# A Memory Selection Algorithm for High-Performance Pipelines

Smita Bakshi
Daniel D. Gajski

Technical Report #95-03
January 15, 1994

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 824-8059

sbakshi@ics.uci.edu
gajski@uci.edu

**Abstract**

*In order to perform high-throughput DSP computations, that are predominantly vector or array based, it is essential that the memory organization satisfy both the storage and the performance requirements of the design. In this report, we present an algorithm to select a memory organization, in addition to selecting a pipeline and other datapath components, given performance constraints. We also conduct experiments to give a quantitative measure of the impact of memory selection on DSP design.*

## Contents

## List of Figures

## 1 Introduction

DSP systems such as filters, FFTs and especially, image processing routines, can be characterized as high-throughput, computation-intensive systems, where the computations are most often vector or matrix based. In order to perform these high-throughput computations, the datapath typically contains a large number of functional units, possibly of different areas and speeds, and a set of memory components, that together satisfy both the storage and the performance requirements of the design. Additionally, DSP designs are usually pipelined into several concurrently executing stages, so as to achieve higher throughput values at lower costs.

DSP synthesis thus includes three important tasks. The first task, defined as component selection, is responsible for deciding the number and type of operators to be used in the design. This selection is made from a realistic library containing multiple implementations per operator type. The aim of this task is to select components such that performance constraints are satisfied at the lowest possible cost. Thus, fast components are selected only when necessary for critical operations, whereas the slower components are used for less critical operations.

The second task is responsible for selecting a memory organization that best satisfies the storage and performance requirements of the design. By a memory organization, we refer not only to the number and type of different memory components used, but also to their interconnection. For instance, a memory organization may consist of four 1-port memories connected in an interleaved fashion, so as to increase the access rate four times than that of a single memory component. Once again, the memories are selected from a library containing several different memory types (characterized by the bitwidth, number of words, number of ports, access delay, and cost), with the aim of satisfying performance constraints at the lowest possible cost.

Finally, the third task refers to pipelining, where the design is partitioned into concurrently executing stages. This results in a higher component utilization, and hence, a higher throughput for the same cost.

It is obvious that these three tasks are interdependent. For instance, the selection of components and a memory organization are dependent on each other, since the memory feeds data to functional units such that its production rate is the same as the consumption rate of the datapath. Similarly, pipelining and component selection are dependent on each other, since division into "equal-delay" stages depends on the delay of the selected components.

In this report, we present an algorithm that selects components, a memory organization, and a pipeline, so as to minimize the cost of the entire system while satisfying throughput and latency constraints. Though we describe the complete algorithm in the report, we concentrate on the impact of memory synthesis on the design.

The rest of the report is organized as follows. In the next section we give an overview of our design approach and the memory synthesis task we perform. In Section 3, we compare our work with previous work done in the area of memory synthesis and DSP synthesis. We then briefly explain our model of pipelined systems. This is followed by a formal problem definition in Section 5 and a description of the memory organizations and memory generation in Section 6. The complete algorithm is described in Section 7. We present results on several examples in Section 8 and finally conclude the report in Section 9.

## 2 Our approach

Component selection and pipelining are essential tasks for synthesizing high-performance applications. In a previous paper [4], we presented an algorithm for datapath pipelining and component selection, where the components were restricted to functional units such as adders and multipliers. In this report, we extend the algorithm to include the selection of memories and address generators (required to generate the sequence of addresses of words to be read/written to a memory).

Our approach for component and memory selection along with pipelining is summarized in Figure 1. (A more formal definition of the problem is presented in Section 5). The algorithm takes as input a $\mathcal{DFG}$, a component library, a memory library, and throughput and latency constraints. Nodes in the $\mathcal{DFG}$ represent operations and edges represent data dependencies. The component library contains functional units such as adders, multipliers, and address generators with different area and delay characteristics. The memory library contains memories with different word, bit, port, access delay, and area characteristics. As the algorithm proceeds, the component selection task makes requests for components with certain type, delay, and cost characteristics. Similarly, the memory selection task makes requests for memories with a set of characteristics, in terms of the bitwidth, number of words, number of ports, delay, and cost. Note that this request, unlike the request for components, is made to a memory generator, and not directly to the memory library. The memory generator first checks to see if a memory with the required characteristics is available in the memory library, and if such a memory is unavailable, the memory generator combines two or more memories from the library in different ways to achieve the given characteristics.

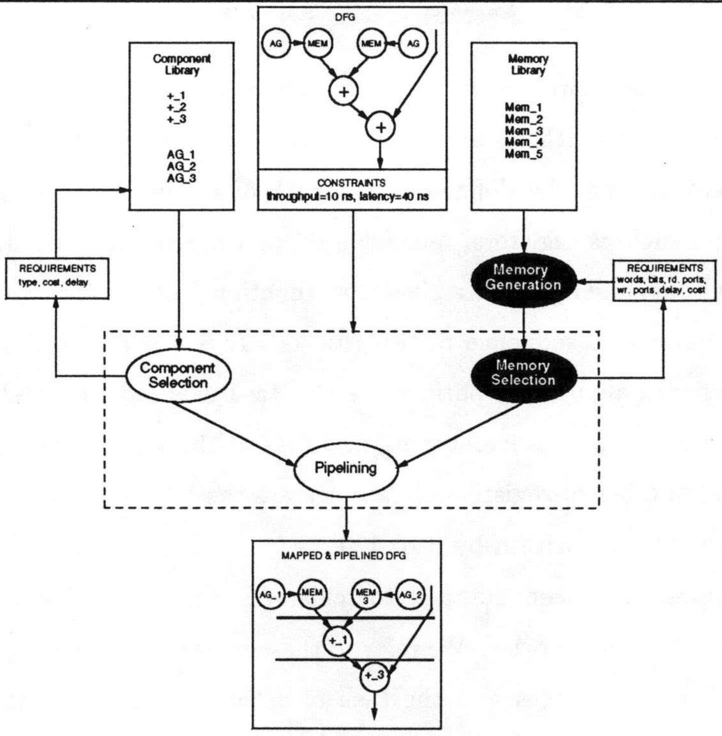


Figure 1: Our approach for component and memory selection and pipelining, with memory generation.

The selection tasks are followed by pipelining, and the three tasks are repeated iteratively, eventually resulting in a mapped and pipelined $\mathcal{DFG}$ that satisfies constraints and is of minimum cost.

The reason why we support memory generation and not component generation is twofold. Firstly, while component generators are available, memory generators are not. Thus, we can use existing component generators to increase the number of components in our library, making it unnecessary for our algorithm to generate these components. However, since memory generators are not available we are faced with a very limited set of memory components. Thus, we try and increase the variety of memories by combining two or more memories in different ways. The second reason for supporting memory generation is that DSP applications are, in general, memory intensive and they place different requirements on the bitwidth, number of words, number of ports and access delay of memories. Hence, in order to obtain cost-optimal designs it is essential to generate memories that fit these requirements as closely as possible.

In this report, we present the complete algorithm for the selection of components, memories, and a pipeline, although we explain the memory selection and memory generation tasks in greater detail.

## 3 Previous work

Having outlined our approach, we now differentiate ourselves from previous research, both in the area of memory synthesis and the area of DSP synthesis.

Memory synthesis can be defined as the task of mapping scalar or array variables to storage elements such as registers, register files, and memories, so as to satisfy constraints (such as throughput) and optimize a given cost function (such as storage area). This problem can be looked upon as a sequence of three tasks. *Task 1* consists of mapping variables to registers. In general, algorithms performing this task evaluate the *lifetime* of variables and then map variables with non-overlapping lifetimes to the same register in such a way as to reduce the total number of registers. Examples are the left-edge algorithm by Kurdahi [12] and the edge-coloring algorithm by Stok [20].

After variables have been mapped to registers, *Task 2* decides the grouping of these registers into "virtual" memories. We refer to these memories as virtual since they represent an ideal grouping of variables and the task of actually realizing them from real memory components still remains. In order to perform *Task 2*, it is essential to determine the *access patterns* of the variables. By access patterns we refer to the clock states in which a variable is read or written. The variables are grouped such that, at any given time, there are no more variable reads and/or writes than the number of available read and write ports.

Several algorithms have been proposed to perform *Task 2*. The algorithm presented in [22] uses a clique partitioning formulation to group registers to single-port memories. MIMOLA [16] groups registers to multi-ported memories without considering the effect of interconnect delays while the algorithms in [5] and [2] use a 0-1 ILP formulation and consider interconnect delays while grouping variables to multi-ported memories. The MeSA [19] algorithm and algorithms proposed by the IMEC [6], [7], [18] and Phillips [14], [23], [15] groups perform *Task 2* specifically for array variables.

The output of *Task 2* consists of a set of virtual memories characterized by a bitwidth, number of words, number of read and write ports and an access delay. *Task 3* consists of realizing these virtual memories from a given library of memory components such that the cost of the realized memory is minimized. Our algorithm performs this task, combined with component and memory selection and pipelining.

In addition to our algorithm, we are aware of one other algorithm by Karchmer and Rose [11] that solves a limited form of *Task 3*. Given a set of virtual memories each characterized by the 3-tuple <*depth* (number of words), *bitwidth*, *access time*> and given a real memory also characterized by the same 3-tuple, this algorithm maps the virtual memories to one or more real memories so as to minimize the cost of the final memory organization. The major difference between our work and [11], is that we solve the problem of memory generation with component and memory selection and pipelining, whereas they solve the memory generation problem in isolation. Furthermore, they offer a limited solution since they can only generate larger memories from one type of component, whereas we can generate larger and faster memories from different types of components. For instance, they can map an array of size 10 to 2 memories of size 5 each; however, if the access time requirement is, say 10 *ns*, their algorithm is unable to map this requirement to memories of delay greater than 10 *ns*. They can only provide a solution for a memory of delay less than 10 *ns*. By using techniques such as interleaving, we can do the mapping with memories of any delay. Other differences include their use of only one real memory type, whereas we place no restrictions on the available memory types.

Having put our work in perspective with previous research in memory synthesis, we now compare our approach to DSP synthesis with the IMEC and Phillips (Phideo) approaches.

Phideo performs datapath synthesis, memory synthesis and finally address generator synthesis in sequence, one after the other while we combine these three tasks since they are all inter-related and the synthesis of one of them depends on the other. We believe this results in more cost-efficient designs. Secondly, Phideo uses single implementations for functional units and address generators whereas we use multiple implementations of components in a design, such that fast components are used on critical paths and the slower components on non-critical paths. Thirdly, Phideo uses a limited memory library with only two memory components: type 1 has 1 R/W port while type 2 has 2 R/W ports. We do not place any restrictions on memory components. Finally, unlike Phideo, we have integrated our selection and pipelining algorithm with memory generation (*Task 3*), giving us the ability to realize our selected memories using different memory organizations such as horizontal, vertical, and interleaved (described in Section 6.1).

We differ from IMEC for similar reasons. They first perform memory synthesis, then datapath synthesis, while we do these tasks simultaneously. Their approach works well for memory-dominating systems, whereas our approach works well for systems in which memory may or may not be the dominating cost factor. This is because by making all memory decisions independent of the datapath, they may select a memory which forces the use of an expensive datapath component, resulting in an overall high cost.

Other differences between IMEC and our work include their use of a more limited library of memories where all memories have the same access delay. Furthermore, they do not perform memory generation and hence, do not support the use of different memory organizations such as interleaved, horizontal and so on. Note that our algorithm for memory generation can serve as a backend for the memory synthesis tasks performed by the IMEC and Phillips groups.

## 4 Model of pipelined systems

Before proceeding with the formal problem definition and algorithm, we would like to describe the underlying model of pipelined systems. The entire system can be viewed as a sequence of communicating pipelined FSMDs (Finite State Machine with Datapath). As an example, consider the MPEG system [21] shown in Figure 2. It consists of six pipelined FSMDs namely the *Decoder*, *Dequantizer*, *IDCT*, *Sum*, *Predictor* and *Display*. The system is pipelined, both at the block level (that is within an FSMD) and at the system level (that is between consecutive FSMDs).

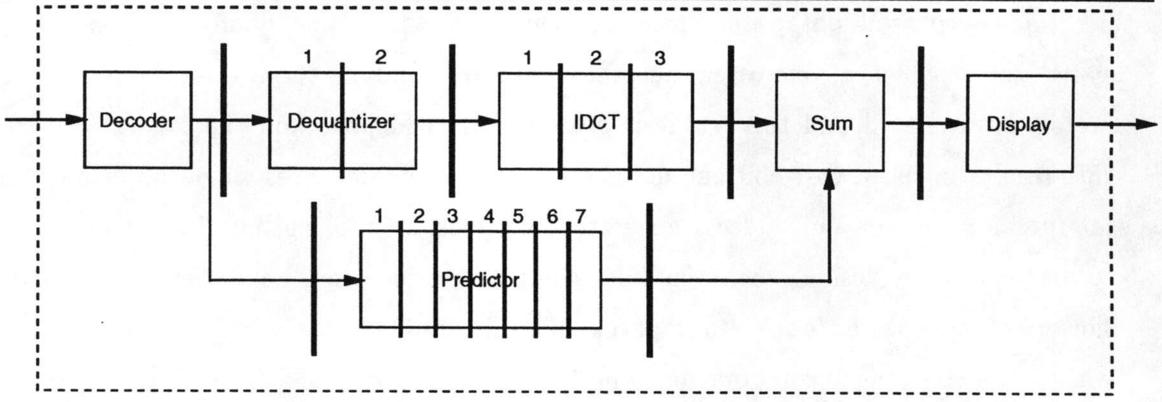


Figure 2: The MPEG: an example of a pipelined system.

Thus, at the system level, while the *Decoder* is operating on the *ith* input sample (assume the input sample is an $8 \times 8$ matrix), the *Dequantizer* is operating on the *(i-1)th* input sample, the IDCT on the *(i-2)th*, and so on. Similarly, within the Dequantizer, while *stage_1* is operating on the *ith* input sample (which is, say, one element of the $8 \times 8$ matrix), *stage_2* is operating on the *(i-1)th* sample, and so on.

Figure 3 zooms into the *Dequantizer*, *IDCT*, and *Sum* blocks and it is used to illustrate the interface between two pipelined FSMDs. The Dequantizer consists of a controller and a 2-stage pipelined datapath. Its output, *Array_A*, a 64-element array, is consumed by the *IDCT* which consists of a controller and a 3-stage pipelined datapath with two memories. The output of the *IDCT*, in turn, is consumed by the *Sum* block.

In order to maintain the flow of data in the pipeline it is assumed that the consumption

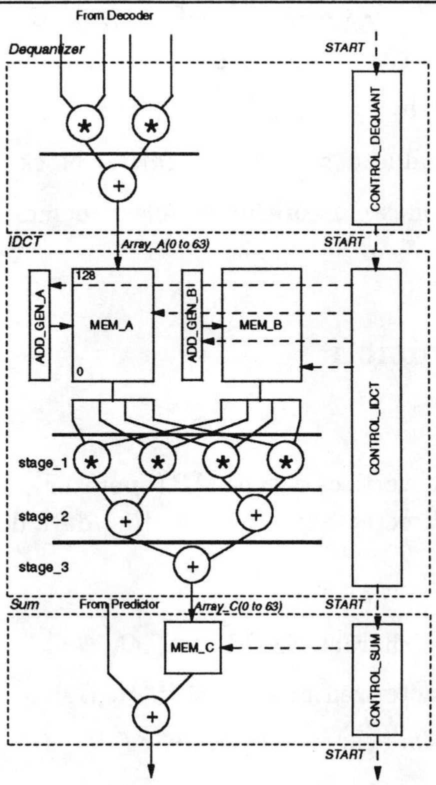


Figure 3: The interface between pipelined FSMDs.

and production rate of an input is the same. Thus, in Figure 3, the *Dequantizer* block produces a sample of the 64-element *Array_A*, say, every 320 *ns* and the *IDCT* consumes it at the same rate. Similarly, the *IDCT* produces a sample of *Array_C* every 320 *ns* and the *Sum* block consumes it at the same rate. This rate is also known as the *throughput* of the system. Within, the *IDCT* however, the rate can be different. For instance, 4 words are read from MEM_A every 5 *ns* and 1 word is written into MEM_C every 5 *ns*.

We make the assumption that all pipelined FSMDs require one sample per input before they can begin processing data. Thus, each producer FSMD sends a START signal to its consumer after producing its first output. An FSMD can start after all its producers have sent START signals indicating that the first set of input samples is available. Thus, in Figure 3, the *Dequantizer* sends a START signal to the IDCT after producing the first set of 64 *Array_A elements.* The IDCT then starts and after producing the first sample of *Array_C* it, in turn, sends a START signal to the *Sum* block. After that point, data is produced and consumed at regular intervals and the START signals need not be asserted any longer.

We make a further assumption regarding the size of the memory. Each pipelined FSMD should have sufficient memory to store two sets of samples per input, one the sample being currently used and the other, the sample being currently produced by the preceding FSMD. For instance, Mem_A has 128 words where 64 words are used for say, the *ith* sample of Array_A being consumed by the *IDCT* block, and the other 64 words are used for the $(i+1)th$ sample being produced by the *Dequantizer* block.

In this report we present an algorithm to select memories, datapath components and a pipeline for each FSMD.

## 5 Problem definition

**Given:**

1. A $\mathcal{DFG}(V,E)$ with $V$ vertices of type $OP$ (operator), $MEM$ (memory) or $AG$ (address generator), and $E$ directed edges representing data dependencies.
2. A library $\mathcal{LIB}$ of
   (a) functional units characterized by $<T,C,D>$,
   (b) memories characterized by $<\mathrm{W},B,P_r,P_w,C,D>$, and
   (c) address generators characterized by $<T,C,D>$.
3. Constraints on $PS$ (pipe stage) *delay* and *latency*.

**Determine:**

1. A mapping of
   (a) $OP$ vertices to functional units,
   (b) $MEM$ vertices to a *memory organization*, and
   (c) $AG$ vertices to address generators.
2. A division of the DFG into pipe stages.

**Such that:**

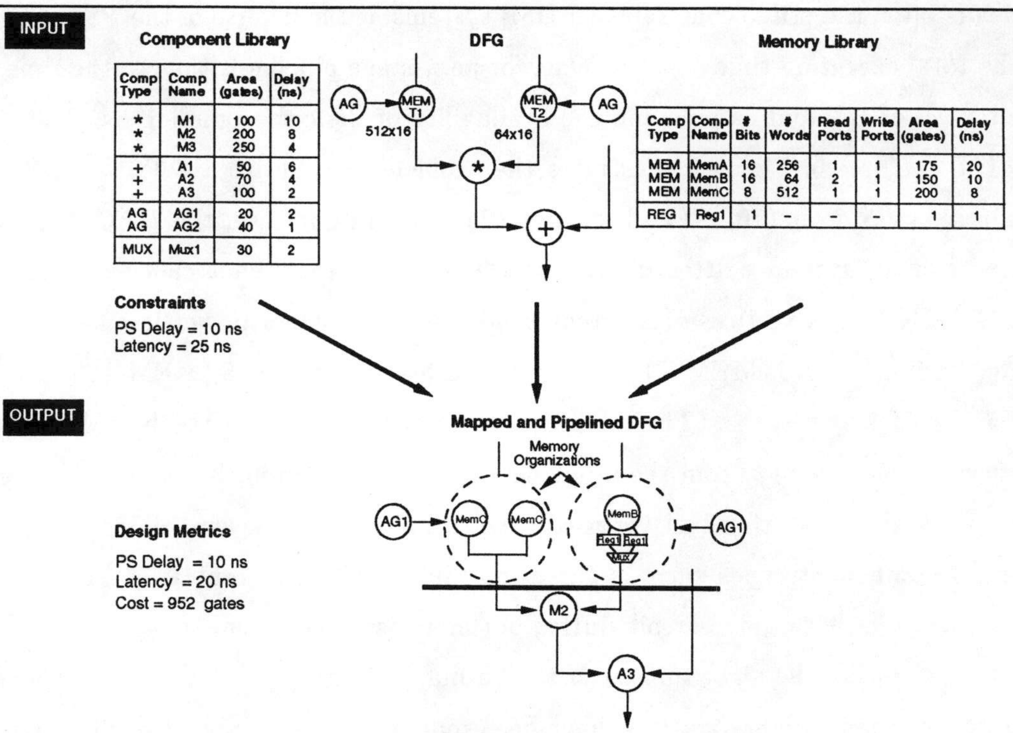
1. *PS delay* and *latency* constraints are satisfied, and
2. Cost of functional units, address generators and memories is minimized.

The elements of the 3-tuple $<T,C,D>$ refer to the type, cost, and delay of functional units and address generators. (We support three different address generator types, discussed later). The elements of $<\mathrm{W},B,P_r,P_w,C,D>$ refer to the number of words, the bitwidth, the number of read ports and write ports, and the cost and access delay of the memory, respectively. Design constraints may be placed on both the *PS delay* and the *latency* of the design. *PS delay* is the sample inter-arrival delay, that is, the delay between the arrival of two consecutive input samples. This is also the clock cycle of the design. Throughput, which is often the prime constraint on DSP systems, is the inverse of the *PS delay*. *Latency* is the total execution time ($n \times$*PS delay*, for an $n$-stage pipeline), that is, the time between the arrival of an input sample and the availability of the corresponding output.

The example in Figure 4 illustrates the problem. Given are a $\mathcal{DFG}$, a $\mathcal{LIB}$, and constraints on *PS delay* (10 *ns*) and *latency* (25 *ns*). (In our case, the $\mathcal{DFG}$ is derived from a model of the system written in a hardware description language called SpecCharts [17]). The $\mathcal{DFG}$ contains vertices of different types: address generator vertices (AG), a memory vertex of size 512$\times$16 (MEM T1), a memory vertex of size 64$\times$16 (MEM T2), and multiply ($\star$) and add (+) vertices. The memory vertex sizes as well as the bitwidth of all other vertices are determined from the input description. (For simplicity, we have not explicitly mentioned the bitwidth of all vertices). The $\mathcal{LIB}$ contains several different implementations of component types such as three multipliers with different area and delay values, three memories with different bitwidths, ports, access delays, and so on.

The output of the algorithm consists of a mapped and pipelined $\mathcal{DFG}$ in which all vertices, other than memory vertices, have been mapped to a component of the corresponding type from the library, while memory vertices have been mapped to one or more memory components. The vertices have also been partitioned into 2 stages of delay 10 *ns* each. The mapping and the partitioning has been done such that the total cost of the design (952 gates) is minimized. Note, that in the output $\mathcal{DFG}$ the memory vertices need not directly correspond to a single memory component from the library, $\mathcal{LIB}$, but to a *memory organization* which consists of an interconnection of one or more memories, registers and multiplexers. The memory organization is designed such that it best satisfies size constraints imposed by the $\mathcal{DFG}$ and performance constraints imposed by the input *PS delay* and *latency* values. As an example, consider the vertex MEM T1 of size 512 $\times$ 16. This has been mapped to two *MemC* components of size 512 $\times$ 8 so as to satisfy bitwidth requirements. Similarly, the vertex MEM T2 has been mapped to *MemB* with its two read ports multiplexed, such that one word can be read every 5 *ns*.

Associated with every memory vertex is an *AG* or address generation vertex which is required to generate the sequence of addresses of words to be read from the memory. We support three different address generator types, though only one type has been shown in the example. Once again, the address generator type associated with each memory vertex is determined from the input description.



| Comp Type | Comp Name | Area (gates) | Delay (ns) |
|---|---|---|---|
| * | M1 | 100 | 10 |
| * | M2 | 200 | 8 |
| * | M3 | 250 | 4 |
| + | A1 | 50 | 6 |
| + | A2 | 70 | 4 |
| + | A3 | 100 | 2 |
| AG | AG1 | 20 | 2 |
| AG | AG2 | 40 | 1 |
| MUX | Mux1 | 30 | 2 |

| Comp Type | Comp Name | # Bits | # Words | Read Ports | Write Ports | Area (gates) | Delay (ns) |
|---|---|---|---|---|---|---|---|
| MEM | MemA | 16 | 256 | 1 | 1 | 175 | 20 |
| MEM | MemB | 16 | 64 | 2 | 1 | 150 | 10 |
| MEM | MemC | 8 | 512 | 1 | 1 | 200 | 8 |
| REG | Reg1 | 1 | – | – | – | 1 | 1 |

Figure 4: An example illustrating the inputs and outputs of the component selection, memory selection, and pipelining algorithm.

## 6 Memory selection

Having given a formal definition of the problem, we now describe the different memory organizations, the algorithm for memory generation, and the address generator types.

### 6.1 Memory organizations

A *memory organization* refers to an interconnection of one or more memory components of a given type, along with registers and multiplexers such that the resulting memory organization has a greater number of words, bitwidth or ports, or a smaller access delay than the memory component from which it is built. We define *memory generation* as the task of generating a memory organization given a memory component and a set of bitwidth, word, port, and access delay requirements.

Figure 5(a) depicts a basic memory component characterized by $<W,B,P_r,P_w,C,D>$, which are the number of words, bits, read ports, write ports and the cost and access delay, respectively. Figures 5(b) to (f) depict five memory organizations obtained by connecting one or more of the basic memory components in different ways. We now define the characteristics of each of the memory organizations with respect to the characteristics $<W,B,P_r,P_w,C,D>$ of the basic memory component and the characteristics $<W',B',P'_r,P'_w,C',D'>$ of the desired

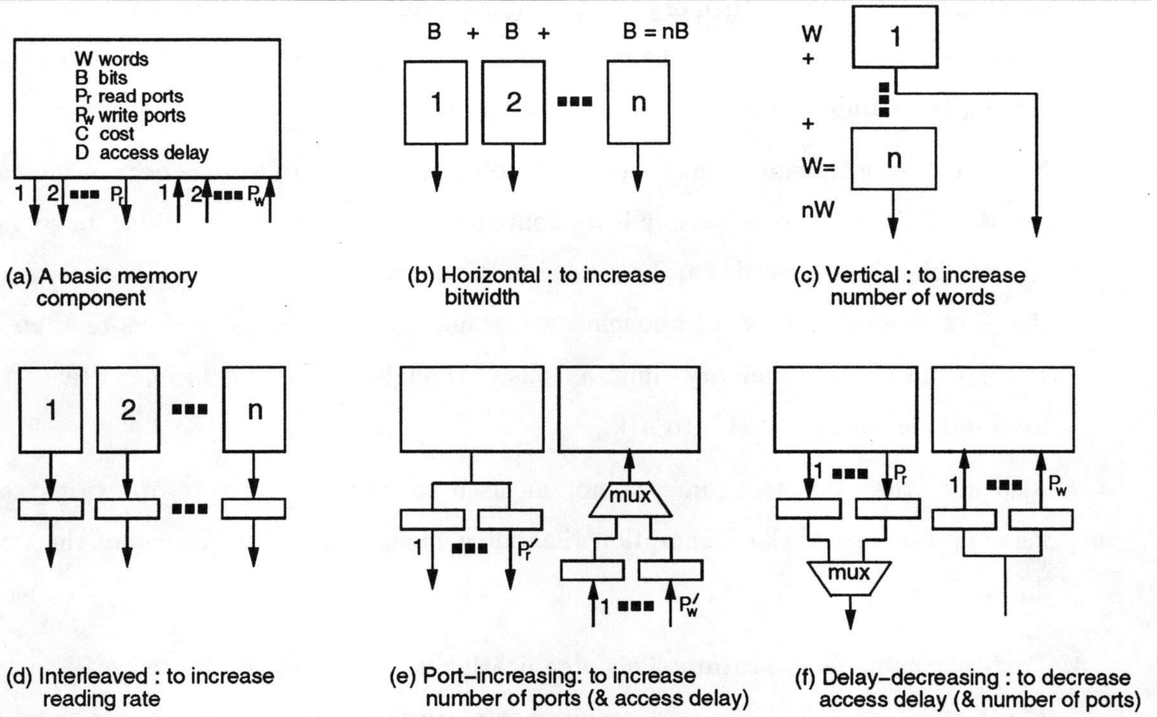

W words
B bits
Pr read ports
Pw write ports
C cost
D access delay
1 2 Pr 1 2 Pw
(a) A basic memory component
B + B + B = nB
1 2 n
(b) Horizontal : to increase bitwidth
W + + W= nW
1 n
(c) Vertical : to increase number of words
1 2 n
(d) Interleaved : to increase reading rate
mux
1 Pr'
1 Pw'
(e) Port-increasing: to increase number of ports (& access delay)
1 Pr
1 Pw
mux
(f) Delay-decreasing : to decrease access delay (& number of ports)

Figure 5: Different memory organizations to increase the bitwidth, number of words or ports, or to decrease the access delay of a basic memory component.

memory component.

1. *Horizontal Organization:* This organization is used when the desired bitwidth $B'$ is greater than $B$, the bitwidth of the basic memory component. Let $n = \lceil B'/B \rceil$. Then, the characteristics of the memory organization are given by:

$$<W,nB,P_r,P_w,nC,D>$$

2. *Vertical Organization:* This organization is used when the required number of words, $W'$, is greater than $W$. Let $n=\lceil W'/W \rceil$. Then, the new characteristics are:

$$<nW,B,P_r,P_w,nC,D>$$

3. *Interleaved Organization:* This organization is used when the desired reading rate $P_r'/D'$, is greater than the available reading rate $P_r/D$. Let $n=\lceil \frac{P_r'}{D'}/\frac{P_r}{D} \rceil$. Then, the memory organization's characteristics are:

$$<W,B,n \times P_r,P_w,n \times (C+(P_r \times Cost_{reg})),D>$$

Here, $Cost_{reg}$ refers to the cost of a $B$ bit register. To illustrate the need for an interleaved organization consider the case when the desired reading rate is $1 word/20 ns$ ($P_r'$=1, $D'$=20 $ns$) and the available memory component has $P_r$=2 and $D$=100 $ns$. We could then interleave 3 memory components giving us a resulting read rate of 6/100 (which is greater than 1/20). Thus, interleaving essentially increases the reading rate, that is, the number of words read per unit time.

Note, that even though $n$ components are used, the number of words does not increase to $nW$. This is because we duplicate contents in all $n$ memory components so that any combination of words can be read simultaneously. If we had actually partitioned the data amongst the $n$ components we would not be able to read more than $P_r$ words from a given memory simultaneously, though, this would have increased the total number of words, $W'$, to $nW$.

Also note that this technique cannot be used to increase the rate of writing to a memory because of the assumption that all $n$ memories contain copies of the same data.

4. *Port-increasing Organization:* This organization is used when the desired reading rate $P_r'/D'$ is satisfied, but the port requirements are not, that is $P_r$ is smaller than $P_r'$. Let $n = \lceil P_r'/P_r \rceil$. Then, the new characteristics are:

$$<W,B,n \times P_r',P_w,(C+(n \times P_r \times Cost_{reg})),n \times D>$$

To illustrate the need for this organization, consider a case when the desired rate is $4words/100\ ns$ ($P_r'$=4, $D'$=100 $ns$) and the basic memory component has $P_r$=1 and $D$=25 $ns$. In this case, $P_r'/D' = P/D$ but $P_r < P_r'$. Words can then be accessed serially, and after 100 $ns$ four words can be read out in parallel.

This organization can also be used when the desired writing rate is satisfied but $P_w$ is smaller than $P_w'$. Once again, words can be written in parallel into registers and then into the memory, serially, one after the other.

In summary, this organization has a larger number of ports and access delay than the basic memory component.

5. *Delay-decreasing Organization:* This organization is used when the desired reading rate $P_r'/D'$ is satisfied but the access delay, $D > D'$. Let n = $\lceil D/D' \rceil$. Then, the characteristics of the memory organization are:

$$<W,B,P_r/n,P_w,(C+(P_r \times Cost_{reg})+(P_r' \times Cost_{n\times 1\_mux})),D/n>$$

Here, $Cost_{n\times 1\_mux}$ is the cost of an $n \times 1$ multiplexer. To illustrate the need for this organization, consider a case when the desired rate is $1word/25\ ns$ ($P_r'$=1, $D'$=25 *ns*) and the basic memory component has $P_r$=4 and $D$=100 *ns*. Words can then be accessed from the memory in parallel 4 at a time, and read serially, 1 every 25 *ns*.

This organization can also be used for the write ports in a similar way when the desired writing rate is satisfied but $D > D'$.

In summary, this organization has a lower access delay and a smaller number of ports than the basic memory component.

## 6.2 Memory generation

Having described the 5 different memory organizations, we now formally define the problem of memory generation.

*Given a memory component M characterized by* $<W,B,P_r,P_w,C,D>$ *and a virtual (or desired) memory characterized by* $<W',B',P_r',P_w',D'>$ *build a memory organization with characteristics* $<W'',B'',P_r'',P_w'',C'',D''>$ *such that* $W''\geq W'$, $B''\geq B'$, $P_r''\geq P_r'$, $P_w'\geq P_w''$, $D'\leq D''$ *and* $C''$ *is minimized.*

In Figure 6 we outline the procedure for generating a memory $<W'',B'',P_r'',P_w'',C'',D''>$ given a memory component $<W,B,P_r,P_w,C,D>$ and a set of desired characteristics $<W',B',P_r',P_w',D'>$. This procedure will be utilized in the main algorithm for selection and pipelining presented in Section 7.

We start with a memory called *new_mem* and initialize its attributes $<W'',B'',P_r'',P_w'',C'',D''>$ to the attributes of the given memory component $<W,B,P_r,P_w,C,D>$ (step 1). We then check the bitwidth of *new_mem* (step 2) and if it is less than the required bitwidth we horizontally connect $\lceil B'/B''\rceil$ *new_mem* components (step 4). This *Horizontal* connection is now of the required bitwidth. Next, we check to see that there are a sufficient number of words (step 8), and once again, if needed, we use a *Vertical* organization to build up the required words. Next, in steps 14 and 20 we check to see if the required reading and writing rate are met by *new_mem*. If the reading rate is not met, we build it up by *Interleaving* $\lceil \frac{P_r'}{D'}/\frac{P_r''}{D''}\rceil$ memories; however, if the writing rate is not met we are forced to exit the program since we are unable to increase the writing rate. By the time we reach steps 23 and 29 *new_mem* has the desired writing and reading rate. However, it may have too large or too small an access delay in which case we use a *Delay-decreasing* or *Port-increasing* organization to get the required read and write ports and access delay. Note, that Figure 6 only shows the the delay-decreasing and port-increasing organizations for read ports; the organizations for write ports are similar.

```
Begin Generate_Memory ($<W',B',P'_r,P'_w,D'>$, $<W,B,P_r,P_w,C,D>$)
    1. new_mem $<W'',B'',P''_r,P''_w,C'',D''>$ = memory component $<W,B,P_r,P_w,C,D>$
    2. If ($B' > B''$)
    3.      $n = \lceil B'/B'' \rceil$
    4.      new_mem = Horizontal(new_mem, n) (Figure 5(b))
    5.      $C'' = n \times C''$
    6.      $B'' = n \times B''$
    7. End if
    8. If ($W' > W''$)
    9.      $n = \lceil W'/W'' \rceil$
    10.     new_mem = Vertical(new_mem, n) (Figure 5(c))
    11.     $C'' = n \times C''$
    12.     $W'' = n \times W''$
    13. End if
    14. If ($P'_r/D' > P''_r/D''$)
    15.     $n = \lceil \frac{P'_r}{D'} / \frac{P''_r}{D''} \rceil$
    16.     new_mem = Interleave(new_mem, n) (Figure 5(d))
    17.     $C'' = (n \times C'') + (n \times P''_r \times Cost_{reg})$
    18.     $P''_r = n \times P''_r$
    19. End if
    20. If ($P'_w/D' > P''_w/D''$)
    21.     Cannot increase write rate. Exit Generate_Memory.
    22. End if
    23. If ($P'_r > P''_r$)
    24.     $n = \lceil P'_r/P''_r \rceil$
    25.     new_mem = Port-increasing(new_mem, n) (Figure 5(e))
    26.     $C'' = C'' + (n \times P''_r \times Cost_{reg})$
    27.     $P''_r = n \times P''_r$
    28.     $D''_r = n \times D''_r$
    29. Else If ($D' < D''$)
    30.     $n = \lceil D''/D' \rceil$
    31.     new_mem = Delay-decreasing(new_mem, n) (Figure 5(f))
    32.     $C'' = C'' + (P''_r \times Cost_{reg}) + (P''_r/n \times Cost_{n \times 1\_mux})$
    33.     $P''_r = P''_r/n$
    34.     $D''_r = D''_r/n$
    35. End if
    36. Output (new_mem $<W'',B'',P''_r,P''_w,C'',D''>$)
End Generate_Memory
```

Figure 6: Algorithm for generating a memory given a set of required characteristics and a memory component.

Finally, in step 36 we output *new_mem* $<W'',B'',P_r'',P_w'',C'',D''>$ where $W'' \geq W'$, $B'' \geq B'$, $P_r'' \geq P_r'$, $P_w' \geq P_w''$ , $D' \leq D''$ and $C''$ is minimized.

## 6.3 Address generator types

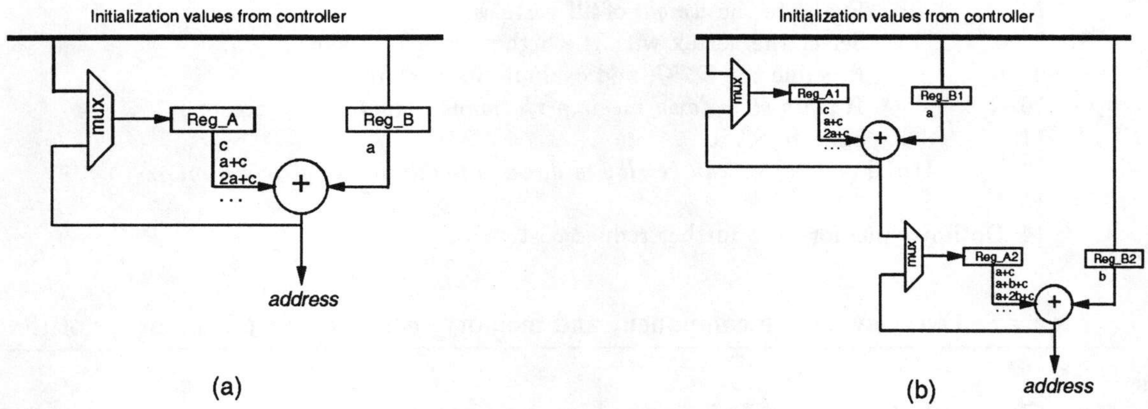


Figure 7: Address generators (a) *Type 2* (b) *Type 3*.

Most array accesses in DSP applications follow a regular pattern that may be represented by the linear expression $(C_1I_1 + C_2I_2 + \ldots + C_nI_n + C_{n+1})$, where $C_1$, $C_2$ ... $C_{n+1}$ are constants and $I_1$, $I_2$ ... $I_n$ are loop indices. In our designs, we make the assumption that all array accesses fit the expression $ai + bj + c$ (that is, $n$=2 in the above expression), where at least one term is non-zero. Thus, we support the following three address generator types:

1. *Type 1*: a=b=0. Thus for an array $A$, the access is of the form $A(c)$. The address generator consists of a register to store the value of $c$.

2. *Type 2*: b=0. Access is of the form $A(ai + c)$. The address generator required for this address sequence is shown in Figure 7(a). It essentially consists of an adder and two registers with *Reg_A* initialized to $c$ and *Reg_B* to $a$. At every loop iteration the value of $a$ is added to the previous *address* resulting in the next *address*.

3. *Type 3*: Access is of the form $A(ai + bj + c)$. The address generator (Figure 7(b) required for this address sequence is similar to the address generator for *Type 2* except that it contains two adder-register pairs, one for each loop index.

The library contains multiple implementations of these three address generator types, obtained by using adders of different implementations.

```
1. Generate memories of specified sizes and a range of access delays.
2. Map all vertices to fastest component of corresponding type, and evaluate performance.
3. If (fastest design does not satisfy constraints)
4.        exit the program.
5. Else
6.        Loop
7.                Evaluate the weight of all vertices.
8.                Select the vertex with the highest weight to slow down.
9.                Pipeline the DFG, and evaluate its performance.
10.               If (this slow down meets performance constraints), accept it
11.               else, reject it.
12.       Until (no vertex can be slowed down without violating constraints).
13. End if
14. Optimize memories to further reduce cost.
```

Figure 8: Overview of the component and memory selection and pipelining algorithm.

## 7 Complete algorithm: component selection, memory selection, and pipelining

Having defined the problem, we now present the algorithm for component selection, memory selection and pipelining with memory generation. We first give an overview of the complete algorithm and then explain individual steps in more detail.

### 7.1 Overview

The algorithm takes as input a $\mathcal{DFG}$, a component library $\mathcal{LIB}$, and a constraint on the *PS delay* and *latency*. It outputs a mapped $\mathcal{DFG}$ where each memory vertex is mapped to a memory organization built out of memory components in $\mathcal{LIB}$ and all other vertices are mapped to components of the corresponding type. The $\mathcal{DFG}$ is also partitioned into $\lfloor latency/PS\ Delay \rfloor$ stages, such that the delay of each pipe stage is less than or equal to the *PS Delay* and the total area of the $\mathcal{DFG}$ is minimized.

The algorithm (Figure 8) starts by generating a list of larger and faster memories from the memory components available in the input library. The size, port, and delay characteristics of these new memories is determined from the size of the memory nodes and from the *PS delay* constraint. Essentially, we enhance the memory library by making sure that it contains memories of the required sizes and delays. This enhanced library can then be used by the rest of the selection and pipelining algorithm.

The procedure to generate these memories is outlined in Figure 9. As input we have a list of all the memory vertices with their sizes, a list of memory components given in

```
Begin Enumerate_All_Memories(memory vertices, memory components, PS delay, MIN, INC )
  1.  For (all memory vertices) loop
  2.      $W'$ = number of words of (current) memory vertex.
  3.      $B'$ = bitwidth of memory vertex.
  4.      $P_r'$ = number of outgoing arcs of memory vertex.
  5.      $P_w'$ = number of incoming arcs of memory vertex.
  6.      For (all memory components) loop
  7.          $<W,B,P_r,P_w,C,D>$ = current memory component.
  8.          max_delay= min(PS delay, D).
  9.          $D'$= MIN.
  10.         Loop
  11.             new_mem = Generate_Memory ($<W',B',P_r',P_w',D'>$, $<W,B,P_r,P_w,C,D>$)
  12.             Add new_mem to component library.
  13.             $D' = D' +$ INC.
  14.         Until ($D' \leq$ max_delay).
  15.     End Loop
  16. End Loop
End Enumerate_All_Memories
```

Figure 9: Procedure for enumerating memories for a given DFG and component library.

the input component library, and the *PS delay* constraint imposed by the designer. For each memory vertex we then generate new memories from each of the memory components using the procedure *Generate_Memory* in Section 6.2. These new memories have the same number of words and bits as the memory vertex, and the same number of read and write ports as the number of outgoing and incoming arcs, respectively. (Outgoing arcs indicate a read operation, while incoming arcs indicate a write operations). The memories have a range of access delays, that start from a specified minimum value (MIN) and vary in fixed size increments (INC) till a maximum value is reached. For instance, the minimum value in the experiments presented in this report is 5 *ns* and the increment is 3 *ns*. The maximum value is either equal to the *PS delay* constraint or the delay of the memory component, whichever is lower.

The procedure essentially contains a triple nested loop, one to iterate over all memory vertices, the other to iterate over all memory components and the third to iterate over access delay values. This procedure could potentially be expensive in terms of CPU time; however, for most DSP applications and for most existing component libraries the number of memory vertices and the number of memory components is quite small (approximately, under 30 each), leading to a low execution time.

After generating these new memory organizations, we perform the core component and memory selection and pipelining task. We first map each vertex to the fastest component of the corresponding type in the library. Thus, all $\star$ nodes are mapped to the fastest multiplier, all $+$ nodes to the fastest adder, all MEM nodes to the fastest memory of the corresponding size and so on. Since this is the fastest possible design it is also the most expensive. We then try and pipeline this design into $\lfloor latency/PS\ Delay \rfloor$ stages of delay *PS delay* each. If we are unable to do this, it implies that the fastest design cannot meet constraints - the only option left is to try again with lower design constraints or a library with faster components.

If the fastest design satisfies constraints, our next step is then to slow down the design by replacing some vertices by slower and cheaper components, such that cost is minimized and constraints are still satisfied. Intuitively speaking, the aim of the algorithm is to slow down as many vertices by as much as possible, and this is achieved by balancing the use of slow and fast components so that the delay of each pipe stage is as close to *PS delay* as possible, and the total cost is minimized.

The slowing down process is carried out iteratively (steps 6 to 12 in Figure 8), by first assigning a *weight* to all vertices, and then replacing the highest *weight* vertex with a slower component. After each slow down, the $\mathcal{DFG}$ is pipelined to ensure that the slow down does not violate constraints. This process is repeated till there are no vertices left to slow down or performance constraints cannot be satisfied any longer. This then will be the minimum-cost design.

The key to the algorithm lies in judiciously selecting vertices to be slowed down in each iteration, since slowing down one vertex may prevent slowing down others due to graph dependencies. Thus, the desirability of slowing down a vertex has to be evaluated with respect to all the vertices that would be affected by its slow down. The vertex *weight* represents this measure of desirability or priority in the slowing down process. The vertex *weight* is briefly described below; a more detailed explanation can be found in [4].

## 7.2 Vertex weight

The *weight* $W(v)$ of a vertex $v$, that is currently mapped to a component $c'$, and that is going to be replaced by a slower component $c''$, is given by:

$$W(v) \quad = \quad \frac{\mathrm{ADG}(v, c', c'')}{\text{Commonality Facto r}(v)} \tag{1}$$

where the area-delay gain of $v$ with respect to $c'$ and $c''$, $ADG(v, c', c'')$ is:

$$ADG(v, c', c'') \quad = \quad \frac{Area(c') - Area(c'')}{Delay(c'') - Delay(c')} \tag{2}$$

The vertex *weight* depends on three factors. First, the area reduction incurred by replacing $c'$ with $c''$. The higher the area reduction, the higher the *weight* indicating the desirability of slowing down the vertex. The second factor is the increase in delay or the extent of the slow down. Higher the increase in delay, lower is the desirability of slowing down a vertex since a large increase in delay could prevent further slow downs.

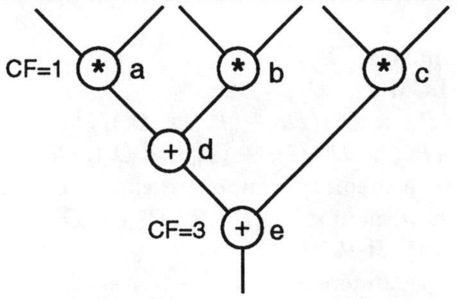


Figure 10: An example to illustrate the commonality factor (CF) of a vertex.

The third factor, defined as the commonality factor (CF) of the vertex, is essentially a measure of the number of input-output (I-O) paths that the vertex is contained in. The slow down of a vertex affects all the paths that it is contained in, and it decreases the possibility of slowing down other vertices in the paths. As an example, consider the $\mathcal{DFG}$ in Figure 10. Vertex $e$ has a CF of 3 since it is contained on three I-O paths ($a-d-e$, $b-d-e$, $c-e$) while CF(a)=1 since $a$ is contained on only one I-O path ($a-d-e$). Let us assume that we can afford to incur a total slow down of only 10 $ns$. If we were to slow down $e$ by 10 $ns$ we would not be able to slow down any other vertex since $e$ is contained on all I-O paths. However, if we slowed down $a$, we could also slow down $b$ and $c$ since $a$ is not on the same I-O paths as $b$ and $c$.

These three factors, the area reduction, the delay gain, and the commonality factor are combined as shown in equations (1) and (2), to give the vertex *weight.*

We now explain the next two steps (steps 9 and 14) of the component selection and pipelining algorithm in Figure 8, namely the procedure for pipelining the $\mathcal{DFG}$ into equal delay stages, and the procedure for optimizing the memory selection.

## 7.3 Pipelining

Given a $\mathcal{DFG}$, in which every vertex has been associated with a component, and a *PS delay* constraint, the pipelining algorithm partitions the $\mathcal{DFG}$ into a minimal number of stages of delay no more than *PS delay.* It traverses the graph in two directions, downward (from the input to the output nodes), and upward (from output to input nodes). As it traverses the graph it keeps accumulating the delay from the boundary of the last pipe stage. A new boundary is set when the performance constraint can no longer be satisfied.

```
$Optimize\_Memory$ (mapped $\mathcal{DFG}$, component library)
1. Get list of only memory nodes from the $\mathcal{DFG}$.
2. Repeat
3.     For (each pair of memory nodes, $i$ and $j$, mapped to memory components characterized by
       $<W_i,B_i,P_{ri},P_{wi},C_i,D_i>$ and $<W_j,B_j,P_{rj},P_{wj},C_j,D_j>$)
4.         $W' = W_i + W_j$.
5.         $B' = \max(W_i, B_j)$.
6.         $D' = \mathrm{LCM}(D_i, D_j)$.
7.         $P'_r = (P_{ri} \times D')/D_i + (P_{rj} \times D')/D_j$.
8.         $P'_w = (P_{wi} \times D')/D_i + (P_{wj} \times D')/D_j$.
9.         For (each memory component characterized by $<W,B,P_r,P_w,C,D>$)
10.            new_mem $<W'',B'',P''_r,P''_w,C'',D''>$ = $Generate\_Memory(<W',B',P'_r,P'_w,D'>,$
               $<W,B,P_r,P_w,C,D>)$
11.            cost_difference = $(C''$ - $(C_i + C_j))$
12.            If (cost_difference is highest so far)
13.                best_pair=i and j.
14.            End If
15.        End For
16.        Combine best_pair, i and j, add combined node to list, and remove i and j from list.
17.    End For
18. Until (no further cost reduction or only one memory node in list)
End $Optimize\_Memory$
```

Figure 11: Procedure for optimizing memories.

The traversal is repeated for both directions, and the pipeline with the fewer number of "cuts" is selected. A "cut" refers to the intersection of an edge of the $\mathcal{DFG}$ with the pipe stage partition, and it corresponds to a pipeline register. Hence, the fewer the number of cuts, the fewer the pipeline registers.

## 7.4 Memory optimization

In steps 6-12 of Figure 8 we iteratively decrease the cost of the $\mathcal{DFG}$ by slowing down vertices, including all memory vertices. However, during this process we assume a one-to-one mapping between vertices and components, and we do not consider the possibility of mapping two or more memory vertices to a single memory component. Mapping two memory vertices to one larger or faster memory component rather than to two smaller or slower ones, could possibly lead to a cost-reduction, if the area of the larger memory is less than the combined area of the two smaller memories.

For instance, two memory nodes of size $16 \times 8$ each, that are mapped to two memories with the characteristics $<16 \times 8$, 1 read port, 200 gates, 50 $ns>$, could instead be mapped to a single memory component with characteristics $<32 \times 8$, 2 read ports, 50 $ns>$, or $<32 \times 8$, 1 read ports, 25 $ns>$, if the single memory component has a lower cost than 400 gates.

The procedure for optimizing the memory selection is shown in Figure 11. We first form a *list* of all the memory nodes in the $\mathcal{DFG}$. For all pairs of memory nodes, we then form a new set of characteristics $<W',B',P'_r,P'_w,D'>$ that a memory would need to store both nodes $i$ and $j$. These characteristics are evaluated as shown in steps 4 to 8. We then generate memories with these characteristics from the memory components in the library, using the procedure *Generate_Memory*. This is indicated in steps 9 to 15 of the algorithm.

As we generate these new memories we keep track of the node pair, say $i$ and $j$, that gives the highest cost reduction. The nodes $i$ and $j$ are then deleted from the initial *list* of memory nodes and a new node with the characteristics $<W'',B'',P''_r,P''_w,C'',D''>$ is added. Steps 2 to 18 are repeated till there is only one memory node in the *list*, or there is no further cost reduction.

## 8 Experimental results

We have implemented the algorithm for component and memory selection, pipelining, and memory generation using C on a SUN SPARC 5 workstation. The first step of the algorithm (*Enumerate_All_Memories*) has a complexity of $O(N_m C_m)$, where $N_m$ is the number of memory vertices in the $\mathcal{DFG}$, and $C_m$ is the number of memory components in the library. The core selection and pipelining algorithm (steps 2 to 13 in Figure 8) has a complexity of $O(N^2 C)$ where $N$ is the total number of vertices in the $\mathcal{DFG}$, and $C$ is the maximum number of implementations of any operator type in the component library. Finally, the complexity of the memory optimization phase is $O(N_m^3 C_m)$.

TABLE 1
MODIFIED DTAS COMPONENT LIBRARY

| Component Type | Component Name | Delay. ($ns$) | Cost (gates) |
|---|---|---|---|
| $\star$ | Mpy1 | 57.97 | 2368 |
| $\star$ | Mpy2 | 44.21 | 2400 |
| $\star$ | Mpy3 | 36.21 | 2600 |
| $\star$ | Mpy4 | 32.98 | 2710 |
| $\star$ | Mpy5 | 28.57 | 2978 |
| $\star$ | Mpy6 | 25.00 | 3500 |
| $\star$ | Mpy7 | 22.50 | 4000 |
| $\star$ | Mpy8 | 20.50 | 4500 |
| +/- | Add1/Sub1 | 25.80 | 62 |
| +/- | Add2/Sub2 | 20.00 | 125 |
| +/- | Add3/Sub3 | 13.50 | 187 |
| +/- | Add4/Sub4 | 10.00 | 250 |
| +/- | Add5/Sub5 | 5.50 | 375 |

In all our experiments we have used a modified version of the DTAS library [9] shown in Table 1 for multiplier and adder/subtractor components. Component cost is in terms of the number of 2-input NAND gates, while delay is in $ns$. The memory components in the library were patterned after the Toshiba gate array memories [1]. The area/bit (gates) and access delay ($ns$) values of four memory types (MEM_A, MEM_B, MEM_C and MEM_D) are shown in Figure 12. Each memory type contains several memories of different bitwidths (ranging from 4 to 32) and a different number of words (ranging from 4 to 1024). In general, for all memory types, increasing number of words and bits leads to a decrease in the area/bit and an increase in the access delay per word, as indicated in the graphs. (The curves are annotated with a few memory sizes to give a feel of the area/bit and delay variation with size).

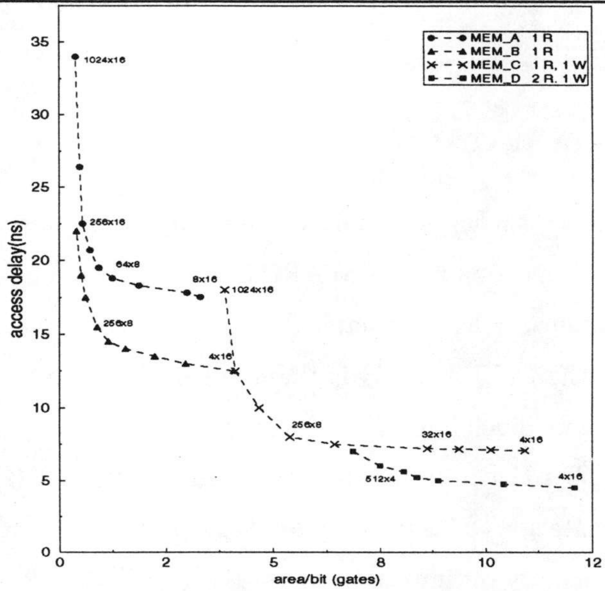


Figure 12: Area/bit vs. access delay of memories used in all our experiments.

We conducted three types of experiments. The first set of experiments demonstrates the importance of allowing memories with different characteristics (in terms of bitwidth, number of words, number of ports, delay and cost), in a design. The second experiment demonstrates the importance of memory optimization by comparing designs obtained with and without the optimization. Finally, the third experiment demonstrates the range of designs obtained by varying the component and memory selection and the pipelining.

## 8.1 Experiment #1: Memory selection

The aim of this experiment is to get a quantitative measure of the importance of designing a DSP system with memories of varying bitwidths, ports, access delays etc. We conducted this experiment for three examples, a differential heat release computation (DHRC) [8], that had 4 arrays of size $489 \times 16$, the Daubechues 4-coefficient wavelet filter (DWF) [13] with arrays of size 256 × 16, 128 × 16 and 4 × 16, and the Kalman filter [10] (KALMAN) with arrays of size 256 × 16 and 16 × 16.

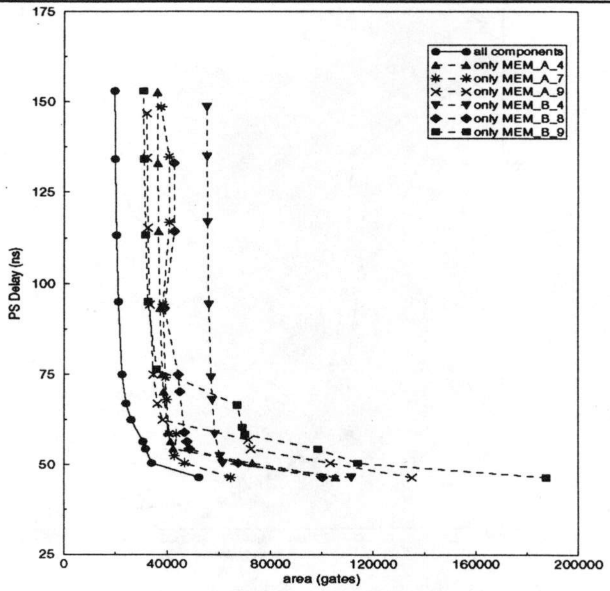


Figure 13: Comparing Area vs. *PS Delay* of KALMAN filter designs obtained by using all memory types vs. only one memory type.

For each of the examples, we first executed our algorithm for a range of *PS delay* constraints using the complete component library, that is, all the adders and multipliers in Table 1 and all the memory components shown in Figure 12. We then obtained a list of the different memory components that were used in the designs, and for each of the memory components, we conducted the experiment again, this time with only that component instead of the entire set of memory components. Note, that the set of adders and multipliers remained the same through all the experiments.

Figure 13 shows the results obtained for the KALMAN example. Designs shown along the solid curve were those obtained by using the complete set of memory components. These designs contained a mix of six types of memory components: MEM_A_4 of size 32 × 16, MEM_A_7 of size 256 × 8, MEM_A_9 of size 1024 × 16, MEM_B_4 of size 32 × 8, MEM_B_8 of size 512 × 8, and MEM_B_9 of size 1024 × 16. Designs shown along the dashed curves were obtained by using only one of the six memory types; hence, there are six dashed curves, one for each memory type. For a given *PS delay* value, these designs are much higher in area than the design obtained by using a mix of memory types. For instance, the area of designs with MEM_B_9 are approximately 50% higher in area at large *PS delay* (75 - 150 *ns*) values and as much as 250% higher at lower *PS delays* (45 - 75 *ns*). Of all the six memory types, the best designs are obtained using MEM_A_7; they are approximately 50% higher in area

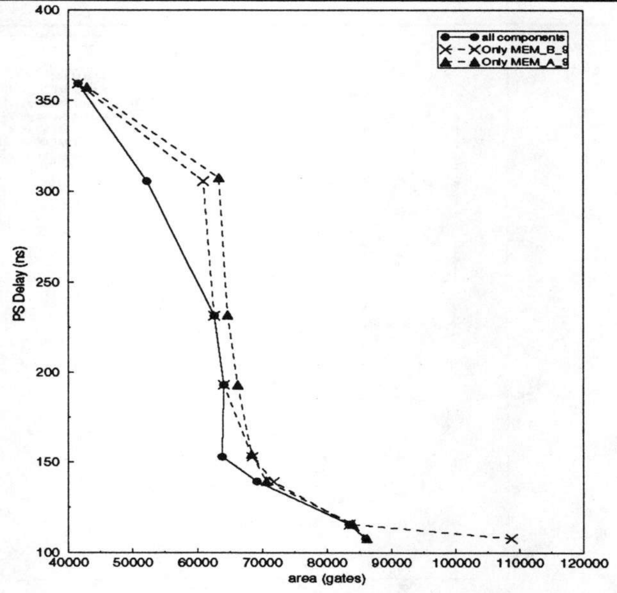


Figure 14: Comparing Area vs. *PS Delay* of DHRC designs obtained by using all memory types vs. only one memory type.

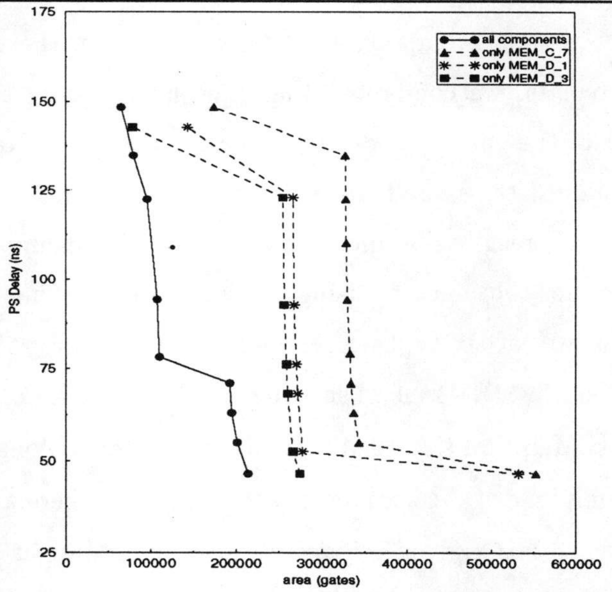


Figure 15: Comparing Area vs. *PS Delay* of DWF designs obtained by using all memory types vs. only one memory type.

over the entire range of *PS delay* values.

Similar experiments were conducted for DHRC (Figure 14) and DWF (Figure 15). Once again, using the complete component library with all the memory components we obtained designs with much lower costs than those obtained by using just one type of memory component. For the DHRC example, only two memory types, MEM_A_9 and MEM_B_9, were used from the complete memory library. Designs with only one of these memories were about 20% higher in area, than designs with both memory types. This difference in design areas between using only one memory type and a mixture of memory types was more significant for the DWF example. Designs with single memory components were between 75% to 175% higher in area than designs obtained with the complete memory library.

In all three examples, we note that the set of designs obtained by using the complete library with all memory components have a lower area than designs with only one memory type. In some examples, this difference in area is as significant as 100%, and for all the examples we ran, no single memory component could consistently give low area designs over all *PS delay* constraints. This experiment indicates the impact of memory selection in obtaining cost-efficient designs.

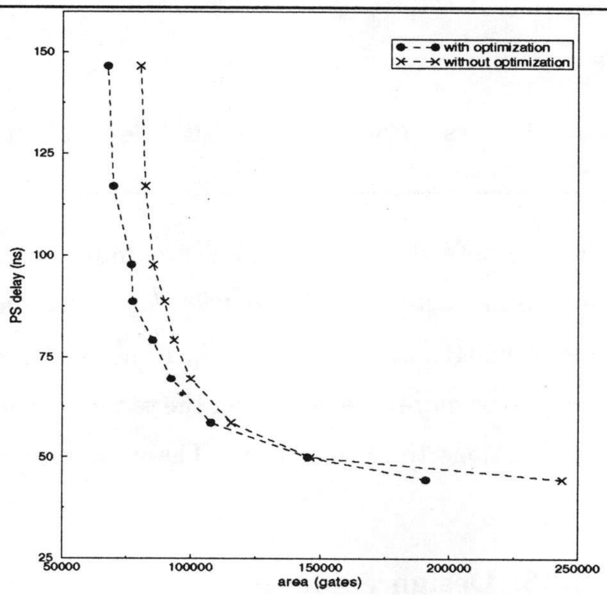


Figure 16: Comparing Area vs. *PS delay* of 4 × 1 beamformer designs obtained with and without memory optimization.

## 8.2 Experiment #2: Memory optimization

This experiment quantifies the area improvement obtained by optimizing the memory after selection and pipelining (step 14 of Figure 8). We conducted this experiment for a

$4 \times 1$ beamformer [3] (Figure 16) containing four 8th-order FIR filters, and for the DHRC example (Figure 17) introduced earlier. For both examples, we compare designs obtained with and without the optimization. At *PS delays* below 130 *ns* the DHRC example did not benefit from optimization but at higher delay constraints the optimized designs are as much as 40% lower in area. Similarly, for the beamformer, at *PS delay* values above 150 *ns*, the optimized designs are, on average, 20% lower in area than the non-optimized designs.

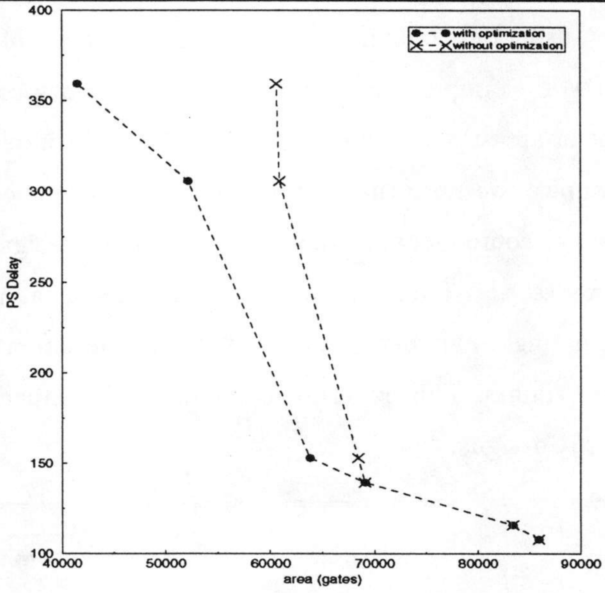


Figure 17: Comparing Area vs. *PS delay* of DHRC designs obtained with and without memory optimization.

For both examples, we note that optimization was more beneficial at larger *PS delay* constraints. This is due to the fact that at lower *PS delay* values when the time constraint is very stringent, there is very little "room" for optimization. At larger *PS delay* values, there is a larger possibility of two or more arrays sharing the same memory and, in addition, there are more memory organizations to choose from. These factors result in better optimized designs.

## 8.3 Experiment #3: Design exploration

This experiment gives a quantitative measure of the design space explored by varying the component selection, memory selection and pipelining in a design. Once again, we conducted this experiment for a $4 \times 1$ beamformer (Figure 18) and for the DHRC example (Figure 19). For the beamformer example, we note that just by varying the component and memory selection alone we obtain designs that vary in delay from 25 to 150 *ns* and in area from 190,000 gates to 60,000 gates. Similarly, for the DHRC example (Figure 19), delay

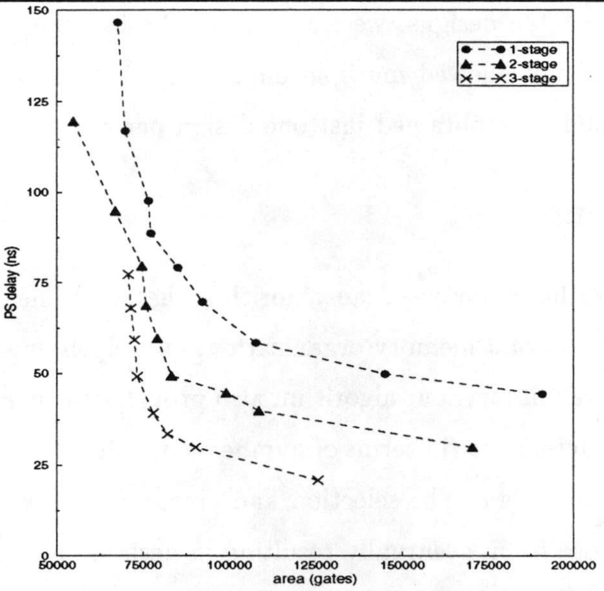


Figure 18: Area vs. *PS delay* of $4 \times 1$ beamformer designs obtained by varying component and memory selection and pipelining.

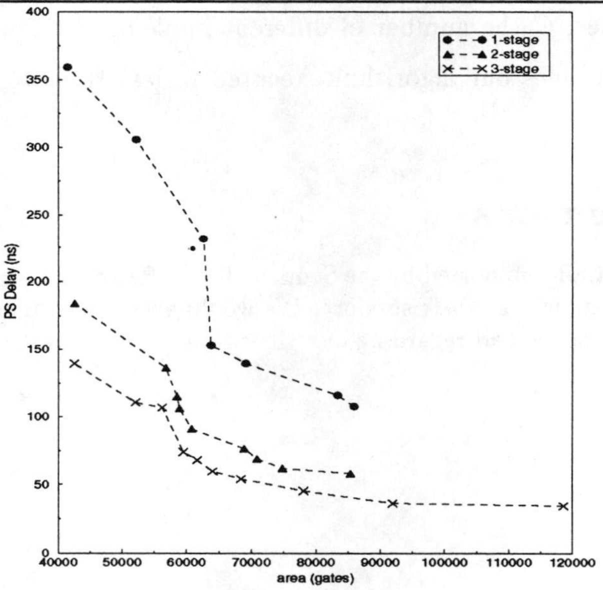


Figure 19: Area vs. *PS delay* of DHRC designs obtained by varying component and memory selection and pipelining.

varies from 40 to 350 *ns* and area from 120,000 gates to 42,000 gates.

Note, that the extent of this variation is due both to the selection and pipelining features. Had we not pipelined the designs, we would have obtained higher costs for the same *PS delay*, and if we had not allowed multiple implementations of operators and memories in our designs, we would have obtained just one design per curve.

## 9 Conclusions

In this report, we have presented an algorithm that combines component selection for operators, the selection of a memory organization, and pipelining. In addition to selecting memories from a given library, our algorithm also provides the capability of generating new memories with characteristics (in terms of number of words, bits, ports or access delay) that are well suited to the design. The selection and pipelining tasks thus have a larger variety of memories to choose from, eventually resulting in designs that are more cost-efficient.

Experiments conducted on several examples indicate the importance of designing a DSP system with memories of several different types. The designs obtained by using a library of different memory components, were consistently lower in area than the designs with memories of only one type. In fact, in some examples, the best designs obtainable by using single memory types had 75% higher area than those obtained with multiple memory types.

Additionally, our algorithm has a polynomial time complexity of $O(N^2C)$, where $N$ is the number of nodes, $C$ the number of different implementations of any component type, and for all our examples our algorithm executed in less than 45 seconds, on a *SPARC 5* workstation.

## Acknowledgements

This work was partially supported by the Semiconducter Research Corporation grant #93-DJ-146, and we gratefully acknowledge their support. We would also like to thank Hsiao-Ping Juan for the many useful discussions we had regarding our algorithm.

## References

[1] *Toshiba ASIC Gate Array Library: TC140G/14L Series, Megacell, Megafunction.* Toshiba Corporation, 1990.

[2] I. Ahmed and C. Chen. Post processor for data path synthesis using multiport memories. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 276–280, 1991.

[3] Smita Bakshi and Daniel D. Gajski. Design space exploration for the beamformer system. Technical Report 93-34, Dept. of Information and Computer Science, University of California, Irvine, 1993.

[4] Smita Bakshi and Daniel D. Gajski. A component selection algorithm for high-performance pipelines. In *Proceedings of EURO-DAC*, pages 400–405, 1994.

[5] M. Balakrishnan and A. Majumdar et al. Allocation of multiport memories in data path synthesis. *IEEE Transactions on Computer Aided Design*, 7(4):536–540, April 1988.

[6] F. Balasa, F. Cathoor, and Hugo De Man. Exact evaluation of memory size for multi-dimensional signal processing systems. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 669–672, 1993.

[7] F. Balasa, F. Cathoor, and Hugo De Man. Datafow-driven memory allocation for multi-dimensional signal processing systems. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 31–34, 1994.

[8] F. Catthoor and Lars Svensson. *Application-Driven Architecture Synthesis.* Kluwer Academic Publishers, P.O. Box 17, 3300 AA Dordrech, The Netherlands, 1993.

[9] N. D. Dutt and J. R. Kipps. Bridging high-level synthesis to RTL technology libraries. In *Proceedings of the 28th Design Automation Conference*, 1991.

[10] N. D. Dutt and C. Ramachandran. Benchmarks for the 1992 High-Level Synthesis workshop. Technical Report 92-107, Dept. of Information and Computer Science, University of California, Irvine, 1992.

[11] D. Karchmer and J. Rose. Definition and solution of the memory packing problem for field-programmable systems. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 20–26, 1994.

[12] F.J. Kurdahi and A. Parker. REAL: a program for register allocation. In *Proceedings of the 24th Design Automation Conference*, 1987.

[13] J. S. Lim. *Two-dimensional image and signal processing.* Prentice Hall Signal Processign Series, 1990.

[14] P. Lippens and J. van Meerbergen et. al. Memory synthesis for high-speed DSP applications. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1991.

[15] P. Lippens and J. van Meerbergen et. al. Allocation of multiport memories for hierarchical data streams. In *Proceedings of the IEEE International Conference on Computer Aided Design*, 1993.

[16] P. Marwedal. The MIMOLA design system: Tools for the design of digital processors. In *Proceedings of the 21th Design Automation Conference*, pages 587–593, 1984.

[17] S. Narayan, F. Vahid, and D.D. Gajski. System specification and synthesis with the speccharts language. In *Proceedings of the IEEE International Conference on Computer Aided Design*, 1991.

[18] S. Note, F. Catthoor, G. Goossens, and H.J. De Man. Combined hardware selection and pipelining in high-performance data-path design. *IEEE Transactions on Computer Aided Design*, II(4):413–423, April 1992.

[19] L. Ramachandran, D. D. Gajski, and V. Chaiyakul. An algorithm for array variable clustering. In *Proceedings of the European Design Automation Conference*, pages 92–95, 1993.

[20] L. Stok. Interconnect optimisation during data path allocation. In *Proceedings of the European Design Automation Conference*, 1990.

[21] Alfred B. Thordarson. Comparison of manual and automatic behavioral synthesis on MPEG-algorithm. Master's thesis, University of California, Irvine, 1995.

[22] C. Tseng and D. Siewiorek. Automated synthesis of datapaths in digital systems. *IEEE Transactions on Computer Aided Design*, 5(3):379–395, July 1986.

[23] J. van Meerbergen and P. Lippens et al. A design strategy for high-throughput applications. In *VLSI Signal Processing, edited by Kung Yao et al.*, pages 150–165, 1992.