Towards Synthesizing Memory Architecture for Applications - Specific Systems* [1]

Pravil Gupta and Alice C. Parker

CENG Technical Report 92-05

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

[1]This work was supported by the Defense Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation under contract No. JFBI90092. The views and conclusions considered in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Defense Advanced Research Projects Agency or the U.S. Government.

1 Introduction

Data paths for application-specific designs may process enormous amounts of real-time data. Such data must be stored in memory structures which are cost-effective and allow access to the data as required by the data paths. Although cost of storage per bit is very low, the total cost of memory may dominate the overall system cost due to the huge memory requirements of today's complex systems. The memory synthesis problem is also important in applications where the data transfer rate is very high (e.g. real-time applications such as personal communications). Data path synthesis procedures must take into account the design of the memory hierarchy which is companion to the data path; the design of the data paths and memory hierarchies must somehow be coordinated. Our focus is on the design of memory architectures for application-specific integrated circuits (ASICs).

2 Related 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 memory access pattern varies from application to application. On the other hand, our tools would 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 reference [7] which was performed a number of years ago.

The MIMOLA system is the first system to make tradeoffs in the use of multi-port memories [6]. Balakrishnan *et al.* [1] presented an approach to use multi-port memories to implement single isolated registers. This approach merges these registers into a homogeneous group of modules. However, we wish to consider various types of storage modules with different characteristics, e.g. we can have n_1 m-port modules, n_2 p-port modules and so on.

Grant *et al.* suggested an approach to group the memory requirements of various operators for more efficient designs in [3]. In an earlier study the same group studied a different aspect of memory synthesis, address generation [4]. Stok [12] optimizes register files during the synthesis process. All these efforts concentrated on separate aspects of memory synthesis. An overall tradeoff approach is yet to be taken.

Recently Lippens *et al.* from the Philips Research Labs [5], in PHIDEO, described techniques to perform automatic memory allocation and address allocation for high speed applications. They synthesize memory after the design of arithmetic units and after scheduling. They distribute the memory among parallel memories consisting of 1 and 2 port RAMs. They do not distinguish between background and foreground memory. In IMEC's CATHEDRAL-II efficient storage schemes and memory access techniques were implemented by De Man *et al.* [14]. They compile multi-dimensional data structures into distributed dual-port register files and single-port SRAMs.

3 Memory Architecture Synthesis Research

3.1 Memory design examples

The input/output interface for systems presents a very important memory design problem, commonly found in image processing applications and many other real-time systems where the data transfer rate is very high and the amount of data to be processed is enormous.

A simple example of I/O storage requirements involves data arriving and being processed at a fixed rate e.g., the LSI Logic 40 MHz Reed-Soloman Encoder-Decoder [13] requires a delay of $n + 91$ cycles in the decoder block. This is achieved simply by using a delay RAM.

In another type of memory design situation we have an internal pool of data which is being read or written randomly: this occurs in digital TV. Each frame buffer might require 2 Mbits of memory [8] with processing rate as high as 100-500 MOPS. Here the typical sampling rate is 14.3 MHz and the transmission bit rate, after High Order-Differential Pulse Code Modulation, is 45Mb/s. To get an idea of the storage access requirements, the number of multiplications required for each picture element, can be as high as 25. A DCT (Digital Cosine Transformation)-based motion-compensation interframe encoder composed of a motion detector, DCT, inverse DCT and an inner loop filter must perform about one billion multiplications per second. Such real-time video signal processors require large storage devices for storing intermediate results as well as inputs and final results. Another noticeable issue is the frame buffer, where all the memory elements are not modified at the same rate; only the ones which change are to be written. In other words, the data points have variable life times. Such a real-time system might require another layer of data storage between the processor and main storage depending on the access time and data transfer rate.

3.2 Characterization of design strategies

Some design concepts and strategies will be briefly reviewed. Note that wiring space and delays are important parameters and can not be ignored in producing near optimal designs.

RAMs are particularly better suited for handling enormous amounts of data, for accessing random addresses from a pool of data or when the lifetime of data is long. Address generation hardware in this case is complex and would need to be synthesized. Multiport memories can be more effective than isolated registers when the number of ports required to meet performance constraints is less than the number of ports available on the memory module [1]. FIFO, queues, and register files are useful when the access pattern is regular and fixed. Address generation in such cases is quite trivial.

A real-time system, like the serial video processor [2] which takes in the data serially and processes it in parallel, illustrates an interesting memory tradeoff. It has 1024 one-bit processing elements working in SIMD mode. The memory system consists of a 40-bit $\times$ 1024-word dual-ported memory, two 128-bit $\times$1024 register files, 4 working registers and one 24-bit $\times$ 1024-word dual-ported memory.

In data paths requiring temporary storage, a standard technique is to group the memory requirements of various operators for more efficient designs. Automating this has been presented in [3]. Individual registers for storing outputs of operators may be a good idea for small systems, but for larger systems, using register files may result in better designs. For even larger memory requirements, RAMs may be more area efficient.

3.3 Our research approach

Using our definition, the memory in a system includes all the storage modules. Based on the function and the part they are interacting with, the memory architecture can be divided into different sub-architectures, which may have similar structures but may vary in functionality. Each sub-architecture may consist of various storage modules and devices like registers, register files, and RAMs. and has the basic structure shown in the Figure 1. Though this categorization is quite similar to the one used for general purpose computer architectures, their data paths and control are clearly different from application-specific designs, and hence the memory design problem differs also. A brief description of these sub-parts follows.

Main memory : The main memory provides large, cheap storage space, the same as in a general purpose computer. It could be **shared** or **distributed** between various parts of the data path or between multiple data paths in a multiprocessor system. We chose to

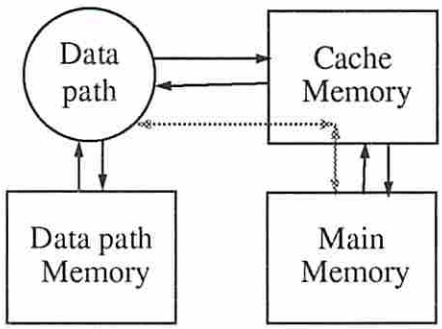


Figure 1: Targeted system architecture

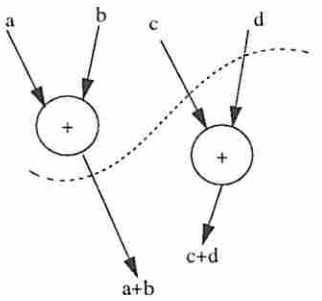
implement distributed memory for the following reasons: (i) the performance requirements are generally quite high for target systems. Distributed memory is faster and more efficient in access as the data values can be accessed directly by the processing elements, and (ii) we know the access pattern of the data elements. So, we can efficiently distribute the memory for different data paths (processors), saving us unnecessary routing and switching.

Our main memory interacts with the I/O and the cache memory. If main memory has to interact with the data path, in our architectural style the interaction will be via cache memory. Main memory has its own address generator and controller and is interfaced to the system via buses along with bus drivers.

Cache memory : Cache memory is defined as memory which interacts with the processors directly and makes the appropriate data values available at every step. It also has its own controller. The data is transferred to the cache from the main memory or I/O before it is processed and then the cache memory provides the required distribution system and ports. Similarly, after processing, the data is stored in this memory before being transferred to the main memory or I/O subsystem. At the beginning of each step the required data is loaded onto the cache memory output ports. Cache memory is also used to store data which will be used in future steps. In case the main memory or I/O subsystem can interface to the data path directly, cache memory is made transparent by using wires to replace modules.

Data path memory: Data path memory consists of all the intermediate storage elements which are used to store different values in the data path. It consists of registers and single or multiport register files and may even use RAMs in case of huge storage requirements. It is distributed throughout the data path.

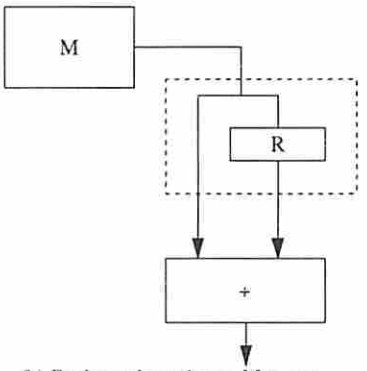
Since, data path memory modules are closely connected to various parts of the system, the interconnections area can be significant. Therefore, interconnection cost is an additional parameter we have to consider while synthesizing this part of the memory.



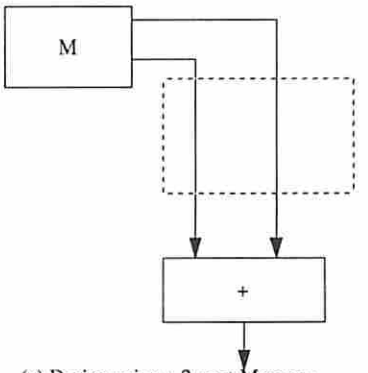
(a) A data flow graph.

Step 1	Read a from M, Store in R	Read a, b from M, Add a+b
Step 2	Read b from M, Add a+b	Read c, d from M, Add c+d
Step 3	Read c from M, Store in R	
Step 4	Read d from M, Add c+d	

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



(b) Design using a 1-port Memory.



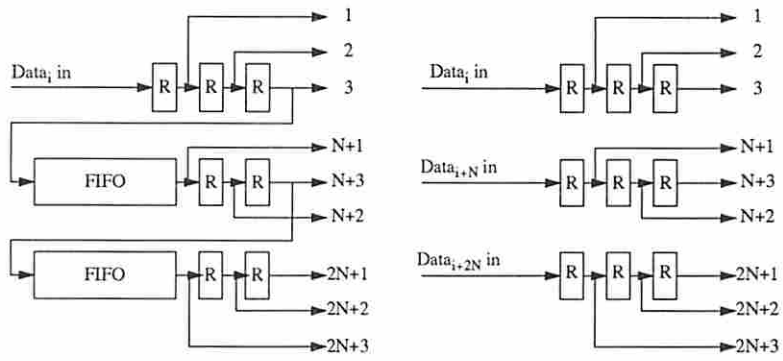
(c) Design using a 2-port Memory.

Figure 2: Number of ports vs. time

3.4 Memory design tradeoffs

In this section, we discuss selected area-time tradeoffs possible in memory synthesis which we have implemented in our prototype software. Time is minimized by optimizing the number of clock cycles needed for data management and by scheduling memory accesses so that the processing is not delayed. We will achieve this by providing the appropriate number of ports and storage size. The sizes of the modules as well as the number of ports can be traded off with the number of cycles needed for data transfer as described below.

Number of ports vs. number of memory cycles Required data transfers can be achieved by having more words transferred every clock cycle (more ports) or using more cycles to transfer those words. This is a trivial tradeoff between space and time multiplexing. In the example (Figure 2), two implementations are shown, the first with a 1-port memory and the second with a 2-port memory. The second design requires two steps while the first takes four steps for execution. To reduce the complexity of the problem we deal with the above two tradeoffs separately. Since the number of ports does not vary widely for practical implementations, we iterate over the number of ports, optimizing the size for each choice, while meeting the performance constraints.



(a) Design with one port. (b) Design with three ports.

Figure 3: Number of ports vs. size of memory

Number of ports vs. size of the memory

In the example (Figure 3) nine data elements are being accessed every clock cycle with a fixed interval among them. In design (3 a) we have only one output port on the main memory and one input port on the cache to access the data and so we use FIFOs to save the intermediate data for future use. In design (3 b) we can access data through three ports on main memory and cache thus increasing the input bandwidth. Therefore, instead of saving data in temporary registers for future use we access it repeatedly whenever we need it.

3.5 Problem approach

The memory synthesis problem is, like many other design automation problems, an extremely difficult one. To make the problem more manageable we restrict it by assuming a certain ordering to design tasks and ignoring some aspects at present.

To reduce the complexity of the problem we synthesize the above-mentioned different parts, (i) main memory, (ii) cache memory, and (iii) data path memory, separately. Dividing the storage architecture also helps us determine the order in which we would like to consider various data demands. We generate the data path schedule and therefore we know the data requirements of the data path in each step. We then synthesize the cache memory, which interacts with the data path directly. After that we consider main memory synthesis, followed by data path memory synthesis. The reasons for this order will be given in the presentation.

Like data path synthesis, each part of the memory is synthesized by decomposing the problem into these two basic steps: **data transfer scheduling** and **module allocation**, as other researchers have suggested for similar problems. Our focus in this summary is on the data transfer scheduling for cache design.

We assume two-phase clocking for our target system. For the main memory, the data

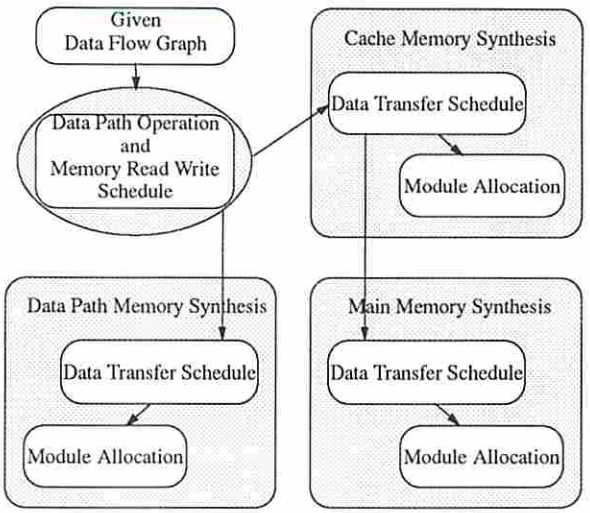


Figure 4: Memory synthesis.

is written in first phase and read in the second phase. For the cache and data path memory, data is read in the first phase and written in the second phase. The data path processes the data in the first phase and writes it back into the cache in the second phase. A simple example illustrating our approach is shown in Figure 5.

We assume the following modules in the module library: register, single-port/multi-port register file, FIFO, and single-port/multi-port on-chip RAM. In case of multi-port modules, each write port has an address bus, an input data bus, and a write enable signal. Similarly, each read port has an address bus, data out bus (standard or tristate), and an output enable bus. Simultaneous reads from the same address are legal.

4 Combined data path-memory synthesis

This step schedules the data path with focus on memory architecture synthesis. During this step we take the memory architecture features into account and make sure that in the following steps when we are actually doing the memory synthesis, the schedule supports the chosen memory architecture.

The problem which we address in this step is: given the behavior of an application-specific system in the form of a data flow graph, the module library with hardware constraints, and an optional input data timing constraint, the scheduling program should schedule the data path meeting all the hardware constraints. The hardware constraints include the number of various functional units, the number of read and write ports on the cache memory and the number of read ports on the main memory available to us in each step.

We insert *read* or *write* nodes in the data flow graph, whenever an input is read from

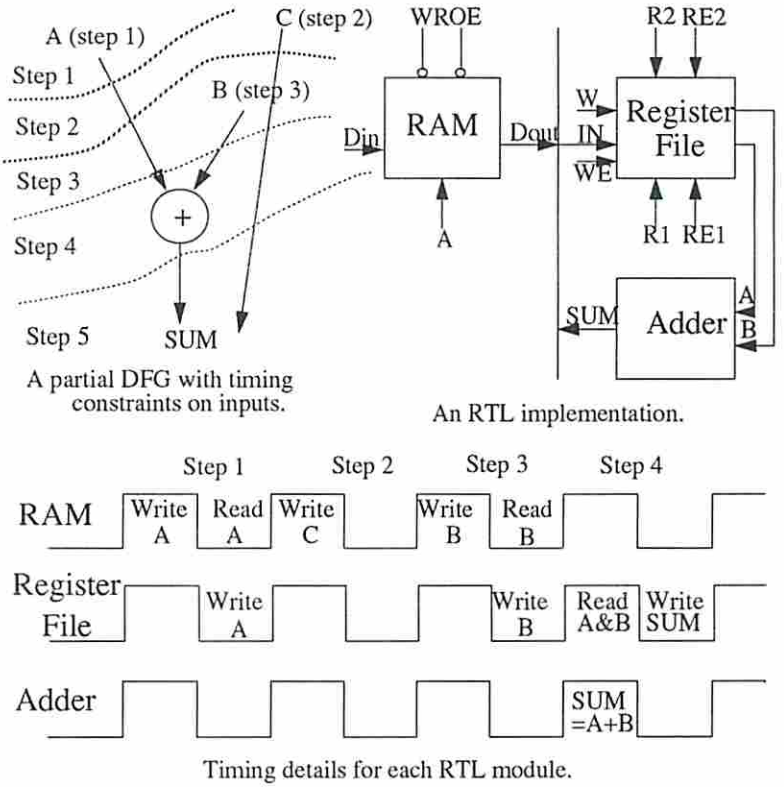


Figure 5: An example illustrating 2-phase clocking.

outside or an output is produced and then treat them as functional operators to perform *data transfer*. We use freedom-based scheduling [9] for the prototype but any other scheduling technique which performs scheduling under hardware constraints can be used. We also take into account the timing constraints on the input data values, whenever imposed by the external world. Distribution graphs of the *read (write)* operations and the data path operations are used to assign them probabilistically to the most suitable control step , as in force-directed scheduling [11]. We also make sure that before an input is to be consumed, it can be prefetched into the cache from the main memory. This is done by scheduling the input read in such a way that there is an available slot to transfer that input to the cache before it is consumed. We use the information on the number of read ports on the main memory to compute the bandwidth available for data transfer between the main memory and the cache. The actual data transfer is scheduled in the next step.

The output of this software consists of an operation schedule, input read and output write schedule, and intermediate variables read/write information.

At present the prototype produces non-pipelined designs without conditional branches.

5 Data transfer scheduling for the cache memory

There is a finite interval from the production of a data value to its consumption; it can be transferred at any time step to the functional module for processing and stored locally. We also have to store a value after it is produced, and before it is required for further processing. Both storing a value and making it available for processing require determination of resources - the storage module size and ports for providing it to the operators.

Our second step in the synthesis process is to schedule cache memory accesses from the schedule obtained in the previous step in section 4, with the objective of minimizing the memory size while meeting timing constraints. The problem is described as follows: given the number of read and write ports on the cache in each step, the time interval during which the data is available, and the data requirement as determined in the data path scheduling step; we determine a data transfer schedule, read time and write time for all the data points, such that the size of the memory is minimized, and the cache provides the data to the data path as they are required, with the given number of ports.

Our scheduling problem is analogous to the scheduling problem in data path synthesis. The schedule length in data path scheduling is quite similar to the schedule length (the time required to transfer the data) here. The other parameter in data path scheduling is the width of the schedule, which reflects the amount of hardware being used. This can be associated with the number of ports in our problem. In our case, the ports are the resources, executing the operation *data transfer*. However, here the operation is single-cycle and is either read, write or both, unlike data path scheduling where we have different types of modules performing various operations. Besides this, we are trying to optimize the size of the storage module, which depends on the sequence of the read/write operations. In other words we are trying to balance the number of reads and writes which occur in each time step.

We use *list scheduling*, with a redefinition of the objective function, to solve our problem. We maintain a list of all the data values to be scheduled. An *urgency factor* is associated with each value which determines the necessity of that value transfer to be scheduled in a particular step.

Updating the data list is based on the *urgency factor* associated with each data value. We try to delay the transfer as much as possible because we know that keeping the value in the memory will contribute to the size of the memory. We want to minimize the presence of data in the buffer and try to write it into the buffer as late as possible. From that point of view it is a greedy algorithm. For such an approach it is useful to process the schedule

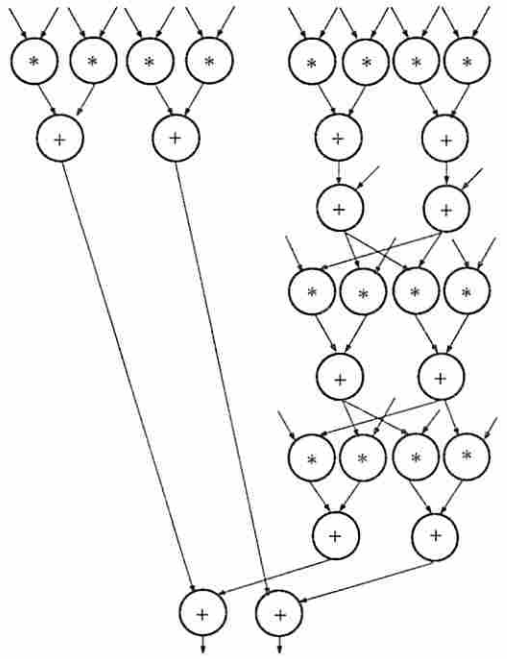


Figure 6: AR lattice filter.

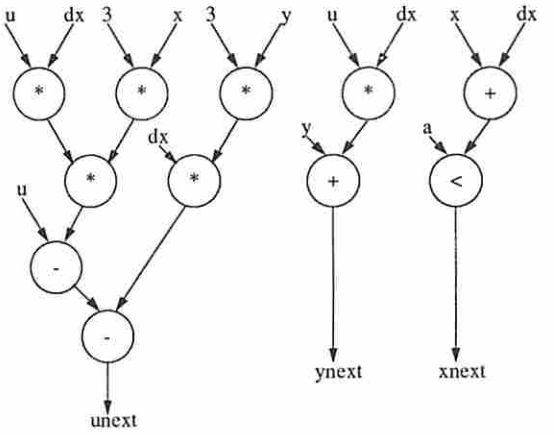


Figure 7: Second-order differential equation example.

backwards. The urgency factor for these values will depend on the following factors: (i) The first factor is the number of steps remaining to prefetch the data into cache. The smaller this number is, the more urgent it is to fetch the data. (ii) If the values is going to be required again by the data path then we can lower its priority because all we need to do is to keep it stored in the buffer.

The output of this program consists of the complete data access schedule for the cache memory. From the access pattern we compute the size of the cache memory, which is the maximum number of data values we need to store in any step.

6 Experimental results

We applied our approach to two representative examples: an AR lattice filter element [10], shown in Figure 6 and a second-order differential equation example [11], see Figure 7.

In these examples we assume that the input data is brought into an on-chip RAM (main memory) and is ready for processing. However, our programs can be given timing constraints on the inputs imposed by the external world. We design the cache memory and the data path along with the data access schedule for the cache memory. The main memory access schedule can be determined by the write schedule for the cache and the input timing from outside. The results are summarized in table 1.

Note that, in examples 2 and 4 the cache sizes are zero. It implies that the data can be read directly from the main memory in each step and we don't need to store any values in the

Example	Input			Output	
	Functional hardware	Bandwidth between cache & data path (No. of R/W ports on cache)	Bandwidth between main mem. & cache (No. of R ports on main mem.)	Number of control steps	Cache size
AR filter	2 adders 2 mults..	2 words/cycle	1 word/cycle	29	4
		2 words/cycle	2 words/cycle	15	0
Sec. order diff. eqn.	1 adder, 1 mult, 1 sub., 1 comp.	2 words/cycle	1 word/cycle	9	3
		2 words/cycle	2 words/cycle	8	0

Table 1: Data path scheduling with cache design.

cache at any time. In examples 1 and 3, since the bandwidth between the main meory and the cache is very restricted, we prefetch the data and store it in cache before it is demanded by the data path. Once it is used we can reuse the memory location if it is not needed again.

In these examples we had a very stringent memory bandwidth constraint. The long execution times are due to the restriction on the number of reads and writes that can be performed in each step. These narrow, but more realistic, bandwidths caused a bottleneck during the execution. A quick analysis of the resource utilization shows us that the read and write ports are heavily active throughout the execution and their scarcity is causing underutilization of other resources. We can also see that in example 1, one-word bandwidth between the main memory and the cache further delayed the execution time while in example 3 it did not affect the execution time drastically. The reason is that the number of inputs in the first example is 26 and we need at least that many steps to fetch all the inputs. In the third example the number of different inputs is only 6 and their access is interleaved with the execution of the data flow graph. A more careful study of these schedules shows that in the case of second-order differential equation example a larger bandwidth between the cache and the data path can improve the execution time more effectively. Layout of an example will be shown in the presentation.

References

[1] 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 CAD*, 7(4):536–540, April 1988.

[2] J. Childers, P. Reinecke, H. Miyaguchi, S. Yamamoto, Y. Takahashi, Y. Yaguchi, and

M. Takeyasu. SVP : Serial Video Processor. In *Proc. of the IEEE Custom Integrated Circuits Conf.*, pages 17.3.1–17.3.4, May 1990.

[3] D.M. Grant and P.B. Denyer. Memory, Control and Communication Synthesis for Scheduled Algorithms. In *Proc. of the 27th DAC*, pages 162–167, June 1990.

[4] D.M. Grant, P.B. Denyer, and I. Finlay. Synthesis of Address Generators. In *Proc. of the ICCAD*, pages 116–118, 1989.

[5] 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.

[6] P. Marwedel. The MIMOLA Design System: Detailed Description of the Software System. In *Proc. of the 16th DAC*, pages 59–62, 1979.

[7] 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.

[8] K. Niw, T. Araseki, and T. Nishitani. Digital Signal Processing for Video. *IEEE Circuits and Devices Magazine*, pages 27–33, Jan. 1990.

[9] A. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proceedings of the 23rd DAC*, pages 461–466. IEEE and ACM, July 1986.

[10] 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 DAC*, pp. 530–534, 1991.

[11] 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.

[12] L. Stok. Interconnect Optimization during Datapath Synthesis. In *Fourth International Workshop on High-Level Synthesis*, pages 1–6, October 1989.

[13] P. Tong. A 40-MHz Encoder-Decoder Chip Generated by a Reed-Soloman Code Compiler. In *Proc. of the IEEE Custom Int. Circuits Conf.*, pages 13.5.1–4, May 1990.

[14] J. Vanhoof, I. Bolsens, and H. De Man. Compiling Multi-dimensional Data Streams into Distributed DSP ASIC Memory. In *Proc. of the ICCAD*, pages 272–275, 1991.