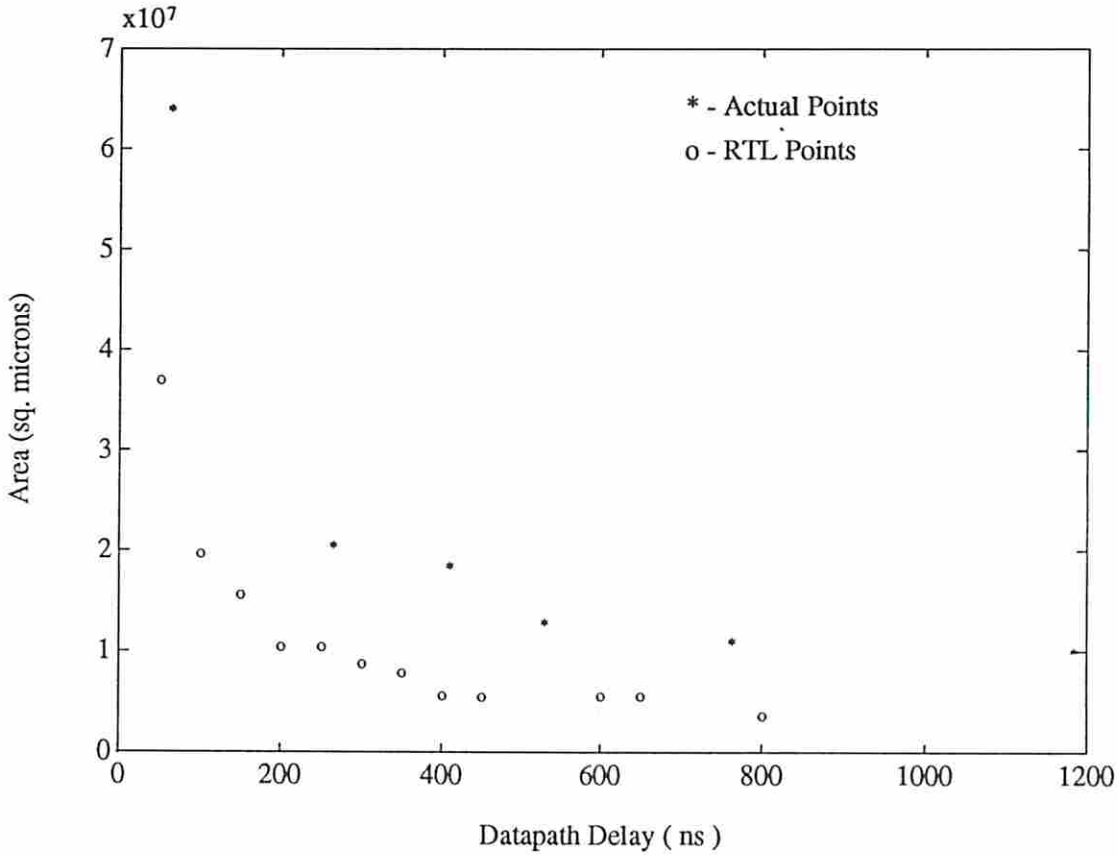


Figure 7.4: Overall Cost-performance tradeoff curve for a 16-bit Pipelined AR Filter Datapath

Design Number	No. of Steps	Design with Registers		Design with RAM		Improvment %
		Register Area $\times 10^6 \mu m^2$	Chip Area $\times 10^6 \mu m^2$	Reg. + RAM* Area $\times 10^6 \mu m^2$	Chip Area $\times 10^6 \mu m^2$	
1.	4	2.28	26.83	1.48	22.87	14.76
2.	8	2.28	24.38	1.16	16.96	30.43
3.	10	2.28	17.74	0.91	11.39	35.79
4.	16	2.28	17.30	0.83	9.88	52.90
5.	18	2.28	14.76	0.75	6.64	55.01

Table 7.3: Storage Area Statistics of Layouts

the number of I/O pins required is unnecessarily very high and results in impractical designs. Buffering I/O with a RAM instead of registers subsequent to these experiments (described in the next section) halved the area of the smallest design.

An important outcome of this simple experiment was the need for high-level synthesis systems to take into account a number of factors, such as efficient buffering and I/O data management that were ignored in the past. Only then we can use them for real designs and make them acceptable to industry.

7.2.2 Experiment2 : input latches vs. input RAM

In a separate experiment, we included the input latches and generated the layouts for 5 non-pipelined AR filter designs with varying parallelism. Then we manually merged these input latches into a RAM and again generated the layouts. On analyzing the layouts we observed a significant improvement in area where registers were merged. In fact, the saving was as high as 50% in some cases. The experimental results are given in Table 7.3. This experiment motivated us to explore memory design tradeoffs possible in this dimension.

*RAM AREA $= 0.5 \times 10^6 \mu m^2$.

7.3 Experiments using SMASH

These experiments were designed to demonstrate the following aspects of our research:

- the capabilities of SMASH in combining the datapath scheduling with I/O access from the buffers,
- improvement in datapath schedules when storage-related constraints are also considered,
- existence of the cost-performance tradeoff in the storage architecture (cost being a function of R/W ports, storage size, and BW_{on-off}),
- the capabilities of the software in terms of handling "real" designs which include descriptions of arrays, loops and conditional branches in the input behavioral specification, and
- the role of SMASH in a system-level design environment like USC.

These experiments have been organized into three categories:

1. experiments with representative high-level synthesis workshop benchmarks,
2. rapid prototyping of a JPEG still image compression system, and
3. enhanced design of the components of the JPEG still image compression system.

The input VHDL behavioral descriptions for the designs were manually written. As mentioned earlier, closeness to "reality" was always emphasized in our research, therefore the module library was generated using a commercial silicon compiler by first laying out different modules; and then by characterizing them individually. The details of the module library are given in the following section.

7.4 Module Library

We used the Epoch Silicon Compiler from Cascade Design Automation to design our library elements. The technology used is based on Epoch's module library and

uses Orbit's 1.2 micron technology. The design as well as the area-delay analysis was done using Epoch's tools. Since, SMASH requires *functional modules* and *storage modules*, the two libraries were developed as described below.

7.4.1 Functional Modules

The functional modules were characterized by (*i*) cost, and (*ii*) execution delay. We designed the required functional operators using Epoch and analyzed the designs to obtain the area and delay of each module. These parameters are summarized in Table 7.4.

Module name	Area (sq. microns)	Delay (ns)
Multiplier	2398490	55
Divider	2398490	55
Adder	81558	30
Subtractor	81558	30
Comparator	81558	30
2-1 Mux	24858	3
3-1 Mux	69834	3
4-1 Mux	76028	3
Distribute*	0	0
Join*	0	0

*dummy module

Table 7.4: Module Library used by SMASH.

7.4.2 Storage Modules

The storage modules were characterized by (*i*) cost of storage per word, (*ii*) maximum storage capacity, and (*ii*) number of read and write ports on the module. This required us to characterize cost as a function of size. To do that we generated several implementations of various types of storage modules (like register files and on-chip RAMs), then by interpolation we derived the relationship between size and cost.

The following storage modules were used in our library:

- Register: The area required by a 16-bit wide register is $37821 \mu m^2$.
- Register file with 1 read port and 1 write port: The following cost function was derived from the layouts of 1R/1W register files with varying sizes (as shown in Plot 7.5).

$$Cost_{Area}(size) = 53707 \times \left(\left\lceil \frac{size}{8} \right\rceil \times 8 \right) - 14405 \ \mu m^2$$

  $Cost_{Area}(size)$ is a step function because Epoch's library required the size of the register file to be in multiples of 8. And the maximum allowed size was 64 words per module.

- Register file with 2 read ports and 1 write port: For 2R/1W register file, $Cost_{Area}(size)$ (from Plot 7.6) is

$$Cost_{Area}(size) = 84606 \times \left(\left\lceil \frac{size}{8} \right\rceil \times 8 \right) - 19135 \ \mu m^2$$

  where size varied in multiples of 8, up to 64 words per register file.

- On-chip RAM with 1 read port and 1 write port: For 1R/1W RAM module, $Cost_{Area}(size)$ (from Plot 7.7) is

$$Cost_{Area}(size) = 9043 \times \left(\left\lceil \frac{size}{2} \right\rceil \times 2 \right) + 147480 \ \mu m^2$$

  where size should an even number (between 4 and 4096) for each RAM.

- On-chip RAM with 2 read ports and 1 write port: For 2R/1W RAM module, $Cost_{Area}(size)$ (from Plot 7.8) is

$$Cost_{Area}(size) = 18329 \times \left(\left\lceil \frac{size}{2} \right\rceil \times 2 \right) + 303680 \ \mu m^2$$

  where size should be an even number (between 4 and 1024) for each RAM.

Epoch's library also provides several other port configurations (like 2R/2W, 3R/1W, 3R/2W, 3R/3W, 4R/1W, and 4R/2W) for RAM modules, but these were

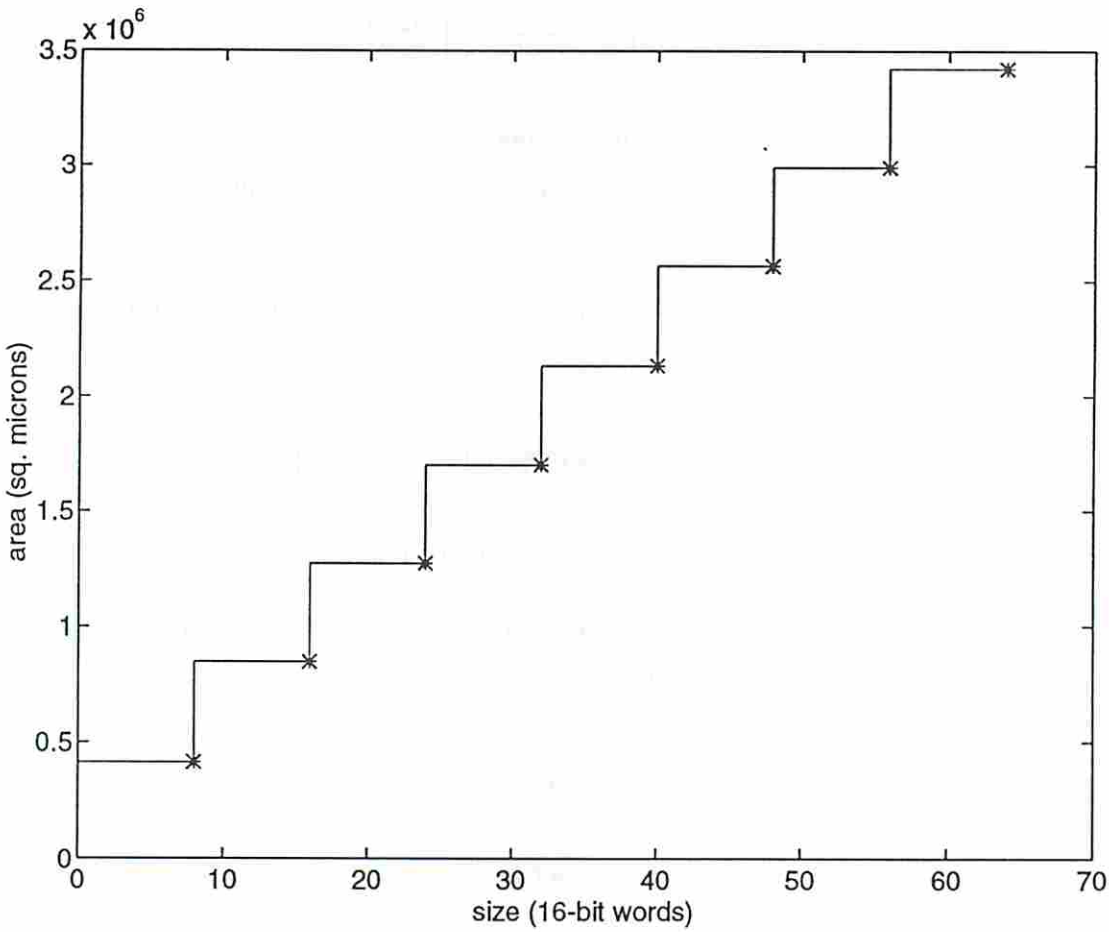


Figure 7.5: Area *vs.* size for 1R/1W Register file in EPOCH

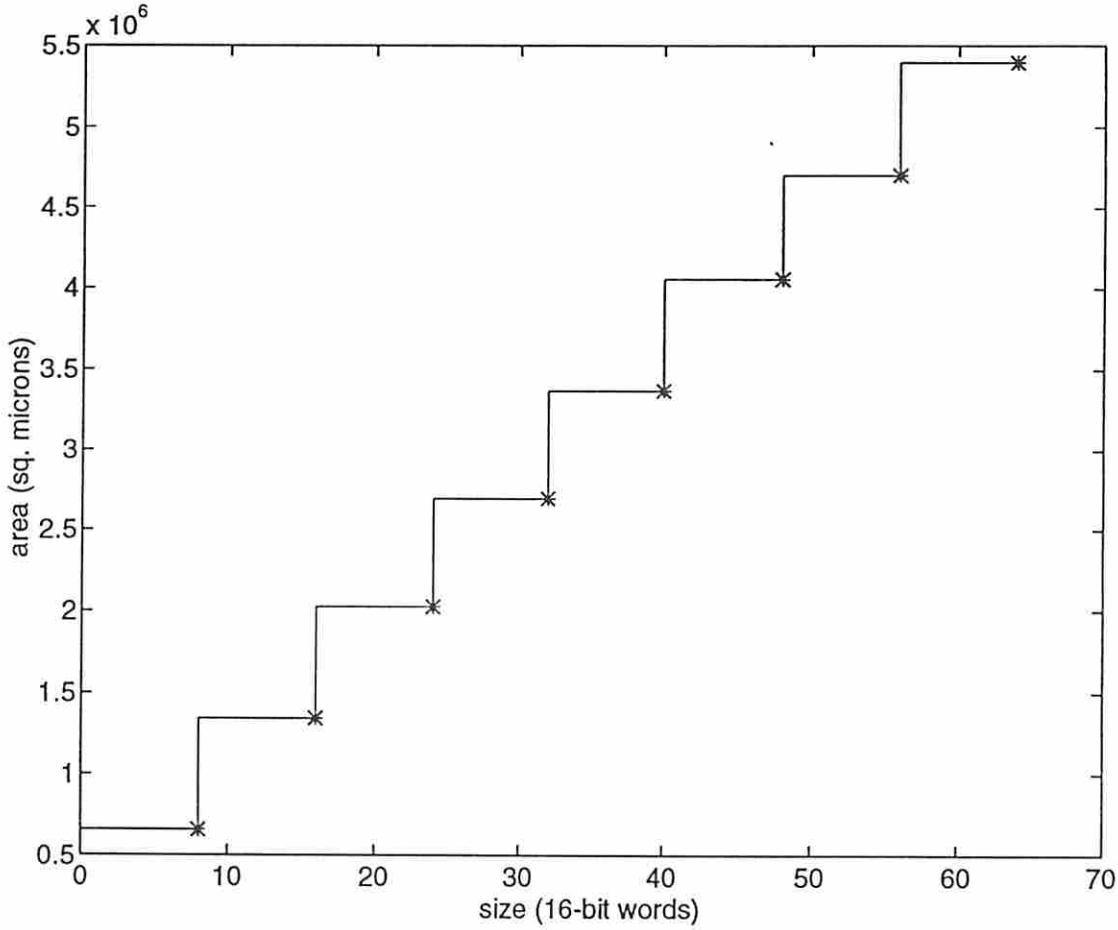


Figure 7.6: Area *vs.* size for 2R/1W Register file in EPOCH

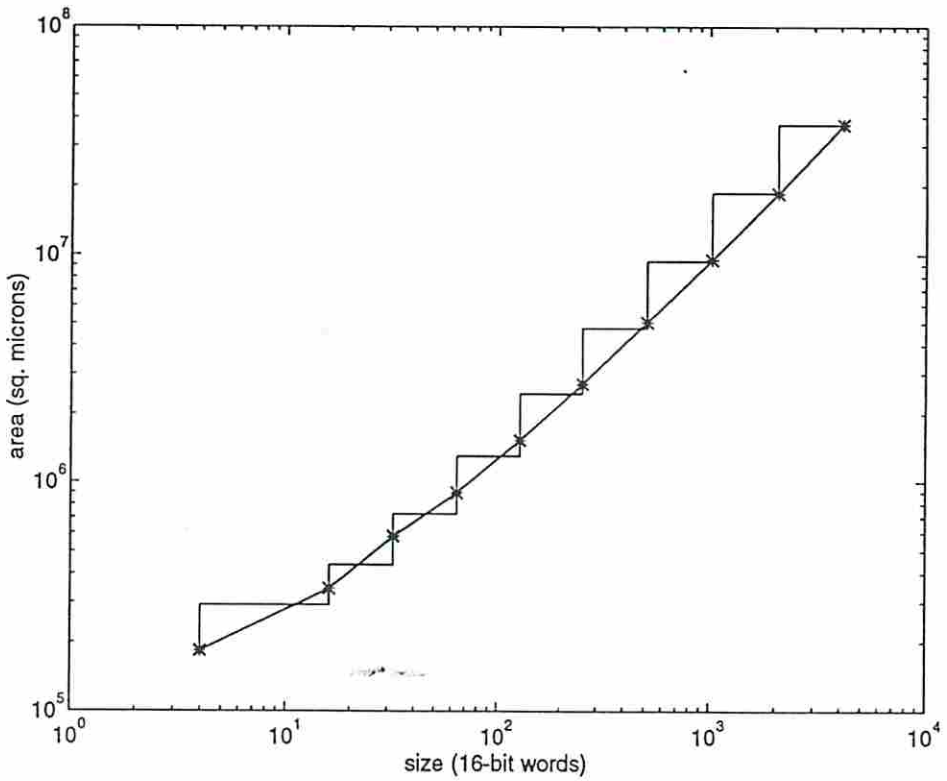


Figure 7.7: Area *vs.* size for 1R/1W RAM Module in EPOCH

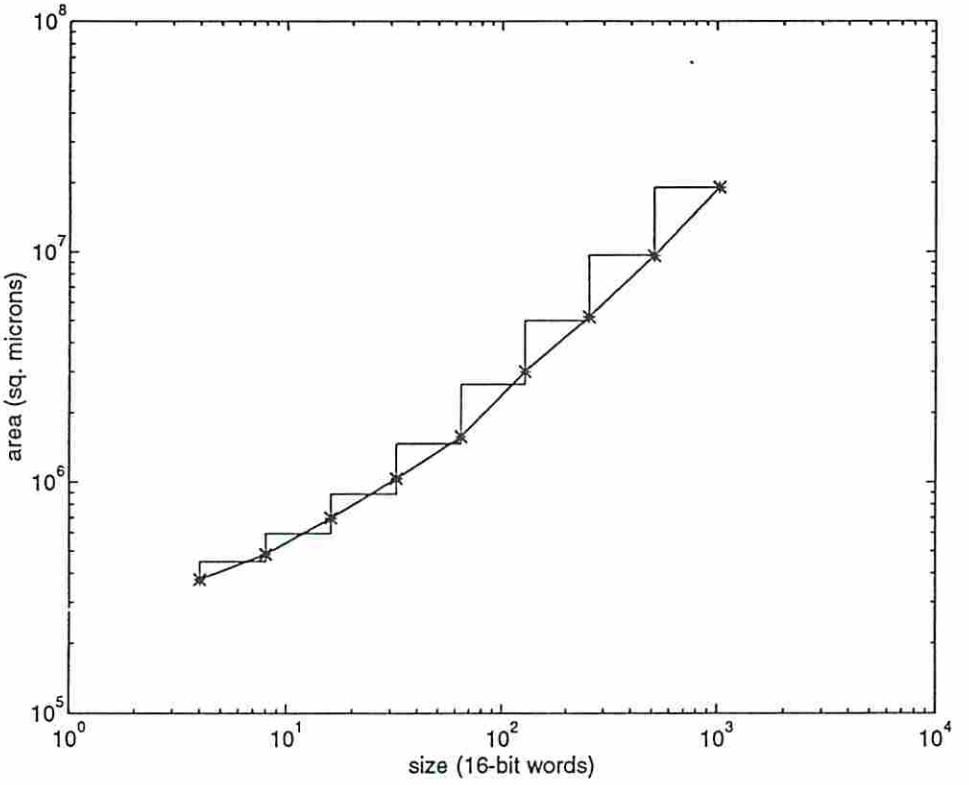


Figure 7.8: Area *vs.* size for 2R/1W RAM Module in EPOCH

not used in our experiments. These modules will be characterized and included in the library in future.

The clock cycle was assumed to be 30ns.

7.5 High-Level Synthesis Benchmark Examples

From the High-Level Synthesis Workshop benchmarks, the following three examples were synthesized to illustrate the above mentioned capabilities of SMASH [GP94]:

1. a second-order differential equation solver [PK89],
2. an AR filter element [PGH91], and
3. an Elliptic wave filter element.

To simplify the experiments, it was assumed that the input data was stored in an off-chip RAM before processing. The software was executed with priority on performance optimization while meeting the area constraint. We also generated some designs without considering storage-related constraints (BW_{on-off} and R_{buf}/W_{buf}). SMASH generated several designs with varying parameters for these examples. These implementations are summarized in Tables 7.5, 7.6, and 7.7. The parenthesized storage architecture parameters indicate the designs without storage-related constraints. These parameters were manually determined when the designs were mapped on our target architecture.

7.5.1 Differential Equation Example

In this section, we analyze the differential equation example in detail. Five different implementations of the differential equation example were generated. These implementations varied in cost and performance. These designs are briefly described below.

- *Design 1:* This design is a high performance but expensive (high cost) design. This design was synthesized without considering the storage-related parameters so that a comparison could be made with the designs where these parameters were also optimized. This design was synthesized by executing SMASH

Design no.	Input constraints		Software output				
	Functional area $10^6\mu m^2$	$BW_{on\text{-}off}$ (words/ cycle)	# ports R_{buf}	# ports W_{buf}	Buffer size words	Functional resources	Execution time ns
1	7.5	(4)	(4)	(2)	NA (6)	<, +, -, 3 *	240
2	7.5	2	4	2	8	<, +, -, 3 *	240
3	6.0	(4)	(4)	(1)	NA (5)	<, +, -, 2 *	300
4	6.0	2	3	1	6	<, +, -, 2 *	300
5	6.0	1	3	1	7	<, +, -, 2 *	330
6	3.0	2	2	1	4	<, +, -, *	450
7	3.0	1	2	1	6	<, +, -, *	480

Table 7.5: Parameters from SMASH for Differential Equation Example

Design no.	Input constraints		Software output				
	Funtional area $10^6\mu m^2$	$BW_{on\text{-}off}$ (words/ cycle)	# ports R_{buf}	# ports W_{buf}	Buffer size words	Functional resources	Execution time ns
1	10.0	(6)	(6)	(2)	NA (8)	2 +, 4 *	390
2	10.0	4	4	2	6	2 +, 4 *	390
3	10.0	3	4	2	10	2 +, 4 *	420
4	5.0	2	4	2	8	2 +, 2 *	600

Table 7.6: Parameters from SMASH for AR Filter Example

Design no.	Input constraints		Software output				
	Functional area $10^6\mu m^2$	BW_{on-off} (words/ cycle)	# ports R_{buf}	# ports W_{buf}	Buffer size words	Functional resources	Execution time ns
1	10.0	(3)	(3)	(4)	(10)	4 +, 4 *	540
2	10.0	3	3	4	10	4 +, 4 *	540
3	7.5	2	2	2	7	3 +, 3 *	600
4	5.5	2	3	3	8	3+, 2 *	690
5	5.0	2	2	1	4	2 +, 2 *	720

Table 7.7: Parameters from SMASH for Elliptic Wave Filter

without constraining BW_{on-off} and R_{buf}/W_{buf}. The parenthesized storage architecture parameters shown in the table were manually determined from the output.

- *Design 2:* This design was generated by executing SMASH under BW_{on-off} constraints. This design requires the same number of functional resources as design 1 and has the same performance. However, BW_{on-off} requirement is 50% lower in this case compared to design 1. An analysis of the operation schedule showed that SMASH achieved this by scheduling the operations in such a manner that the operations requiring inputs were postponed till the inputs were available and other operations were moved ahead.

- *Design 3:* Again, this design was synthesized without considering the storage-related parameters. It is a slower but cheaper design compared to the above two designs.

- *Design 4:* This design has the same functional cost and performance as Design 3. However, BW_{on-off} is 50% lower, and even the number of read ports on the buffers R_{buf} is lower.

- *Design 5:* This design was synthesized to show the tradeoff in BW_{on-off}. Here, the functional cost is the same as designs 3 and 4 but BW_{on-off} is lower,

hence the longer execution time. The lower BW_{on-off} also requires inputs to be prefetched and stored in the buffers, therefore, the buffer size is higher than design 4.

- *Design 6:* This is a slow and cheap design. There is substantial of resource sharing and therefore, the execution time is longer. The buffer size decreased in this case, as longer execution time provided enough flexibility for data transfers. SMASH could schedule the operations such that data could be transferred whenever required without storing them in the buffers.

- *Design 7:* Again, this design has the same functional cost as design 6, but lower BW_{on-off}. As a result, some operations were postponed until the required data could be fetched, hence longer execution time. Also, some inputs were prefetched and stored in the buffers to avoid further delay in execution, hence there is higher buffer size than Design 6.

Designs 1 and 3 *vs.* 2 and 4 respectively clearly demonstrate the effectiveness of our techniques in reducing the unnecessary storage cost. Schedules with storage constraints achieved the same performance with the same functional area while reducing BW_{on-off} and R_{buf}/W_{buf}. This was because SMASH distributed the data requirement uniformly and overlapped the data transfer with the execution.

Designs 5, 6 and 7 were generated to demonstrate the tradeoff curve existing in the storage architecture. Designs 4 vs. 5 and 6 vs. 7 show that for the same functional resources, varying storage parameters resulted in different execution delays. Designs 5 and 7 are slower than the designs 4 and 6 respectively. Analysis shows that this was because of the limited bandwidth allowed in 5 and 7. Furthermore, in designs 5 and 7, as the I/O transfer was the bottleneck, unnecessary transfers were avoided by storing the data value in the buffers for further use, hence the buffer size increased. Note that though these slower designs require bigger buffer sizes, they are still cheaper because of the lower bandwidth requirement (which in turn also determines the number of pins on the chip).

Figure 7.9 shows the datapath schedule in detail for design 5, excluding the R/W nodes. This figure also does not include the on-chip off-chip data transfer schedule, as that task is performed in the second design step. The results obtained from the

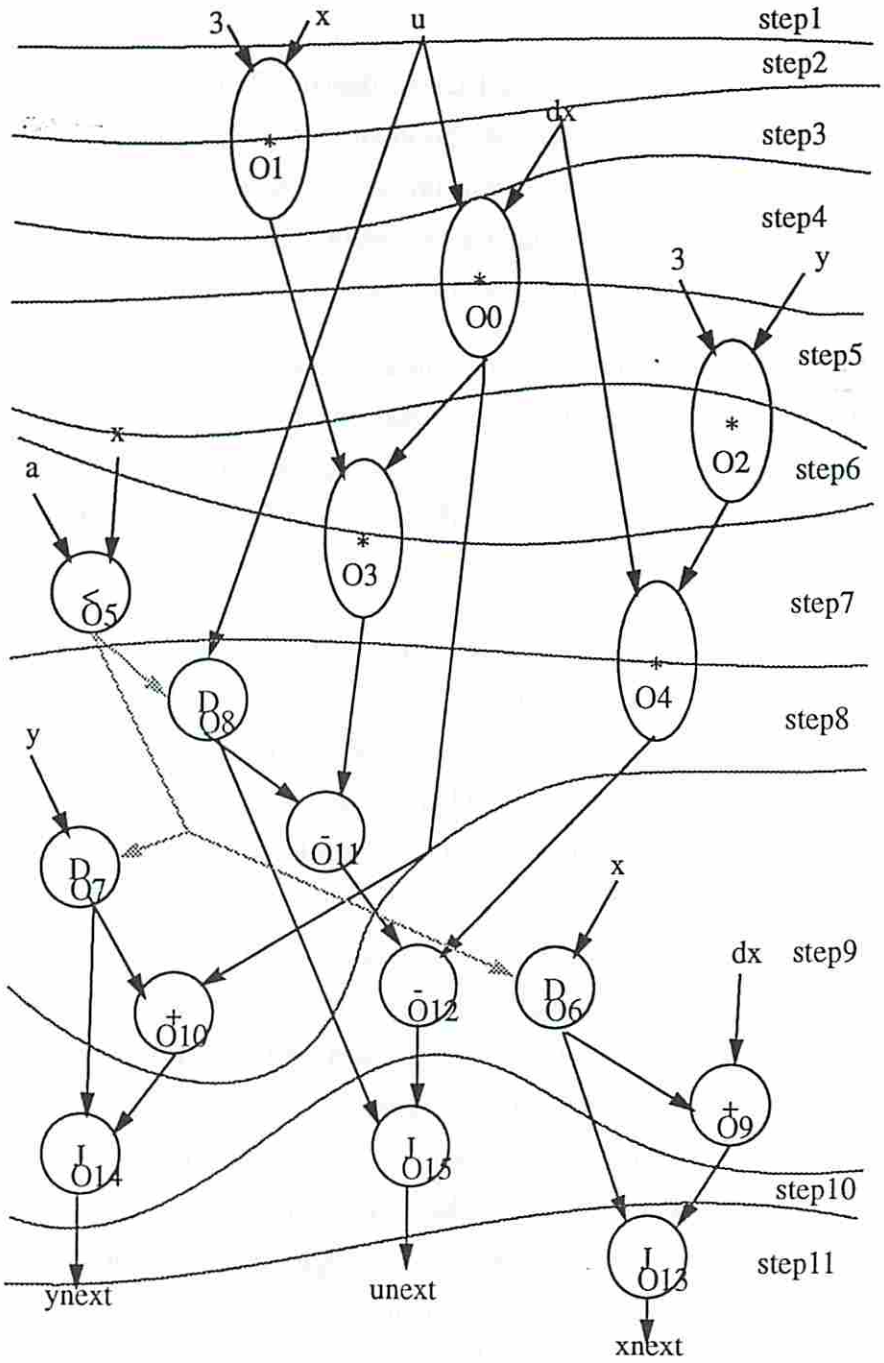


Figure 7.9: Scheduled CDFG for Design 5 (2nd-order Differential Equation).

Step number	Buffers-Off chip data trasfer		Buffer contents	Buffers-Datapath data transfer	
	Write into buffer	Read from buffer		Read from buffer	Write into buffer
1	x	-	x	-	-
2	3	-	3, x	3,x	-
3	dx	-	3, dx	-	-
4	u	-	3, dx, u	u,dx	-
5	y	-	3, dx, y	3,y	-
6	x	-	dx, x, y	-	-
7	a	-	dx, x, y, a	a,dx,x	-
8	u	-	dx, x, y, u	y,u	-
9	-	ynext	dx, x	dx,x	ynext
10	-	unext	-	-	unext
11	-	xnext	-	-	xnext

Table 7.8: Data transfer schedule for design 5 (2nd-order differential equation).

data transfer scheduling software are summarized in Table 7.8. Observe how inputs 'x' and 'u' are read twice into the buffer because there was an empty 'slot' available for re-transfer before they were required again, and storing them in the buffers would have resulted in an increase in the buffer size. On the other hand, 'dx' was stored in the buffers for future use because its re-transfer was not possible before the given time. Also, note that after re-fetching 'x' in step 6 (for consumption in step 7) it was saved in the buffers for step 9. Fetching 'x' again in step 8 (for step 9) instead of 'u', and saving 'u' in the buffers could have been another option, but that would have resulted in a buffer size of 5 words (in step 7). Therefore, SMASH decided to save 'x' and re-fetch 'u'.

7.5.2 AR Filter and Elliptic Wave Filter Examples

The AR filter example showed the same tradeoffs as shown by the differential equation example. Designs 1, 2, and 3 had the same functional cost but varying storage cost and performance. Design 1 was produced without considering the storage cost, therefore it resulted in high storage cost. Both BW_{on-off} and R_{buf} were high in this case. When considering the storage cost during synthesis (design 2), SMASH could reduce both the BW_{on-off} and R_{buf} while achieving the same performance with the same functional hardware. In design 3, the BW_{on-off} was further reduced but with an increase in the execution time. Design 4 was synthesized with tighter area and BW_{on-off} constraints. It is the slowest and cheapest of the four designs.

The elliptic wave filter example was also synthesized to show above mentioned tradeoffs. It clearly shows a tradeoff between the cost (both the functional cost as well as storage cost) and the execution time.

7.5.3 Discussion

To summarize, these designs showed the tradeoffs in the storage design in addition to the tradeoffs in the datapath. They showed why storage-related constraints must be considered during datapath synthesis to get better designs. They also showed SMASH's capability of determining the parameters for storage structure and scheduling the required data transfers.

7.6 Rapid Prototyping of a JPEG Still Image Compression System

The objective of this system-level experiment [CGDBP94, GCDBP94] was twofold: first, to synthesize more realistic designs with SMASH, and second, to illustrate the way SMASH can be used in a system-design environment.

In this experiment, we chose to focus our design activity on a standard for still image compression, JPEG [Wal91]. We chose to synthesize the system shown in Figure 7.10, and used SMASH along with other system-level tools (ProPart and SOS) to explore design possibilities.

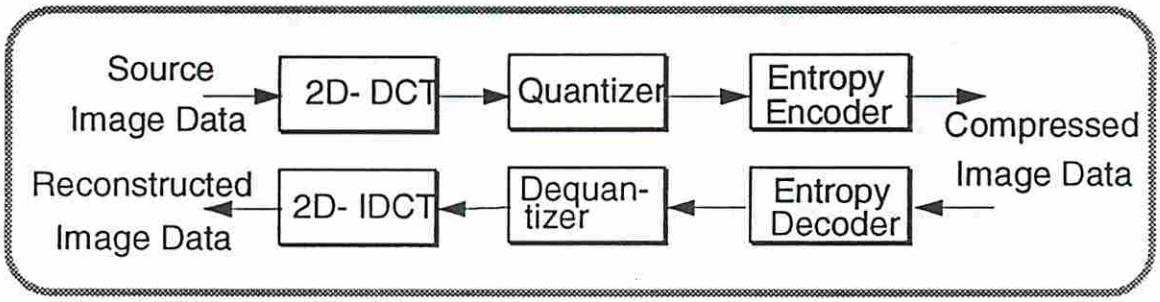


Figure 7.10: JPEG Still Image Compression System

Figure 7.11 shows the design flow used for this experiment. It is important to note that the design flow has some bottom-up portions, which represent the flow between the application of each tool, each of which operates in essentially a top-down fashion. Thus the design flow is both top-down and bottom-up.

We began with the synthesis of the DCT (Discrete Cosine Transform) function. The 2D-DCT was decomposed into repeated row-column 1D-DCT's prior to the application of the system-level tools (Figure 7.12). The 1D-DCT macro was

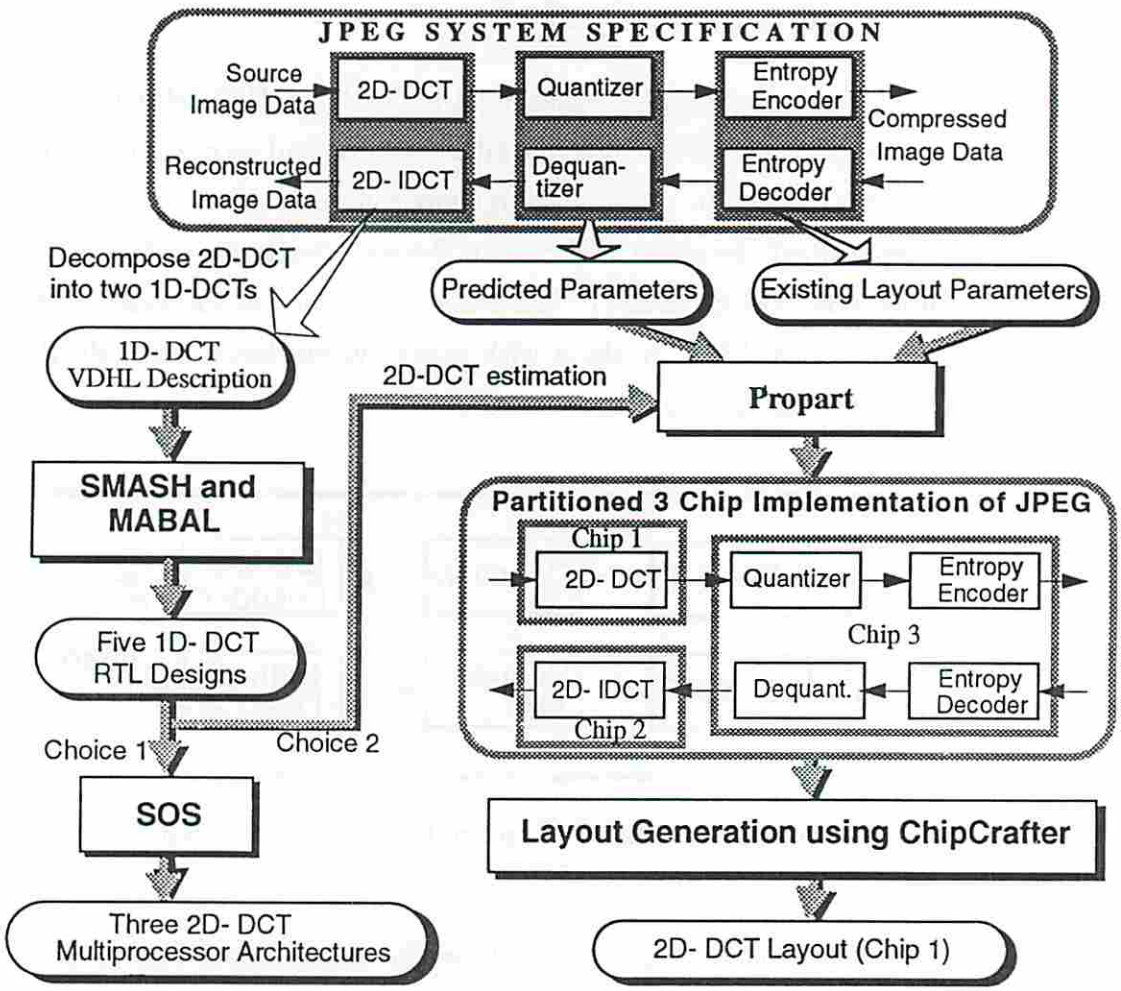


Figure 7.11: Design Flow for Still Image Compression System Example

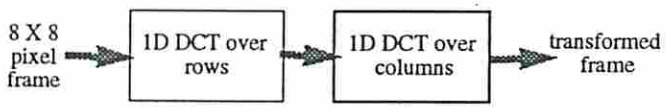


Figure 7.12: 2D DCT implementation from 1D DCTs

synthesized first and used to construct a 2D-DCT, clearly a bottom-up step. The 8-point 1D-DCT matrix is as follows[FLS+92]:

$$\begin{bmatrix} X_0 \\ X_4 \\ X_2 \\ X_6 \\ X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} = \begin{bmatrix} d & d & 0 & 0 & 0 & 0 & 0 & 0 \\ d & -d & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & b & f & 0 & 0 & 0 & 0 \\ 0 & 0 & f & -b & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & c & e & g \\ 0 & 0 & 0 & 0 & c & -g & -a & -e \\ 0 & 0 & 0 & 0 & e & -a & g & c \\ 0 & 0 & 0 & 0 & g & -e & c & -a \end{bmatrix} \times \begin{bmatrix} x_0 + x_7 + x_3 + x_4 \\ x_1 + x_6 + x_2 + x_5 \\ x_0 + x_7 - x_3 - x_4 \\ x_1 + x_6 - x_2 - x_5 \\ x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix}$$

This 1D-DCT description was translated into a behavioral VHDL description. SMASH was used to generate five schedules from this VHDL description. The module library used is shown in Table 7.4. These datapath schedules with varying cost and performance are shown in Table 7.9. SMASH also determined buffer size and bandwidth requirement as shown in Table 7.9.

Design number	Input constraints		SMASH output				
	Functional area (10^6 µm^2)	$BW_{on\text{-}off}$ (words/ cycle)	R_{buf}	W_{buf}	Buffer size (words)	Functional resources	Execution time (cycles)
1	30	5	12	6	21	3 +, 4 -, 12 *	8
2	20	3	6	5	14	3 +, 3 -, 8 *	13
3	18	3	4	4	11	3 +, 3 -, 7 *	15
4	10	3	3	3	7	2 +,2 -, 4 *	20
5	6	2	3	3	9	2 +, 2 -, 2 *	26

Table 7.9: 1D-DCT design parameters obtained using SMASH

The 1D-DCT schedules were then processed by another ADAM tool called MABAL [KP90] to generate the RTL netlists. These netlists were analyzed to obtain the area characteristic of the datapath as shown in Table 7.10. The area for functional units, multiplexers and registers was determined from the netlists, and wiring

area was estimated manually using a rule-of-thumb which we observed in our earlier experiments [PGH91].

Design Number	Functional area (10^6 µm^2)	Interconnect area (10^6 µm^2)		Total area (10^6 µm^2)
		Muxes and Registers	Wiring	
	A	B	C = 2(A+B)	
1	29.35	3.74	66.18	99.27
2	19.68	3.87	47.09	70.63
3	17.20	4.05	42.50	63.74
4	9.92	3.67	27.18	40.77
5	5.12	4.12	18.48	27.73

Table 7.10: 1D DCT RTL designs from MABAL

Next, for system-level design we estimated the performance and silicon area of the remaining components of the system. A 2D-DCT architecture consisting of two 1D-DCT modules and an 8 × 8 frame buffer was selected as shown in the literature [FLS+92]. The worst-case datapath delay was used to calculate the performance for each implementation with a two-phase non-overlapping clocking scheme. The quantizer performance and silicon area were estimated similarly, and the parameters used are comparable to those reported in the literature [FLS+92]. For the Huffman codec, parameters of an existing chip were used [PP93].

After estimating the performance and silicon area of all the parts in the compression system, it was partitioned by ProPart. ProPart selected the 2D-DCT design which is constructed using the second 1D-DCT design produced by SMASH.

Finally, the layout of the 1D-DCT macro and 2D-DCT chip were generated using Epoch from Cascade Design Automation. These layouts are shown in Figures 7.13 and 7.14, and the analysis of the area distribution is shown in Table 7.11.

A comparison of our chip set with others is shown in Table 7.12 [CS93]. Since we obtained the Huffman coding chip parameters from another source, they are only compared here to show that the parameters we are using are comparable to those in the literature. The die we did design, the DCT, has somewhat larger die size than

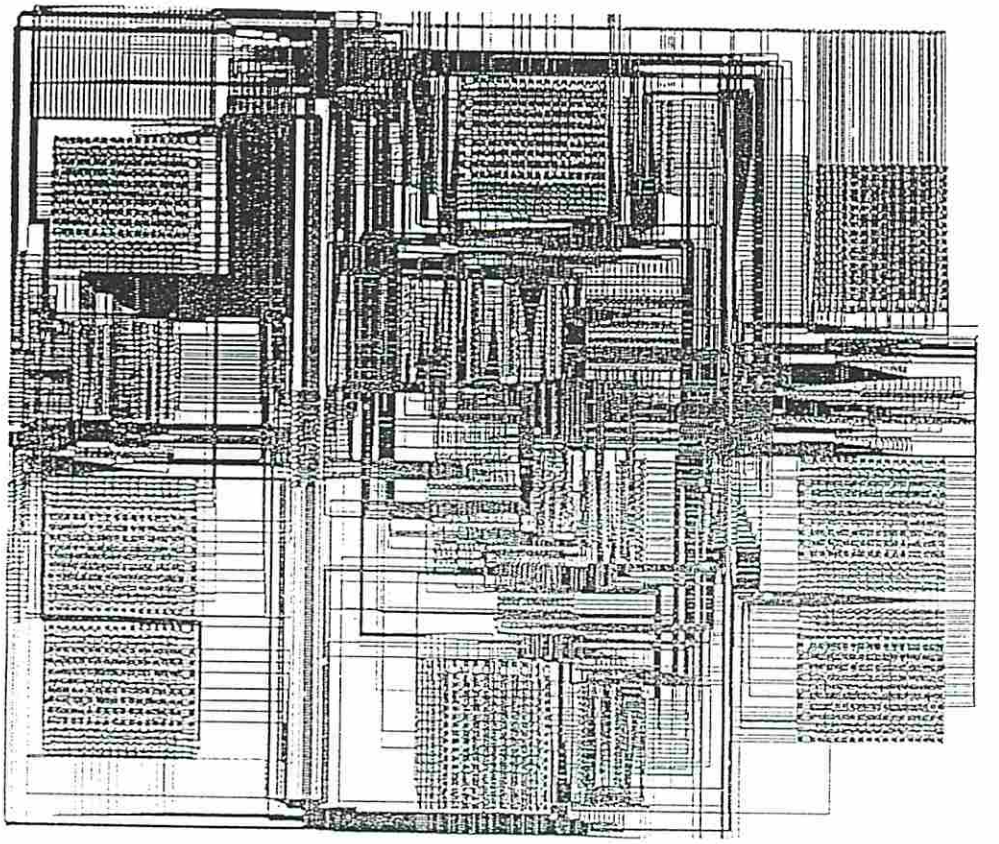


Figure 7.13: Layout of 1D DCT module

Area $10^6\mu m^2$	Total	Functional	Controller	Interconnect	
				Muxes + Registers	Wiring
1D-DCT	83.72	19.83	1.03	4.67	58.19
2D-DCT	209.94	39.66	2.06*	13.91**	154.31

* controller for the frame-buffer and I/O pads not included
** a 64 word on-chip RAM included.

Table 7.11: Area analysis for the layouts

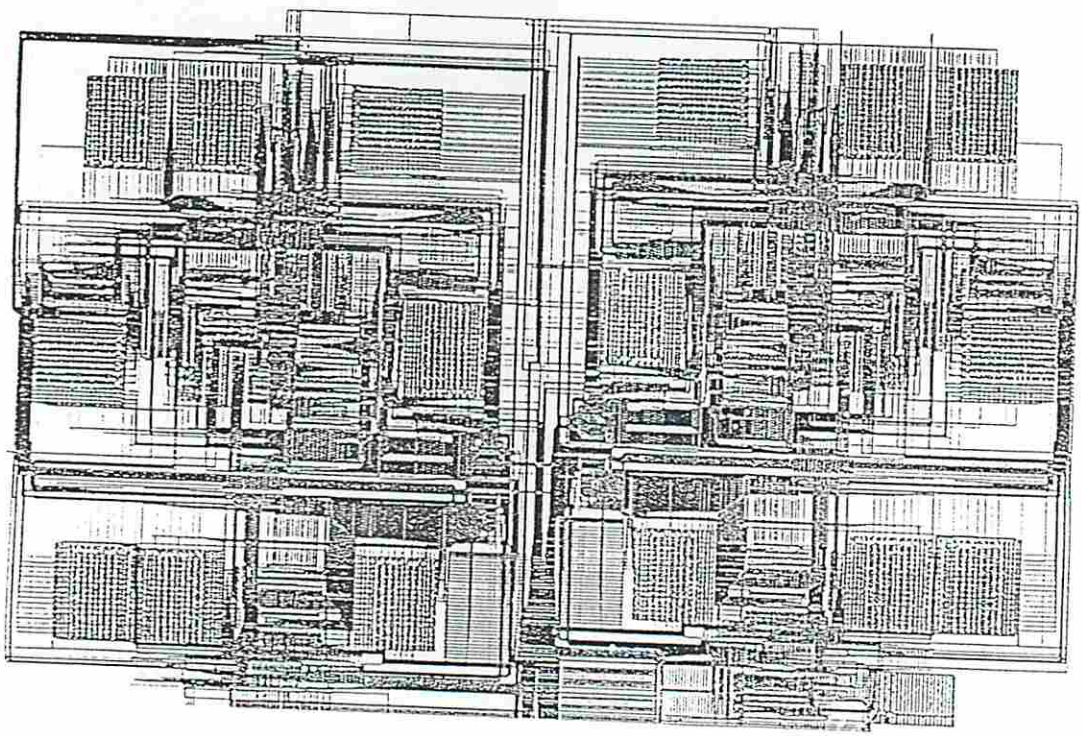


Figure 7.14: Layout of 2D DCT chip

the industrial chips, but the performance was comparable. The technologies used by the industrial chips was not mentioned in the referenced article [CS93], so we were not able to determine whether our 1.2 micron CMOS technology was inherently larger than the industrial technologies.

	Area (mm^2)		
	Ours	Bellcore	LSI
2D DCT/IDCT	11.8 x 17.7	10.7 x 10.2	9.5 x 9.5
Quant./Dequant.	(4.4 x 4.4)	9.0 x 9.0	9.9 x 9.9
Encoder	(3.5 x 3.5)	6.6 x 6.2	7.4 x 7.4
Decoder	(3.5 x 3.5)	7.5 x 8.4	7.4 x 7.4

Table 7.12: Chip-set parameters

To search the design space for a wider range of implementations, the SOS multiprocessor synthesis tool was used. Cost/performance parameters predicated from the RTL netlists for the 1D-DCT implementations (Table 7.10) were input to SOS, so that it could choose from all five 1D-DCT implementations. The design space was searched for various performance constraints with the objective of minimizing the cost. The sets of 1D-DCT implementations selected by SOS for various timing constraints are shown in Table 7.13.

7.6.1 Discussion

The primary outcome of the experiment was a clear understanding of how SMASH can be used in a system-level design environment. It showed how high-level synthesis tools like SMASH can be used not only in designing the systems but also in performing 'what-if' analysis during the design process. Even if the designer doesn't want to use high-level synthesis for actual design, he/she can perform a quick 'what-if' analysis of his decisions using these tools. Thus, the design space can be explored very quickly.

Design number	Input to SOS	Outputs from SOS			
	Time constraint	Processors*	Cost	Execution time/ 8x8 frame	Pixel rate
	ns		10^6 µm^2	ns	10^6pixel/s
1	6400	4 P5	110.90	6350	10.08
2	3200	2 P1, 2 P5	255.08	3200	20.00
3	950	5 P1, 2 P2, 1 P3	1402.93	950	67.37

* P1 ... P5 are the five 1D-DCT designs from SMASH.

Table 7.13: 2D-DCT implementations from SOS

7.7 Enhanced Design of JPEG Components

After the initial success with the JPEG system, we decided to

1. synthesize additional components of the JPEG system, and
2. improve the 1D-DCT design.

The design of the quantizer and enhanced 1D-DCT demonstrate the capability of SMASH in designing "real" designs. These VHDL descriptions included conditional-branches, loops, arrays, and on-chip constants.

7.7.1 Synthesis of the Quantizer

In the initial JPEG system design experiment, the quantizer parameters were manually estimated assuming only one possible implementation. The design space was not explored at all because the functional description of the quantizer (Figure 7.15) is so trivial that there is no obvious tradeoff in the functional hardware. The quantizer consists of a single division by a predetermined coefficient, therefore, there is no area-delay tradeoff in the functional hardware. However, there is a fair amount of I/O activity in the design as the whole stream of 8 elements has to be quantized and that motivated us to explore the tradeoffs possible due to I/O. We anticipated

a tradeoff in storage-related parameters and performed this experiment to demonstrate the existence of tradeoff curve due to high I/O activity even though there was no tradeoff possible in the functional hardware.

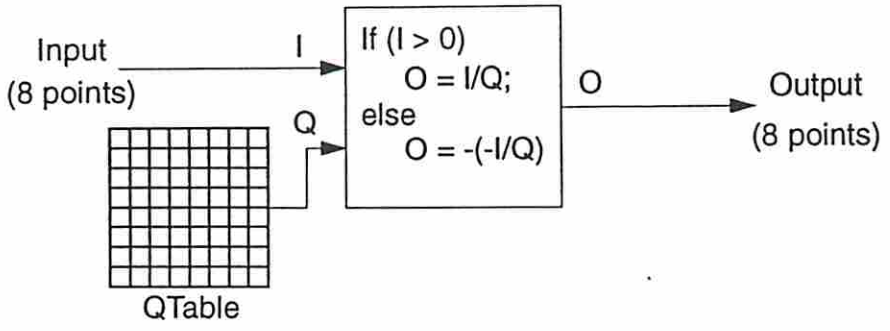


Figure 7.15: Quantization in JPEG Image Compression System

Observe that the description also includes an *if-then-else* condition to handle a negative input coefficient in case the divider is not designed to handle negative values.

The design parameters obtained from SMASH are summarized in Table 7.14. As anticipated, the two designs show a tradeoff between the storage-related cost and

Design no.	Input constraints		SMASH output					Loop latency (ns)
	Fun. area (10^6 μm^2)	$BW_{on\text{-}off}$ (words/ cycle)	R_{buf}	W_{buf}	Buffer size (words)	Fun. resources	Execution time (ns)	
1	3.0	1	1	1	2	<, /, Neg, +	720	60
2	3.0	2	1	1	3	<, /, Neg, +	600	60

Background memory size
= 8 (input stream) + 64 (QTable) + 8 (output stream)

Table 7.14: Quantizer Design Parameters Obtained from SMASH

execution time, though none in the functional hardware. They differ in bandwidth constraints, buffer size, and execution time. The functional hardware cost in both the designs is the same.

The first design is the cheaper and slower of the two designs. Our analysis of the design showed that I/O transfer is the bottleneck in this design and most of the time was being spent in transferring the I/O and the coefficients. There are 8 inputs, 8 quantization-coefficients, and 8 outputs which are accessed by the datapath. The BW_{on-off} allows 1 data transfer per cycle (30 ns) requiring at least 720 ns for data transfer. By overlapping the datapath execution with the data transfer, the design achieves the optimal execution time of 720 ns.

After realizing the bottleneck we increased the BW_{on-off} to 2 data transfers per cycle. The result was a faster design. However, the total execution time did not reduce proportionately because now the datapath execution dominated the total time. And of course, the speed up was gained at the expense of more bandwidth which implies higher cost.

7.7.2 Synthesis of 1D-DCT with Inner Loops

Our next step was to make the 8-point 1D-DCT, which is the most crucial component of the JPEG system, more efficient and realistic. To make it more efficient we decided to use a different description with fewer operations (in particular multiplications) [NK93]. And, to make the description more realistic, we (*i*) introduced arrays for the frame buffer instead of individual points, (*ii*) defined the on-chip constants instead of assuming them to be inputs, and (*iii*) included the loop definition inside the VHDL description instead of assuming it to be an outer-loop. By bringing the loop inside the description, SMASH could exploit the tradeoff associated with folding the loop.

The data-flow for the 8-point 1D-DCT (inside the loop body) is shown in Figure 7.16. The VHDL description of the whole design is included in Appendix B. The complete CDFG of the design including read and write nodes is shown in Figure 7.17.

The design parameters obtained from SMASH are summarized in Table 7.15. A brief description of these designs follows:

- *Design 1:* This design is the fastest design with execution time of 840 ns for the whole frame buffer. It is also the most expensive design in terms of total (functional and storage) cost. It is highly parallel and requires a large number

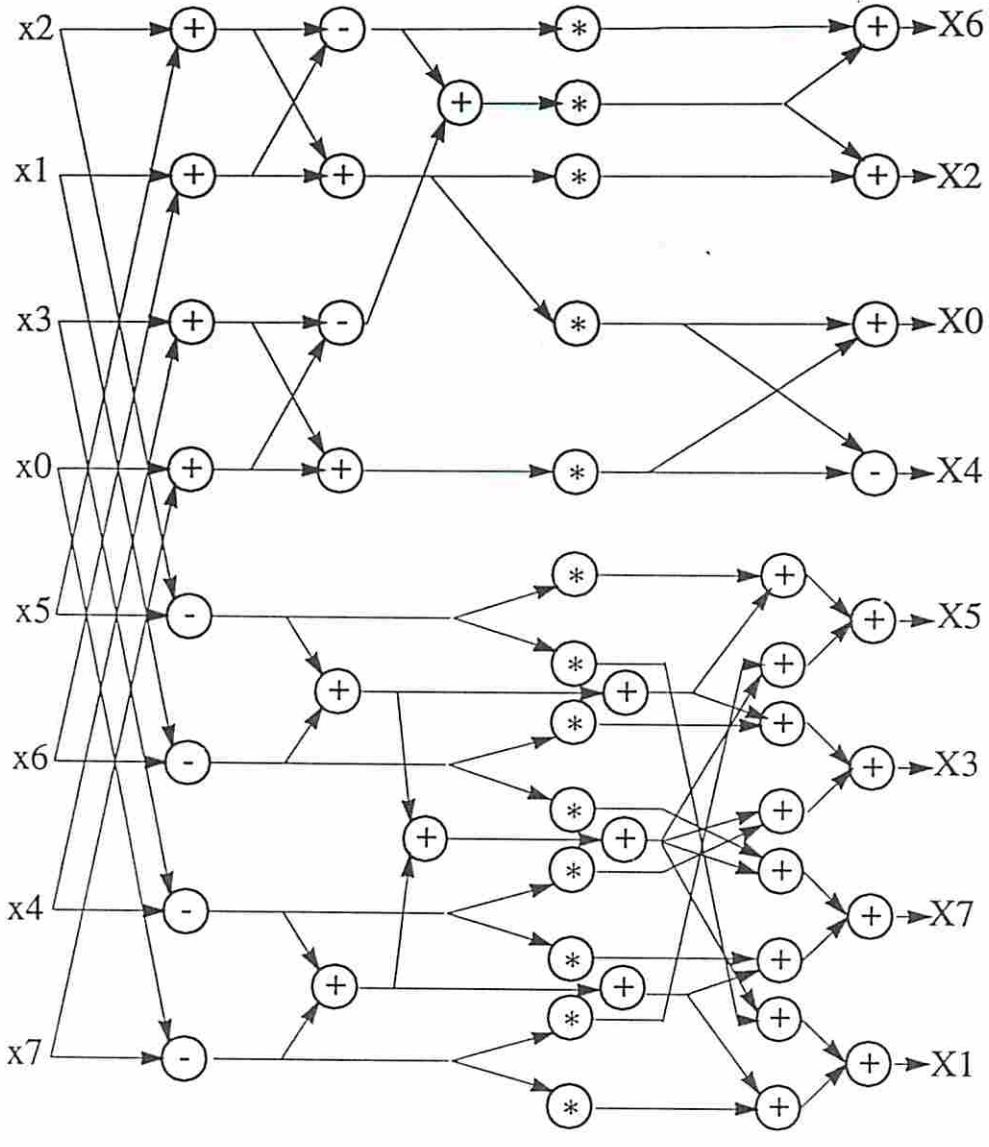


Figure 7.16: Data Flow for the 8-point 1D-DCT

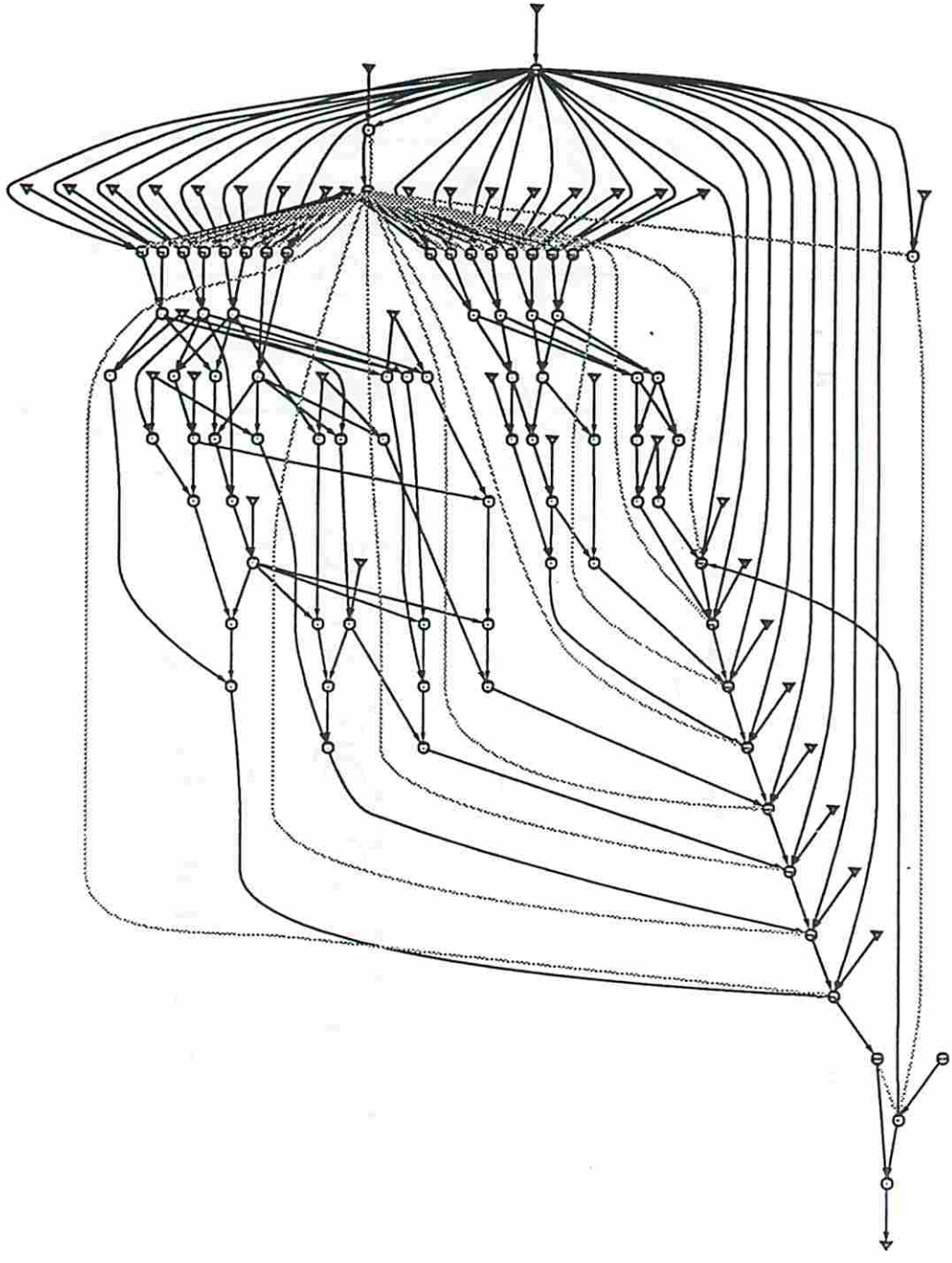


Figure 7.17: The Whole 8-point 1D-DCT CDFG

8 × 8 1D-DCT design parameters from SMASH

Design no.	Input constraints		SMASH output					Loop latency (ns)
	Fun. area (10^6 µm^2)	$BW_{on\text{-}off}$ (words/ cycle)	R_{buf}	W_{buf}	Buffer size (words)	Fun. resources	Execution time (ns)	
1	50	6	8	8	32	<, 16 *,5 -, 14 +	840	60
2	50	4	8	8	28	<, 16 *, 7 -, 14 +	960	60
3	35	4	6	6	20	<, 12 *, 4 -, 12 +	1080	90
4	25	4	4	3	7	<, 8 *, 3 -, 7 +	1170	120
5	20	4	4	3	7	<, 7 *,3 -, 10 +	1590	180
6	15	2	3	3	7	<, 5 *, 4 -, 8 +	2190	240

Background memory size = 64 (input frame) + 64 (output frame)

Table 7.15: Enhanced 1D-DCT Design using SMASH

of funtional modules. Furthermore, the buffer size is large because, in order to meet the high performance of the datapath, the inputs were required to be prefetched and stored in the buffers.

- *Design 2:* This design is a bit slower and cheaper than design 1. Though, the functional cost of this design is higher than design 1, it is a cheaper design because of the lower BW_{on-off} (which translates into number of pins on the chip). An analysis of the design showed that the extra delay was spent on transferring the data on and off chip as BW_{on-off} is lower. Buffer size continued to be high in this case also, due to prefetching of inputs to meet the high datapath performance.

- *Design 3:* This design is substantially cheaper and slower than the above two designs. Even the buffer size is smaller in this design; this is true because the longer execution time allowed more flexibility in scheduling the I/O transfers. Later, during the data-transfer scheduling SMASH could scatter the transfers in such a way that the memory locations could be shared.

- *Design 4:* This design was allowed a limited functional area and that resulted in a lower performance. Notice that in this case prefetching of inputs is not required. the buffers are there just to latch the inputs and outputs in each

step. This is because the long execution time and sufficient BW_{on-off} allowed the inputs to be fetched just in time and outputs to be transferred back immediately. Therefore no extra storage is required.

- *Design 5:* This design is quite close to design 4 in terms of cost but very slow in execution. Our analysis showed that it was primarily because of the 1D-DCT description itself. The data dependencies in the 8-point 1D-DCT CDFG is such that for up to 8 multipliers, the execution schedule is very regular; however, when there are less than 8 multipliers, some of the operations are delayed significantly.

- *Design 6:* This is the cheapest and the slowest design with a lot of resource sharing. The storage cost is also low because the longer execution time allowed a lower bandwidth (providing any higher bandwidth is of no use as the datapath requires data at a slower rate) and a smaller buffer.

7.7.3 Discussion

The design of the quantizer and the 1D-DCT with inner loop showed demonstrated the existence of cost-performance tradeoff in datapath as well as the storage structure. The quantizer example showed the tradeoffs in the storage structure. The 1D-DCT design showed how designs with arrays, on-chip constants and loops can be handled by SMASH. The design contained an inner loop which was *folded* to achieve higher performance.

7.8 Summary

In this chapter we presented the design experiments performed using SMASH along with other USC and commercial tools. We first presented two layout experiments which were conducted prior to development of SMASH.R The remaining chapter described three set of experiments performed using SMASH.

The first set of experiments was to synthesize some representative benchmark examples from the high-level synthesis workshop. These designs showed the tradeoffs in the storage structure in addition to the tradeoffs in the datapath. They also

showed that the storage-related constraints must be considered during datapath synthesis to achieve a cost-effective storage structure. Separating the synthesis of datapath from the synthesis of storage structure results in an inefficient design.

The second experiment involved the design of the JPEG Still Image Compression System. SMASH was used in the experiment to implement the 1D-DCT module. This experiment showed how high-level synthesis tools like SMASH can be used in designing the systems. It also showed that SMASH like tools can be used in exploring the design space quickly by performing 'what-if' analysis during the design process. The designer can choose to change the system-specification and study the impact of such changes on the final implementations immediately.

The third set of experiments consisted of the design of the quantizer used in the JPEG system and an improved design of the 1D-DCT. These specifications contained on-chip constants, arrays and loops. SMASH generated several implementations of these designs with varying cost and performance.

Chapter 8

Conclusion and Future Research

8.1 Introduction

In this thesis we have looked at the techniques that support automatic synthesis of memory-intensive application-specific systems. Based on these techniques, a software tool called SMASH (Synthesis of Memory-intensive Application-Specific Hardware) has been developed. The primary objective of this research was to broaden the scope of high-level synthesis systems by considering memory related issues during the synthesis process.

Several memory-intensive application-specific systems were studied to understand various issues in design of these systems. The following observations were made during the preliminary research:

- Synthesis tools must consider design of storage architectures and other system modules in order to be accepted by the industry.
- The storage architecture is closely connected to the datapath and isolating its synthesis from the datapath synthesis may not result in an efficient solution. Datapath synthesis procedures themselves must take into account the design of the memory hierarchy, and the design of the datapaths and memory hierarchies must be coordinated.
- Every application requires a unique strategy to design the most efficient storage architecture in terms of both cost and performance. However, a more general approach must be developed which can be applied to automate the design process while producing efficient and correct designs.

- "Real" design specifications contain a wide variety of features like mixed control and data flow, I/O timing constraints, on-chip constants, arrays, conditional branches, and loops. These features should be handled by the synthesis system.

These observations motivated us to develop synthesis techniques which (*i*) combine the datapath synthesis with memory hierarchy design, (*ii*) handle the above mentioned features in "real" design specifications, and (*iii*) exploit all the advantages design automation has over manual designs like faster design time, less errors, and exploration of a larger design space.

In this chapter we have summarized our contributions. Next, we have outlined the areas for future research.

8.2 Contributions

In summary, our contributions are as follows:

8.2.1 Development of a High-Level Synthesis System

The main contribution of this research is SMASH this tool set, *given*

- a behavioral VHDL description of a memory-intensive application-specific system,
- a module library consisting of (i) functional modules, and (ii) storage modules,
- area-performance constraints,
- a clock cycle, which is the duration of each control step in the datapath,
- input/output timing constraints imposed by the external world, and
- memory bandwidth constraints,

produces a target system with

- a datapath consisting of operators and operation schedule,

- size and port configuration for on-chip foreground memory to store inputs, outputs and intermediate variables,
- data-transfer schedule between the datapath and on-chip memory,
- size and port configuration off-chip (or on-chip) background memory for bulk storage, and
- data-transfer schedule between the foreground and background memory.

The development of this tool set required identification of the problem, the issues and the design parameters involved. Next, we developed the required techniques to achieve our goals. The main contributions in these areas are described below.

8.2.2 Identification of Design Parameters

Based on the study of various memory-intensive examples, the design parameters relevant to each memory structure were identified and the following architecture with two levels of memory hierarchy was proposed to solve the problem:

1. *on-chip foreground memory*: which consists of I/O buffers to store inputs and outputs, and datapath memory to store intermediate variables, and
2. *off-chip background memory*: for bulk storage.

For the I/O buffers, the relevant parameters which must be optimized by the synthesis software are

1. the number of read ports and write ports accessible to the datapath, which is the maximum number of inputs and outputs accessed by the datapath in any given control step, and
2. total buffer size, which is determined by the maximum number of inputs and outputs stored in the buffers in any given control step.

The parameters which determine the datapath memory are

1. the number of intermediate variables, and

2. the lifetimes of these variables.

Datapath memory synthesis has already been performed by other researchers [BMB+88, Che91, Sto89] and was not addressed in this thesis.

For the off-chip memory, the relevant parameters that must be optimized by the synthesis software are

1. the number of read/write ports, and

2. size, which is expected to be large compared to the size of on-chip storage size and is implicitly determined by our software as a side-effect of the data transfer scheduling.

Our next contribution was to develop techniques to determine these parameters for a given behavioral description.

8.2.3 Combined Datapath Scheduling with I/O Accesses

We developed techniques to combine datapath scheduling with I/O access scheduling. We considered I/O accesses from/to the I/O buffers as read/write operations and scheduled them concurrently with the datapath operations. Besides combining the datapath scheduling with the I/O access scheduling, our datapath scheduling techniques consider

- the storage architecture parameters so that the schedule is guaranteed to satisfy the constraints during the storage synthesis,

- the tradeoffs in the storage architecture so that the schedule does not require alterations during storage design, and

- storage and functional cost estimations to evaluate the impact of high-level decisions on the final design so that potentially inferior designs are discarded early in the design process.

8.2.4 Storage Tradeoffs

We identified the following tradeoffs which could be made in the storage structures:

1. storage size *vs.* number of execution cycles,
2. number of ports *vs.* number of execution cycles, and
3. number of ports *vs.* storage size.

8.2.5 Storage Cost Estimations

We developed techniques to estimate the storage cost. The storage cost estimation consists of the following three steps:

1. determining a lower bound on the number of read and write ports on the buffers,
2. determining a lower bound on the total size of all the buffers, and
3. implementing these requirements on lowest-cost storage modules from the library.

We also presented the required theoretical basis to prove these bounds.

8.2.6 Upper Bounds on Design Parameters

We developed theories to determine the upper bounds for

1. the number of read and write ports on the buffers,
2. the total size of all the buffers, and
3. the number of functional modules in the design.

8.2.7 Storage Synthesis

We proposed a two step design process for storage synthesis. These steps are

1. data transfer scheduling, and
2. storage module allocation.

We developed techniques to perform the data transfer scheduling. The second step, the module allocation, requires further research and will be completed in the future.

8.2.8 Experiments

We conducted a number of design experiments to test and verify our techniques and software. The experiments included

1. designs from high-level synthesis workshop benchmarks,
2. rapid prototyping of a JPEG still image compression system, and
3. enhanced design of two of the components (1D-DCT and quantizer) of the JPEG still image compression system.

The following aspects of our research were successfully demonstrated though these experiments:

- the capabilities of SMASH in combining the datapath scheduling with I/O access from the buffers,
- improvement in datapath schedules when storage-related constraints were also considered,
- existence of a cost-performance tradeoff in the storage architecture (cost being a function of R/W ports, storage size, and BW_{on-off}),
- the capabilities of the software in terms of handling "real" designs which include arrays, loops and conditional branches in the input behavioral specification, and
- the role of SMASH in a system-level design environment like USC.

8.3 Future Directions

The following issues were not addressed in this research. They must be addressed in the future in order to make the synthesis system versatile and thorough.

- high-level memory management,
- storage module allocation,

- improvement in datapath memory synthesis, and
- interfacing SMASH with DPSYN.

8.3.1 High-Level Memory Management

It has been well recognized that transformations on algorithm description are crucial in obtaining efficient implementations. The requirement of these transformations arises because user-specified algorithms may not be efficient. For instance, we do not need to keep independent operations together (as in loop boundaries) just because of the algorithmic construct. Studies on memory organization for multi-dimensional signals have shown that loop transformations can have effects on final implementation costs [FBS+93, LvMvdW+91]. These transformations have been found to be very effective in achieving two goals: parallelism and efficient use of memory hierarchy.

In case of arrays, transferring huge arrays into on-chip buffers could be very expensive. In such cases, memory accesses can be greatly optimized by considering data locality. The consideration of data locality makes it more important to apply loop transformations in a systematic manner. Another transformation that can be used is decomposition of arrays into smaller arrays so that only that part of the array which is useful for the processing can be transferred. Unfortunately, decomposing each array into individual data points will increase the complexity of the problem as now we will have to deal with more arrays. Therefore, the decomposition must be done in such a way that there are not excessive arrays to handle and each array is small enough to avoid unnecessary data transfers.

Similar issues are being addressed by several researchers working on advanced compilers [WL91, Lov77, PW86]. These ideas can be extended to our applications.

8.3.2 Storage Module Allocation

As described earlier, in our research we have not addressed the issue of allocating a physical memory location to each data value. After the data transfers are scheduled and the data requirements in each control step are known, we need to allocate each value to an appropriate physical module. The objective of this step is to distribute

these data values among different modules such that there is no access clash and also access time and storage cost (both storage area and interconnection area) are minimized.

This step consists of the following basic tasks:

1. selecting the storage modules,

2. distributing the data values among the selected modules, and

3. completing the interconnection network and addressing hardware.

Notice that during the data transfer scheduling it was ensured that the feasibility of a complete allocation was guaranteed. Therefore, the choice of the module type will only depend on the following requirements:

1. the number of read/write ports on the modules, and

2. the size of the modules.

Next, the data values can be assigned to the selected modules. This is a rather difficult task as different module types have different numbers of ports and sizes. Distributing the data values among such heterogeneous modules has yet to be researched. Existing approaches do not consider distribution of data among heterogeneous modules at the same time. Another issue is allocation of arrays as a single entity rather than expanding them to individual elements. PHIDEO assigns data streams to multiport memory modules by dividing the overall problem into subproblems which are then solved using ILP formulation [LvMVvdW93]. Though, this approach is based on PHIDEO's *data stream* model, it can be modified to allocate arrays to background memory in our case.

Furthermore, the interconnect cost also must be considered during the storage construction. To reduce the interconnection overhead, we prefer to keep the data required by a functional module in the storage module to which that functional module is already connected. We can start distributing the data values from control step one. To perform a heuristic module allocation we can first choose a value then compute the cost of assigning it to a particular module. The cost should include the interconnection cost. Depending on the cost we can assign it to the cheapest

module. We choose another value from the set of values remaining in that control step and assign it to an appropriate module, and so on.

Overall, this problem needs to be researched extensively.

8.3.3 Improvement in Datapath Memory Synthesis

The datapath memory synthesis problem has been extensively studied by Balakrishnan et al. [BMBL87, BMB+88]. However, their approach considers the same type of modules. Also, they optimize the interconnection separately. A more efficient solution could be obtained if heterogeneous module types (like register files with varying number of ports, registers, and on-chip RAMs with varying number of ports) were allowed. Also, the interconnection cost could be considered while merging the registers into larger modules. Heterogeneous modules allow us to avoid using expensive modules when they are not required. Furthermore, registers should be merged selectively without insisting on merging all the registers into a large module. Registers which are used most frequently should not be merged as this might increase the access-clash with other variables stored in that module.

A possible approach to solve this problem could be to first form a no-conflict graph for all the data values. Two nodes of this graph would connected if they could be put in the same module and could be accessed in the same time step without causing an access clash. Then, based on the access pattern, various compatible sets could be formed. A *compatible set* is the set of data values which can be allocated to the same module without any access clash. All the elements of the same compatible set could be allocated to the same storage module (register file or RAM). In order to minimize the interconnection overhead, the storage modules must be assigned based on their connectivity with the functional modules. In other words, first clusters of data values which can be put together should be formed then depending on the functional modules they are interacting with, the storage modules could be assigned (allocate the data values to register files or RAMs).

8.3.4 Address and Control Generation

Synthesis of the address and control signal generator is another issue which must be considered in the future. In case of non-addressable memory modules like registers and register files, the control signal generation is the issue; whereas, in case of addressable memory modules like RAMs address generation is the issue. For addressable memories like on-chip RAMs, the address to access a memory location can be generated using a variety of algorithms and hardware. The cost of the decoding logic can be very significant [Bur90].

Grant et al. studied the address generation problem for the case when the block of memory is accessed repeatedly [GDF89]. Comprehensive address generation problem is yet to be addressed. Since, address generation is directly related to the module allocation, we recommend that this issue be considered during the module allocation step.

8.3.5 Interfacing SMASH with DPSYN

DPSYN is an RTL synthesizer from COMPASS Design Automation Inc. It accepts a scheduled data flow graph in the form of a finite state machine and generates the layout of the design. DPSYN was acquired quite recently and is under evaluation. The required interfaces between SMASH and DPSYN need to be developed in the near future.

Appendix A

MABAL to SSCNET Netlist Translator

A netlist translator was written to complete the pathway from specification to layout with automation of all major design steps. The translator translates the ADAM RTL output netlist to the Cascade Design Automation ChipCrafter (now Epoch) bit-level netlist. The program has the following features:

- It expands the RTL netlist to a bit-level netlist for a user-specified bitwidth.
- It constructs some 'complex' modules from existing basic modules. For example, a subtracter is constructed by using adder and inverter cells. Similarly, a shifter is implemented by just shifting the bit connections by the required number of bits.
- It integrates the controller and the data path by connecting the control signals to the data path modules appropriately.
- It can translate testable designs having BILBO or SCAN registers.
  - For BILBO methodology, it can create the BILBO registers for the given polynomial.
  - For SCAN methodology, it can connect all the scan registers in a scan chain.
- It is modular. Any new type of module can be included simply by writing a function defining the attributes of the new module.

This work helped us in performing a number of experiments:

- More than 50 layouts have been generated until now to study the effects of high-level decisions on the final layouts.
- An arithmetic Fourier transform chip was designed from the VHDL description to physical layout within 48 hours.
- Some AR filter designs were made testable to study the impact of adding testability on the area and the performance of the final designs.
- A 1D-DCT and a 2D-DCT chips were designed as a part of a JPEG-image compression system design.

Appendix B

VHDL Descriptions

In this appendix the VHDL descriptions of the examples synthesized in this thesis are included. These examples are:

1. a second-order differential equation solver,
2. an AR filter element,
3. an Elliptic wave filter element,
4. a 1D-DCT with inner loop, and
5. a quantizer.

B.1 VHDL description of 2nd Order Differential Equantion Solver

```
-- diffeq.vhdl
-- author       : chih-tung chen
-- source       : diffeq.v in high level synthesis workshop.
-- description  : This is a VHDL behavioral description of
--                a differential equation.
--                It is modifed from diffeq.v in a way that
--                the internal loop of the original description
--                is transformed using the mitchell's method in
--                order to have a loopless description.
-- assumption   :
--      1. there are external feedbacks as follows:
--         oxport -> ixport, oyport -> iyport,
--         and ouport -> iuport.
--      2. control signals such reset, ready, nxt and over
--         should be added externally.
--      3. It should be implemented as a non-piplined design
--         due to the unfixed loop in the original algorithm.

package TYPES is
  type SixteenBitVector is array(15 downto 0) of Bit;
end TYPES;
use work.TYPES.all;
package OPR16 is
  function "+"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function "-"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function "*"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function less(opn1,opn2:SixteenBitVector) return Boolean;
end OPR16;
use work.TYPES.all;
use work.OPR16.all;
```

```
entity diffeq is
  port(aport, dxport : in SixteenBitVector;
       ixport, iyport, iuport : in SixteenBitVector;
       oxport, oyport, ouport : out SixteenBitVector);
end diffeq;
architecture behaviour of diffeq is
begin process
variable a,x,y,dx,u,u1,u2,u3,u4,u5,u6,y1:SixteenBitVector;
constant three: SixteenBitVector := X"0003";
constant five : SixteenBitVector := X"0005";
begin
  x := ixport;
  y := iyport;
  u := iuport;
  dx := dxport;
  a := aport;
  u1 := u * dx;
  u2 := three * x;      -- modified by Pravil
  u3 := three * y;
  y1 := u1;             -- original y1 := u * dx;
  u4 := u1 * u2;
  u5 := dx * u3;
  if less(x, a) then
   x := x + dx;
   y := y + y1;
   u6 := u - u4;
   u := u6 - u5;
  end if;
  oxport <= x;
  oyport <= y;
  ouport <= u;
end process;
end behaviour;
```

B.2 VHDL description of an AR Filter Element

```
-- arf.vhdl
-- author       : chih-tung chen
-- date         : 3/18/92
-- source       : originated from Jagan's description.
-- description  : an AR filter element in VHDL behavior description
package TYPES is
  type SixteenBitVector is array(15 downto 0) of Bit;
end TYPES;
use work.TYPES.all;
package OPR16 is
  function "+"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
  function "*"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
end OPR16;
use work.TYPES.all;
use work.OPR16.all;
entity arf is
  port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,
       a13,a14,a15,a16,a17,a18,a19,a20,a21,a22,
       a23,a24,a25,a26 :in SixteenBitVector;
       out1,out2:out SixteenBitVector);
end arf;
architecture behaviour of arf is
begin process
  variable e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,
   e13,e14,e15,e16,e17,e18,e19,e20,e21,e22,e23,e24,
   e25,e26,e27,e28,e29,e30: SixteenBitVector;
  begin
    e1:=a1 * a2;
    e2:=a3 * a4;
    e3:=a5 * a6;
    e4:=a7 * a8;
```

```
    e5:=a9 * a10;
    e6:=a11 * a12;
    e7:=a13 * a14;
    e8:=a15 * a16;
    e9:=e1 + e2;
    e10:=e3 + e4;
    e11:=e5 + e6;
    e12:=e7 + e8;
    e13:=e11 + a17;
    e14:=e13;
    e15:=a18 + e12;
    e16:=e15;
    e17:=a19 * e15;
    e18:=e13 * a20;
    e19:=e14 * a21;
    e20:=e16 * a22;
    e21:=e17 + e18;
    e22:=e21;
    e23:=e19 + e20;
    e24:=e23;
    e25:=a23 * e23;
    e26:=e21 * a24;
    e27:=a25 * e22;
    e28:=a26 * e24;
    e29:=e25 + e26;
    e30:=e27 + e28;
    out1<=e9 + e29;
    out2<=e10 + e30;
  end process;
end Behaviour;
```

B.3 VHDL description of an Elliptic Wave Filter Element

```
-- ellipf.vhdl
-- date         : 9/15/93
-- source       : ellipf.vhdl in HLS92 benchmarks.
-- assumption   :
--      1. control signals such reset, ready, nxt and over
--         should be added externally.
--      2. only the internal body of the while loop is considered.

package TYPES is
  type SixteenBitVector is array(15 downto 0) of Bit;
end TYPES;
use work.TYPES.all;
package OPR16 is
  function "+"(opn1,opn2:SixteenBitVector) return SixteenBitVector;
end OPR16;
use work.TYPES.all;
use work.OPR16.all;
entity ellipf is
    port ( inp : in SixteenBitVector;
           outp : out SixteenBitVector;
           sv2, sv13, sv18, sv26, sv33, sv38, sv39 :
                 in SixteenBitVector;
           sv2_o, sv13_o, sv18_o, sv26_o, sv33_o, sv38_o, sv39_o :
                 out SixteenBitVector);
end ellipf;
architecture ellipf of ellipf is
begin process
    variable n1, n2, n3, n4, n5, n6, n7 : SixteenBitVector;
    variable n8, n9, n10, n11, n12, n13 : SixteenBitVector;
    variable n14, n15, n16, n17, n18, n19 : SixteenBitVector;
```

```
    variable n20, n21, n22, n23, n24, n25 : SixteenBitVector;
    variable n26, n27, n28, n29 : SixteenBitVector;
begin
    n1 := inp + sv2;
    n2 := sv33 + sv39;
    n3 := n1 + sv13;
    n4 := n3 + sv26;
    n5 := n4 + n2;
    n6 := n5 ;
    n7 := n5 ;
    n8 := n3 + n6;
    n9 := n7 + n2;
    n10 := n3 + n8;
    n11 := n8 + n5;
    n12 := n2 + n9;
    n13 := n10 ;
    n14 := n12 ;
    n15 := n1 + n13;
    n16 := n14 + sv39;
    n17 := n1 + n15;
    n18 := n15 + n8;
    n19 := n9 + n16;
    n20 := n16 + sv39;
    n21 := n17 ;
    n22 := n18 + sv18;
    n23 := sv38 + n19;
    n24 := n20 ;
    n25 := inp + n21;
    n26 := n22 ;
    n27 := n23 ;
    n28 := n26 + sv18;
    n29 := n27 + sv38;
    sv2_o <= n25 + n15;
```

```
    sv13_o <= n17 + n28;
    sv18_o <= n28;
    sv26_o <= n9 + n11;
    sv38_o <= n29;
    sv33_o <= n19 + n29;
    sv39_o <= n16 + n24;
    outp <= n24;
 end process;
end ellipf;
```

B.4 VHDL description of 8-point 1D-DCT

```
-- dct.vhdl
  --
  -- source      : IEEE Transaction on CAD August 1993 Vol 12
  --               Number 8 page 1120.
  -- description: an 8X8 16-bit DCT VHDL behavior. Consists of
  --               8 8-point dct's in a loop.
  -- date        : Dec. 23rd 1993.
  -- written by : Pravil Gupta

  package TYPES is
    type Frame is array(0 to 7, 0 to 7) of integer;
  end TYPES;

  -- Entity declaration doesn't determine
  -- the number of ports on the chips.
  use work.TYPES.all;
  entity dct is
    port(InFrame :in Frame;
         OutFrame :out Frame);
  end dct;

  architecture behavior of dct is
  use work.TYPES.all;

  begin process

  variable t1,t2,t3,t4,
     m1,m2,m3,m4,m5,m6,m7,m8,
     n1,n2,n3,n4: integer;
  variable  I: integer;
  constant c1,c2,c3,c4,c5,c6,c9,c12,c14,c15,c16 :integer := 0;
```

```
-- each constant must be assigned appropriately.

begin
        I := 0;
        while I < 8 loop
                t1 := InFrame(0,I)+InFrame(7,I);
                t2 := InFrame(3,I)+InFrame(4,I);
                t3 := InFrame(1,I)+InFrame(6,I);
                t4 := InFrame(2,I)+InFrame(5,I);

                m1 := t1+t2;
                m2 := t3+t4;
                m3 := t1-t2;
                m4 := t3-t4;

                 -- d = c12 = c13
                OutFrame(0,I) <= c12*(m1+m2);
                OutFrame(4,I) <= c12*(m1-m2);

                -- c14+c15 = b & c15 = f
                -- c15 = f & c15+c16 = -b
                n1 := c15*(m3+m4);
                OutFrame(2,I) <= c14*m3 + n1;
                OutFrame(6,I) <= c16*m4 + n1;

                m5 := InFrame(0,I)-InFrame(7,I);
                m6 := InFrame(1,I)-InFrame(6,I);
                m7 := InFrame(2,I)-InFrame(5,I);
                m8 := InFrame(3,I)-InFrame(4,I);

                -- c6 = c, c3 = g-c, c9 = -a-c
                n2 := c6*(m5+m6+m7+m8);
                n3 := c3*(m5+m8);
```

```
                n4 := c9*(m6+m7);

                -- c2 = c10, c1 = c8, c4 = c11, c5 = c7
                OutFrame(1,I) <= c1*m5 + n3 + n2 + c2*m7;
                OutFrame(3,I) <= c5*m8 + n2 + c1*m6 + n4;
                OutFrame(5,I) <= c2*m5 + n2 + n4 - c4*m7;
                OutFrame(7,I) <= n3 + c4*m8 + n2 + c5*m6;
                I := I + 1;
        end loop;

  end process;
end behavior;
```

B.5 VHDL Description of a Quantizer

```
-- dct.vhdl
--
-- source      :
-- description:
-- date        : March 12th 1994
-- written by  : Pravil Gupta

package TYPES is
  type Row is array(0 to 7) of integer;
  type Frame is array(0 to 7, 0 to 7) of integer;
end TYPES;

-- Entity declaration doesn't determine the
-- number of ports on the chips.
use work.TYPES.all;
entity quant is
  port(InRow  :in Row;
       QTable :in Frame;
       OutRow :out Row);
end quant;

architecture behavior of quant is
use work.TYPES.all;

begin process

  variable temp : integer;
  variable I : integer;
  constant J : integer := 0;

  begin
```

```
    I := 0;
    while ( I <  8 ) loop

      temp := InRow(I);

      -- divide by QTable[J,I], ensuring proper rounding

      if (temp < 0)
      then
        temp := -temp;
        temp := temp/QTable(J,I);
        temp := -temp;
      else
        temp := temp/QTable(J,I);
      end if;

      OutRow(I) <= temp;
      I := I + 1;
    end loop;

  end process;
end behavior;
```

Reference List

[AC91] I. Ahmad and C. Y. Roger Chen. Post-Processor For Data Path Synthesis Using Multiport Memories. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 276–279, 1991.

[BCM+88] R.K. Brayton, R. Camposano, G. De Micheli, R.H.J.M. Otten, and J. van Eijndhoven. *Silicon Compilation,* ed. D.D. Gajski, chapter "The Yorktown Silicon Compiler System", pages 204–310. Addison-Wesley, 1988.

[BMB+88] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, J.G. Linders, and J.C. Majithia. Allocation of Multiport Memories in Datapath Synthesis. *IEEE Trans. on Computer Aided Design*, 7(4):536–540, April 1988.

[BMBL87] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, and J.G. Linders. Allocation of Multiport Memories in Datapath Synthesis. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 266–269, 1987.

[Bur90] W. P. Burleson. Memory Design for Bit-level VLSI Architectures. In *Proc. of the IEEE Int'l Symp. Circuits and Systems*, pages 2308–2311, 1990.

[Cas93] Cascade Design Automation Corporation, 3075 112th Avenue NE, Bellevue, WA 98004. *Epoch Designer's Handbook*, 1993.

[CGDBP94] C. T. Chen, P. Gupta, J. C. DeSouza-Batista, and A. C. Parker. Rapid Prototyping of a JPEG Image Compression System using Synthesis Tools. In *IEEE Data Compression Conf.*, March 1994.

[Che91] C. H. Chen. Allocation of Multiport Memory with Ports of Different Types in Register Transfer Level Synthesis. In *Proc. of the Int'l Conf. on Computer Design*, pages 418–421, 1991.

[CP91] C. T. Chen and A. C. Parker. VHDL2DDS: A VHDL Language to DDS Data Structure Translator. Technical Report CEng 91–21, Department of EE-Systems, University of Southern California, July 1991.

[CS90] C. H. Chen and G. E. Sobelman. Singleport/Multiport Memory Synthesis in Data Path Design. In *Proc. of the IEEE Int'l Symp. Circuits and Systems*, pages 1110–1112, 1990.

[CS93] C.F. Chang and B. J. Sheu. A Multi-Chip Module Design for Portable Video Compression Systems. In *IEEE Multi-Chip Module Conf.*, pages 39–44, 1993.

[DPST81] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas. A Design Methodology and Computer Aids for Digital VLSI Systems. *IEEE Transactions on Circuits and Systems*, CAS-28:634–645, July 1981.

[FBS+93] Frank Franssen, Florin Balasa, M. Swaaij, F. Catthoor, and H. De Man. Modeling Multidimenstional Data and Control Flow. *IEEE Tran. on VLSI Systems*, 1(3):319–327, September 1993.

[FLS+92] H. Fujiwara, M.L. Liou, M.T. Sun, K.M. Yang, M.M. Maruyama, K. Shomura, and K. Ohyama. An All-ASIC Implementation of a Low Bit-Rate Video Codec. *IEEE Trans. on Circuits and Systems for Video Technology*, 2(2):123–134, June 1992.

[GBK85] E.F. Girczyc, R.J. Buhr, and J. Knight. Applicability of a Subset of ADA as an Algorithmic Hardware Description Language for Graph Based Hardware Compilation. *IEEE Trans. on Computer Aided Design*, CAD-4(2), April 1985.

[GCDBP94] P. Gupta, C. T. Chen, J. C. DeSouza-Batista, and A. C. Parker. Experience with Image Compression Chip Design using Unified System Construction Tools. In *Proc. of the 31st Design Automation Conf.*, June 1994.

[GD90] D.M. Grant and P.B. Denyer. Memory, Control and Communication Synthesis for Scheduled Algorithms. In *Proc. of the 27th Design Automation Conf.*, pages 162–167, June 1990.

[GDF89] D.M. Grant, P.B. Denyer, and I. Finlay. Synthesis of Address Generators. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 116–118, 1989.

[GK84] E. Girczyc and J. Knight. An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling. In *Proc. of the Int'l Conf. on Computer Design*, pages 726–729, October 1984.

[GKP85] J. Granacki, D. Knapp, and A. C. Parker. The ADAM Design Automation System: Overview, Planner and Natural Language Interface. In *Proc. of the 22nd Design Automation Conf.*, pages 727–730, June 1985.

[GP94] P. Gupta and A. C. Parker. SMASH: A Program for Scheduling Memory-Intensive Application Specific Hardware. In *7th Int'l Symposium on High-Level Synthesis*, May 1994.

[HHL90] C. T. Huang, Y.C. Hsu, and Y.L. Lin. Optimum and Heuristic Data Scheduling under Resource Constraints. In *Proc. of the 27th Design Automation Conf.*, pages 65–70, June 1990.

[HMFK90] S. Hirofumi, N. Matsumoto, K. Fujimori, and S. Kato. A Flexible Multi-Port RAM Compiler for Datapath. In *Proc. of the IEEE Custom Integrated Circuits Conf.*, pages 16.5.1–16.5.4, May 1990.

[HP90] A. Hemani and A. Postula. A Neural Net Based Self Organising Scheduling Algorithm. In *Proc. of the European Design Automation Conf.*, pages 136–139, March 1990.

[JKMP89] R. Jain, K. Küçükçakar, M. J. Mlinar, and A. C. Parker. Experience with the ADAM Synthesis System. In *Proc. of the 26th Design Automation Conf.*, pages 56–61, June 1989.

[JMP88] R. Jain, M. J. Mlinar, and A. C. Parker. Area-Time Model for Synthesis of Non-Pipelined Designs. In *Proc. of the Int'l Conf. on Computer Aided Design*, November 1988.

[JPO93] A. Jerraya, I. Park, and K. O'Brien. AMICAL: An Interactive High-Level Synthesis Environment. In *EDAC 93*, February 1993.

[JPP87] R. Jain, A. C. Parker, and N. Park. Predicting Area-Time Tradeoffs for Pipelined Design. In *Proc. of the 24th Design Automation Conf.*, pages 35–41. IEEE and ACM, July 1987.

[KP83] D. Knapp and A. C. Parker. A Data Structure for VLSI Synthesis and Verification. Technical report, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, October 1983.

[KP85] D. Knapp and A. C. Parker. A Unified Represention for Design Information. In *Proceedings of the IFIP Conf. on Hardware Description Languages*, August 1985.

[KP90] K. Küçükçakar and A. C. Parker. Data Path Tradeoffs using MABAL. *Proc. of the 27th Design Automation Conf.*, June 1990.

[KT83] T. J. Kowalski and D. E. Thomas. The VLSI Design Automation Assistant: Prototype System. In *Proceedings of the 20th Design Automation Conf.*, 1983.

[Kuc91] K. Kucukcakar. *System-Level Synthesis Techniques with Emphasis on Partitioning and Design Planning.* PhD thesis, University of Southern California, September 1991.

[LHL89] J.H. Lee, Y. C. Hsu, and Y. L. Lin. A New Integer Linear Programming Formulation for the Scheduling Problem. In *Digest of*

Technical Papers of the Int. Conf. of Computer Aided Design, pages 20–23, November 1989.

[Lov77] D. B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the Association for Computing Machinery*, 24(1):121–145, January 1977.

[LT89] E. Lagnese and D. E. Thomas. Architectural Partitioning for System Level Design. In *Proc. of the 26th Design Automation Conf.*, June 1989.

[LvMvdW+91] P. E. R. Lippens, J. L. van Meerbergen, A. van der Werf, W. F. J. Verhaegh, and B. T. McSweeney. Memory Synthesis for High Speed DSP Applications. In *Proc. of the IEEE Custom Integrated Circuits Conf.*, pages 11.7.1–11.7.4, May 1991.

[LvMVvdW93] P. E. R. Lippens, J. L. van Meerbergen, W. F. J. Verhaegh, and A. van der Werf. Allocation of Multiport Memories for Hierarchical Data Streams. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 728–735, 1993.

[Mar79] P. Marwedel. The MIMOLA Design System: Detailed Description of the Software System. In *Proc. of the 16th Design Automation Conf.*, pages 59–62, 1979.

[MMC88] A. C. Parker M. McFarland and R. Camposano. Tutorial on High-Level Synthesis. *Proc. of the 25th Design Automation Conf.*, Jul 1988.

[NK93] J.A. Nestor and G. Krishnamoorthy. SALSA: A New Approach to Scheduling with Timing Constraints. *IEEE Trans. on on Computer-Aided Design*, 12(8):1107–1122, August 1993.

[NP77] A. Nagle and A. Parker. Hardware/Software Tradeoffs in a Variable Word Width, Variable Queue Length Buffer Memory. In *Proc. of the 4th Annual Symposium on Comp. Architecture*, pages 159–163, March 1977.

[PCG93] A. Parker, C. T. Chen, and P. Gupta. Unified System Construction. In *Proc. of the Synthesis And Simulation Meeting and International Interchange*, October 1993.

[Pen86] Z. Peng. Synthesis of VLSI Systems with the CAMAD Design Aid. In *Proc. of the 23th Design Automation Conf.*, pages 278–283. IEEE and ACM, June 1986.

[PG87] B.M. Pangrle and D.D. Gajski. Design Tools for Intelligent Silicon Compilation. *IEEE Trans. on Computer Aided Design*, pages 1098–1112, November 1987.

[PGH91] A. C. Parker, P. Gupta, and A. Hussain. The Effects of Physical Design Characteristics on the Area - Performance Tradeoff Curve. In *Proc. of the 28th Design Automation Conf.*, pages 530–534, June 1991.

[PK87] P. Paulin and J. Knight. Force-Directed Scheduling in Automatic Data Path Synthesis. In *Proc. of the 24th Design Automation Conf.*, pages 195–202. IEEE and ACM, July 1987.

[PK89] P.G. Paulin and J.P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Tran. on Computer Aided Design*, pages 661–679, June 1989.

[PK90] C.A. Papachristou and H. Konuk. A Linear Program Driven Scheduling and Allocation Method. In *Proc. of the 27th Design Automation Conf.*, pages 77–83, June 1990.

[PP88] N. Park and A. C. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Trans. on Computer-Aided Design*, March 1988.

[PP93] H. Park and V.K. Prasanna. Area Efficient VLSI Architectures for Huffman Coding. *Int. Conf. on Acoustics, Speech and Signal Processing*, 1993.

[PPM86] A. C. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proc. of the 23th Design Automation Conf.*, pages 461–466, July 1986.

[Pra78] W. K. Pratt. *Digital Image Processing*, pages 319–321. Wiley, 1978.

[PW86] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[PWGH90] A. C. Parker, Jen-Pin Weng, P. Gupta, and A. Hussain. The Effects of Physical Design Characteristics on the Quality of Synthesized Designs. In *Canadian Conf. on VLSI Design*, pages 1.1.1–1.1.7, 1990.

[RMV+88] J. Rabaey, H. De Man, J Vanhoof, G. Goossens, and F. Catthoor. *Silicon Compilation,* ed. D.D. Gajski, chapter CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems, pages 311–360. Addison-Wesley, 1988.

[RP90] J. Rabaey and M. Potkonjak. Resource Driven Synthesis in the HYPER System. In *Proc. Int'l Symposium on Circuits and Systems*, pages 2592–2595, May 1990.

[SJ94] A. Sharma and R. Jain. Estimating Architectural Resources and Performance for High-Level Synthesis Applications. In *Proc. of the 30th Design Automation Conf.*, pages 355–360, June 1994.

[Sto89] L. Stok. Interconnect Optimization during Datapath Synthesis. In *Fourth International Workshop on High-Level Synthesis*, pages 1–6, October 1989.

[Sto91] L. Stok. *Architectural Synthesis and Optimization of Digital Systems.* PhD thesis, Technishe Universiteit Eindhoven, April 1991.

[TS83] C.-J. Tseng and D.P. Siewiorek. Facet: A Procedure for the Automated Synthesis of Digital Systems. In *Proc. of the 20th Design Automation Conf.*, pages 490–496, June 1983.

[VBM91] J. Vanhoof, I. Bolsens, and H. De Man. Compiling Multi-dimensional Data Streams into Distributed DSP ASIC Memory. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 272–275, 1991.

[Wal91] G. K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):31–44, April 1991.

[WL91] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[WP91] J. Weng and A. C. Parker. 3D Scheduling: High-Level Synthesis with Floor planning. In *Proc. of the 28th Design Automation Conf.*, pages 668–673, July 1991.

[Zim79] G. Zimmermann. The MIMOLA Design System: A Computer Aided Digital Processor Design Method. In *Proc. of the 16th Design Automation Conf.*, pages 53–58. ACM SIGDA, IEEE Computer Society - DATC, June 1979.