HISS: A Prototype Program for Hierarachical Storage Synthesis

Pravil Gupta and Alice Parker

CENG Technical Report 93-07

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

HISS: A Prototype Program for HIerarchical Storage Synthesis.*

Abstract

While synthesizing memory-intensive ASICs, data path synthesis procedures must take into account the design of storage hierarchy. The storage architecture is closely connected to the data path of the system and synthesizing it separately may not result in an efficient solution. HISS is a prototype program which combines storage hierarchy design with data path synthesis. It uses appropriate system parameters in order to coordinate between the synthesis of different sub-architectures of the system and schedules data transfers between them. We synthesized some designs using HISS and have included the results in this paper.

Topic of Interest : 4.4

*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 the 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 can 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 datapath; the design of the data paths and memory hierarchies must somehow be coordinated.

In application-specific systems, knowledge of the application makes it practical to extract the partially-deterministic data access patterns beforehand. We can prefetch the data as it is required.

HISS is a program which designs such data paths and the required storage architectures for application-specific integrated circuits (ASICs).

2 Problem statement

HISS works as follows: given the behavior of an application-specific system in the form of a data flow graph, the module library, hardware constraints, memory bandwidth constraints and input/output timing constraints, HISS produces a data path (processor) operation schedule, reads and writes for the foreground memory *viz.* cache and intermediate variables, and the background memory *viz.* main memory.

We simplify the overall synthesis problem by dividing it into two subproblems: data path structure synthesis with a partial design of storage structure, which is performed first, and storage architecture structure synthesis, where we complete the storage structure. We

further classify the storage architecture in a hierarchical manner consisting of (i) cache, (ii) main memory, and (iii) data path storage. The cache is designed first and then the main-memory synthesis follows. Finally, data-path-memory synthesis is performed. We ensure that each step of the stepwise construction of the system provides enough flexibility for the next step to make its own tradeoffs; at the same time it supports the synthesis of the next subpart. Various global design parameters, like the memory bandwidth and timing constraints are appropriately used to construct the partial design in each step to tie the whole synthesis process together.

Data path design requires scheduling the data flow graph and then binding the operations to the hardware modules, while taking into account cache bandwidth. In order to satisfy read/write port constraints on the storage architecture, we schedule the data path operations and the cache reads and writes simultaneously. Unavailability of the data would result in a processing delay and unnecessary transfers would result in extra bandwidth and possibly an increase in the cache size requirements. Therefore, we provide the data to the data path only when it is required. This can be achieved by scheduling an operation in such a way that the required data is either present in the cache or can be prefetched from the main memory. This feature helps us in obtaining a schedule which will support the cache and main memory design to be done in the subsequent steps.

The four steps in HISS include the synthesis of the memory hierarchy. As mentioned earlier, this part of the system consists of three subparts: (i) cache, (ii) main memory, and (iii) data path storage. Synthesis of each part involves scheduling the data transfer and data value bindings to the physical storage modules. In the second step, we first design the cache as it is, physically and conceptually, closer to the data path than the main memory. The data path schedule obtained in the first step (data path scheduling) determines the data transfer between the data path and the cache. The objective in this second step is to schedule the writes to the cache from the main memory and the writes back into the main memory from the cache such that the cache size is minimized. While doing so the memory bandwidth constraints between the cache and the main memory and the timing constraints on the inputs are checked for violations. In order to minimize the cache size we transfer the

data only if it is required and overwrite it whenever possible. The outcome of this step is a complete data transfer schedule in and out of the cache. The schedule can also help us in our third step, which is the determination of the physical architecture of the cache. For example, if the access is sequential we can use a FIFO or if it is random then we may have to use a register file or a RAM depending on the size. Once the data transfer from main memory to the cache is known, we have the read schedule for the main memory. If there are timing constraints on the input data then even the main memory writes of the input data are scheduled, otherwise we schedule them the way we schedule the cache writes in the fourth step. The more important issue here is to allocate main memory locations to the incoming data efficiently. This may require distributed main memory. The final step, local data path storage design has been researched in the literature quite extensively and is not complete in HISS.

3 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. Therefore, for these machines the memory design is based on probabilistic models. On the other hand, HISS will be used for systems designed for specific applications. In our case, the memory access pattern is not only relatively fixed but also known before hand. This most deterministic access characteristic helps us in being more specific, hence more efficient in our designs. This also makes it feasible to automate the design process. An example of the kind of tradeoff study related to our work is the work by Parker and Nagle [9] which was performed a number of years ago.

Many researchers have talked about the need for fast, large memories for high-performance systems. Multi-port RAMs can provide high throughput as simultaneous access is possible. The MIMOLA system is the first system to make tradeoffs in the use of multi-

port memories [8]. Balakrishnan et.al. [2] presented an approach to use multi-port memories to implement single isolated registers. This approach "packs" these registers into a homogeneous group of modules.

Chen [4] [3] explored the design space for multiport memory synthesis. The memory modules were generated in sequence which resulted in locally optimal solutions and in some cases failed to generate the globally optimal solution of minimizing the number of multiport memories [4]. Later on the work was modified and extended to generate a more globally optimal solution in [3].

Ahmad and Chen [1] use 0-1 integer-linear programming to group intermediate variables in the data path into a minimum number of multiport memories depending on their ports and their access pattern.

Grant et al. suggested an approach to group the memory requirements of various operators such that control and communications may be optimized [5]. They consider single-port memory modules. To optimize the communication network between the functional units and memory, simple heuristics are used, which optimize the write-bus network and read-bus network. They also optimize the controller by optimizing the control bit sequence. In an earlier study the same group studied a different aspect of memory synthesis, address generation [6].

Stok [13] optimizes register files during the synthesis process. It is done by splitting read and write phases of registers and by considering parallel storage and rewrites of values that have to be read several times.

Recently Lippens et al. from the Philips Research Labs [7], 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 assume a limited number of memory types available to them (1 and 2 port RAMs) and so their approach is to distribute the data among parallel memories. They do not distinguish between background and foreground memory.

In IMEC's CATHEDRAL-II, efficient storage schemes and memory access techniques were implemented by De Man et al. [14]. According to them, efficient storage schemes and memory access are as crucial as allocation and scheduling of data paths in DSP ASIC design. They compile multi-dimensional data structures into distributed dual-port register files and single-port SRAMs.

All the above efforts concentrated on separate aspects of memory synthesis. An overall approach which addresses the full memory hierarchy has not been reported on.

4 Our approach

The storage architecture is closely connected to the data path of the system, and isolating its synthesis from data path synthesis may not result in an efficient solution. Therefore, HISS performs a combined data path and storage architecture design.

5 Target system architecture

As mentioned earlier, the target system produced by HISS consists of a data path (processor) and a hierarchy of storage modules.

5.1 Data path

The data path consists of functional hardware to execute the data flow (and control flow) graph which specifies the behavior. The salient attributes, like bitwidth, area, and execution time, of these functional units are specified in the module library by the user. In our model, the data path *per se* does not have any kind of storage capability. All the intermediate variables are stored in data path memory, which is described later.

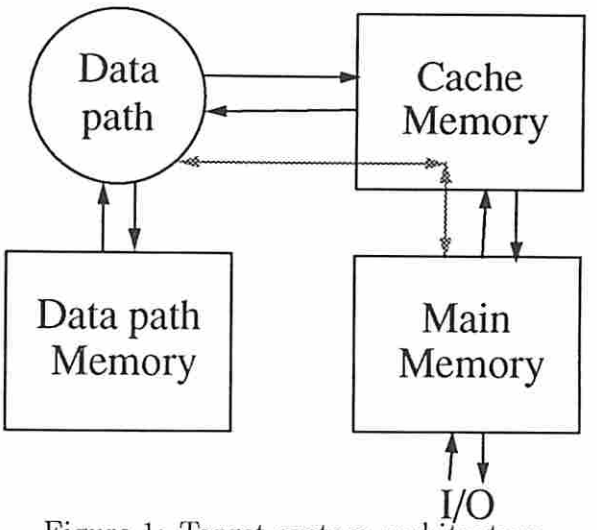

Figure 1: Target system architecture

5.2 Storage architecture

Using our definition, the memory in a system includes all the storage modules. Each storage module interacts with a different part of the system. Based on the function and the part of the data path requiring storage, the memory architecture can be divided into different sub-architectures, cache, main memory and data path memory which may have similar structures but may vary in functionality. The overall architecture of the system may be fixed but each sub-architecture is determined by HISS and may consist of various storage modules and devices like registers, register files, and RAMs. HISS designs three major memory sub-architectures in each design, based on their functionality in the system, as shown in Figure 1.

Though this categorization is quite similar to the one used for general purpose computer architectures, their data path and control architectures are clearly different from application-specific designs, and hence the memory design problem itself differs due to the special-purpose nature of the hardware being designed.

5.2.1 Cache

A cache is a small, but fast memory that can store a part of the overall memory of the system. From our point of view, cache is that part of memory which interacts with the processors directly and makes the appropriate data values available at every step. It has its own controller. The data is transferred to the cache from the main memory before it is processed and then the cache provides the required distribution system and ports. Similarly, after processing, the data is stored in this memory before being transferred to the main memory. At the beginning of each step the required data is loaded onto the cache output ports. Cache is also used to store data which will be used in future steps. In cases where the main memory can interface to the data path directly, the cache is made transparent by using simple wires to replace modules. As the processor reads data from the cache, the controller adds data to the cache from the main memory. If the operation reads the main memory sequentially, access latency will be transparent to the system.

5.2.2 Main memory

The purpose of the main memory is to provide large and cheap storage space, the same as in a general purpose computer. In our designs, the main memory interacts with the I/O interface and the cache. If main memory must interact with the data path, in our architectural style the interaction will be considered to be via cache, even if the cache degenerates into a simple set of wires, when it is not needed. Main memory has its own address generator and controller and is interfaced to the system via buses along with bus drivers.

5.2.3 Data path memory

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

6 Step One: Data Path Scheduling Combined with Memory Synthesis

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

The function which HISS performs 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, 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.

HISS inserts *read* or *write* nodes appropriately into the data flow graph, whenever an input is read from outside or an output is produced, and then treats them as functional operators to perform *data transfer*. HISS uses freedom-based scheduling in this first step [10] for the prototype but any other scheduling technique which performs scheduling under hardware constraints can be used. HISS also takes 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 to the operation, as in force-directed scheduling [12]. 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. HISS uses 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 step two. The detailed flowchart of the implemented algorithm is shown in Figure 2.

The output of this part of HISS consists of an operation schedule, input read from the cache and output write to the cache schedule, and read/write information for intermediate variables from/to the data path memory.

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

7 Storage Architecture Synthesis

7.1 Storage module library

A brief description of the memory modules HISS has in the module library is given in this section.

- Register or latch: The simplest storage element. It can store only one word at a time. Its bitwidth is variable.
- Single-port register file: A register file is a collection of registers with addressing hardware. The number of words it can store at a time is variable and so is the bitwidth of these words.
- Multiport-port register file: Generally, there is only one write port and multiple read ports. Each read port has an address bus, an output data bus, and a read enable signal. Similarly, the write port also has an address bus, an input data bus, and a write enable signal. Simultaneous read from the same location is possible but in case of multiple write ports, simultaneous write to the same location is prohibited.
- FIFO: The FIFO is a first-in, first-out "fall-through" memory. Word and bit dimensions are user specified.
- Single port RAM: RAM modules consider here are assumed to be on-chip. Each module has a data input bus, address bus, an output enable, and an write enable. The output

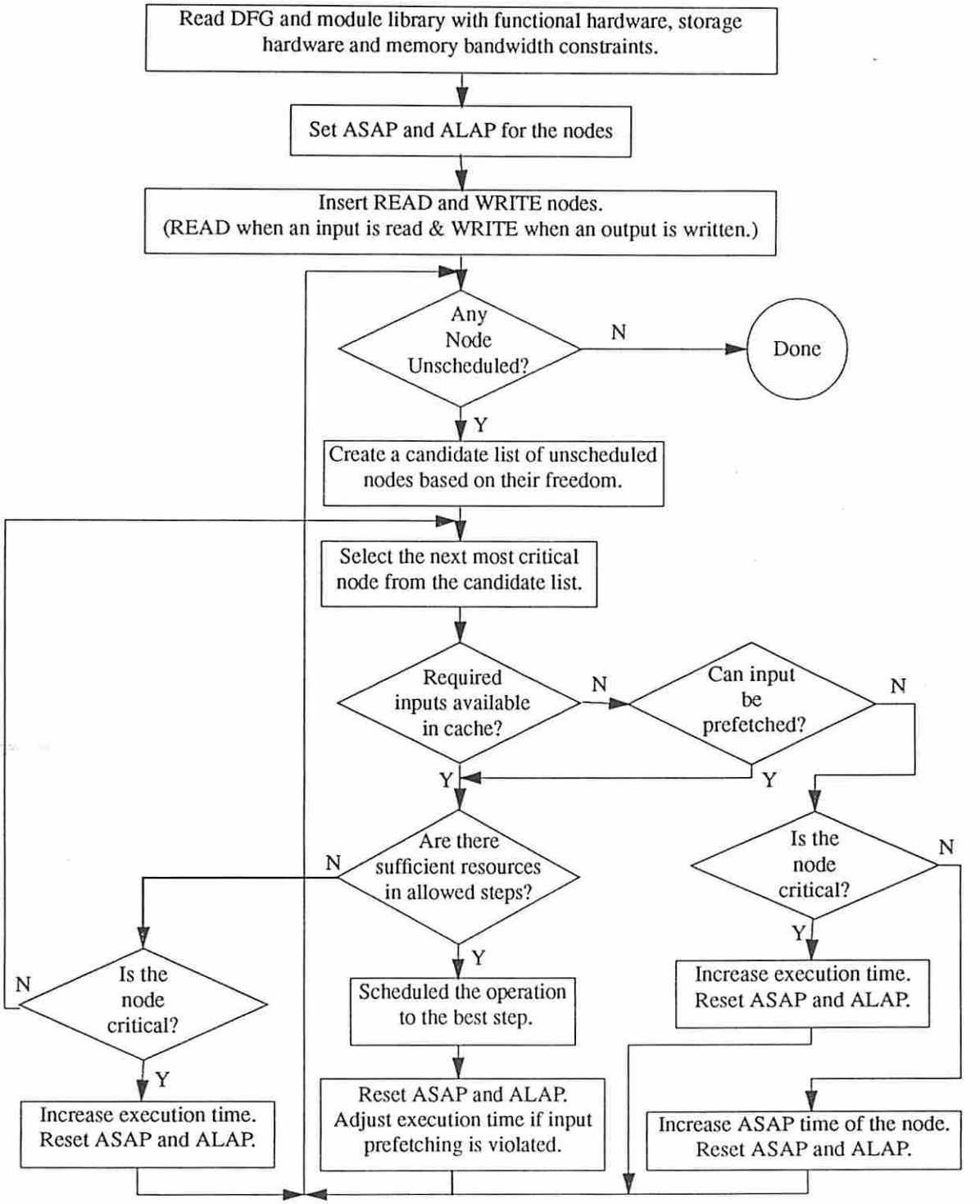

Figure 2: Combined data path and memory scheduling algorithm in HISS.

can be standard or tristated. The bitwidth and the number of words per module are user specified.

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

7.2 Memory design tradeoffs

In this section we discuss three different area-time tradeoffs possible in synthesis of each sub-architecture, including main memory, cache and data path memory.

Time is minimized in HISS by minimizing the number of clock cycles needed for data management and by scheduling memory accesses so that the processing is not delayed. We achieve this by allowing the user to provide the appropriate number of ports to HISS and letting HISS determine the required storage size.

The sizes of the memory modules, the number of ports and the number of cycles needed for data transfer can be traded off with each other as described below.

7.2.1 Memory size vs. number of memory cycles

Sometimes memory modules cannot accept more data because they are too small. Later processing may be delayed because the data transfer may not be complete, resulting in extra clock cycles or time. By allowing more number of clock cycles HISS can often reduce the size of the memory.

7.2.2 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. The user can increase the number of ports allowed on each part of the memory hierarchy, and rerun HISS to reduce the number of memory cycles.

7.2.3 Number of ports vs. size of the memory

From the above two tradeoffs one can deduce the tradeoff between the number of ports and the size of the memory. For example, when a data value is needed again in the future, we may have to save it for future use but this way we will increase the cache size. On the other hand, if we have more ports to read (i.e. larger bandwidth) on the main memory we can access the datum repeatedly whenever we need it thus saving on the size of the memory. Note that we need to increase the number of ports on the cache as well in order to provide a larger bandwidth. Again the user can increase the number of ports available and rerun HISS to reduce the size of the memory.

7.2.4 3-way tradeoff

We have seen that in our problem there are three parameters which vary while making the cost-performance tradeoff, (i) number of ports, (ii) size of the storage architecture, and (iii) execution time. To deal with this 3-way tradeoff, we decided to iterate on the number of ports by repeatedly invoking HISS and for each choice of number of ports let HISS tradeoff between the size and the execution time. This can be done because the number of ports is small and does not vary too much in practical designs. Finally, we can choose the most cost-effective design from these designs.

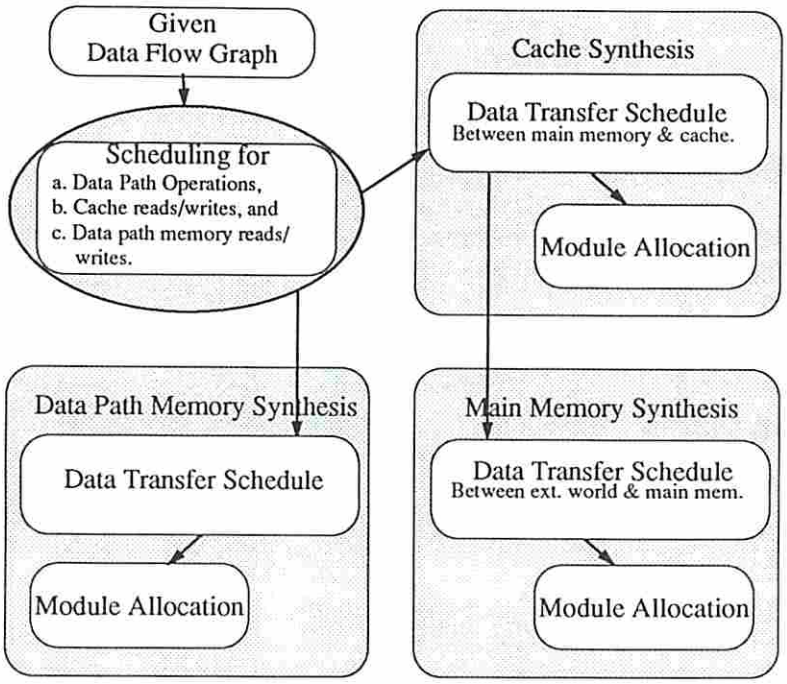

Figure 3: Memory synthesis.

8 Data transfer scheduling for the cache memory

To reduce the complexity of the problem HISS synthesizes the above-mentioned different structures, (i) main memory, (ii) cache memory, and (iii) data path memory, separately.

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. HISS makes use of the above mentioned 3 tradeoffs during synthesis of each sub-architecture. In this paper we give the details for data transfer scheduling for the cache, as an example. The overall synthesis approach and the order are shown in Figure 3. We assume two-phase clocking for our targeted system. For the main memory, the data is written in one phase and read in the other 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 cache in the second phase. In case the cache is transparent, the data path interacts directly with the main memory; so instead of writing the data back into cache, it writes it into the main

memory in the second phase. A simple example illustrating our approach is shown in Figure 4.

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 is to schedule data accesses from the main memory to the cache to satisfy the schedule obtained in the previous step in Section 6, 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 complete data transfer schedule to and from main memory, read time and write time for all the data points, such that the size of the cache is minimized, and the cache provides the data to the data path as they are required, with the given number of ports.

The scheduling in HISS is analogous to the scheduling problem in data path synthesis. HISS uses *list scheduling*, with a redefinition of the objective function, to schedule cache/main memory reads and writes. HISS maintains 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. So, unless it is necessary to transfer the value into the memory transfers we will postpone it. From that point of view it is a greedy algorithm. For such an approach it is useful to process the

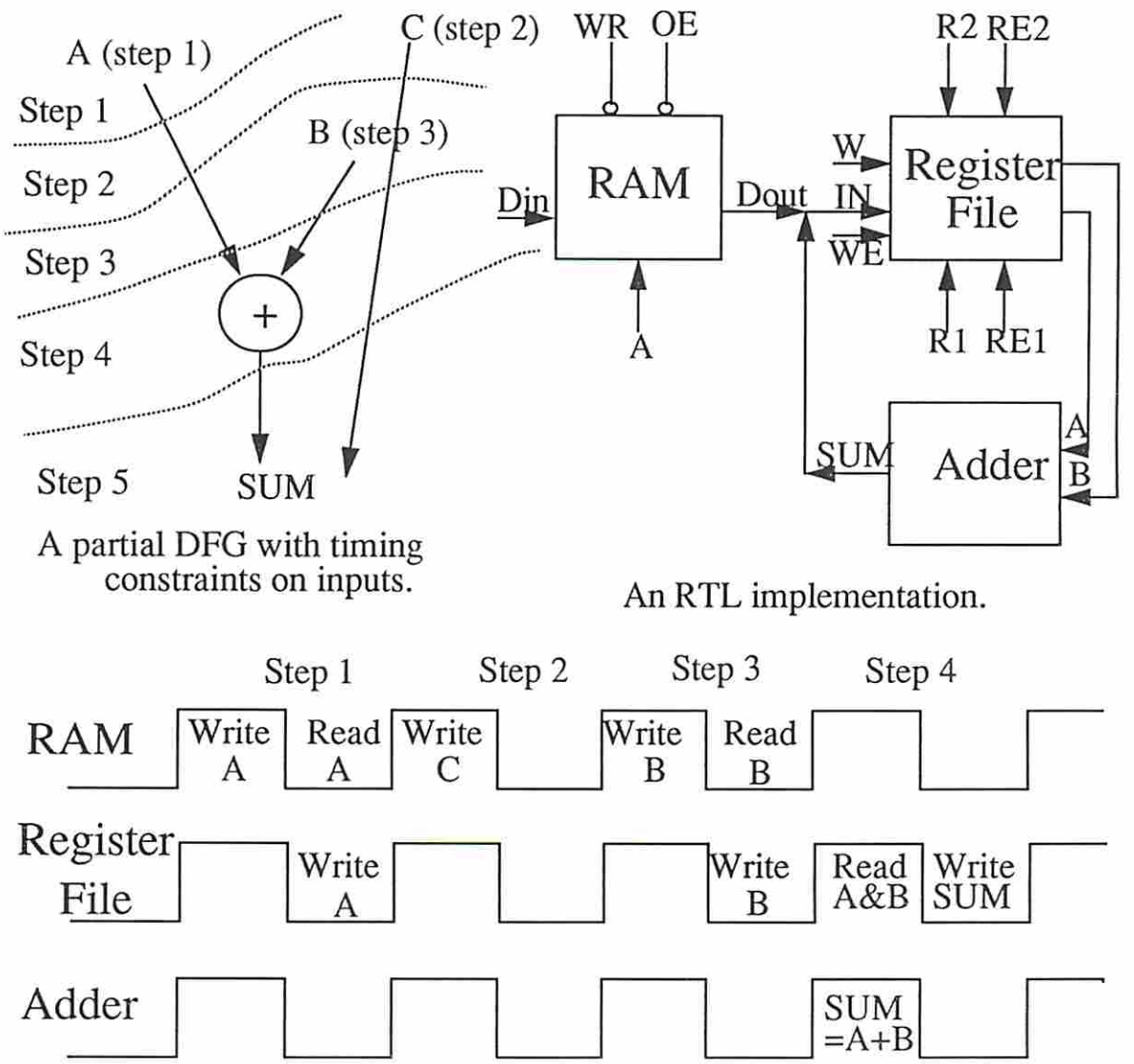

Figure 4: An example illustrating 2-phase clocking.

schedule backwards. We start from the last time step and work backwards to the first step. The urgency factor for these values will depend on the following factors: (i) 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 value 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 flowchart of the algorithm is given in Figure 5.

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.

9 Experimental results

We applied our approach to two representative examples: an AR lattice filter element [11], shown in Figure 6 and a second-order differential equation example (Figure 7) [12]. 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. The results are summarized in Table 1. Figure 8 gives the data path schedule and data transfer schedule for the cache for design 3. The data path schedule lists the operations and the corresponding step number. The input data transfer schedule for the cache gives the read/write schedule for the cache. For example, in step 3 we have,

```
STEP 3 --> READs (2) -->  x  dx
           Memory Contents ->  x  dx
           WRITEs (1) -->  dx
```

This means that in step 3 one input 'dx' is being written into the cache from the main memory, the cache contents are 'x' and 'dx', and in this step we read 2 data values - 'x' and 'dx' from the cache for the data path. Note that 'x' was discarded after this read

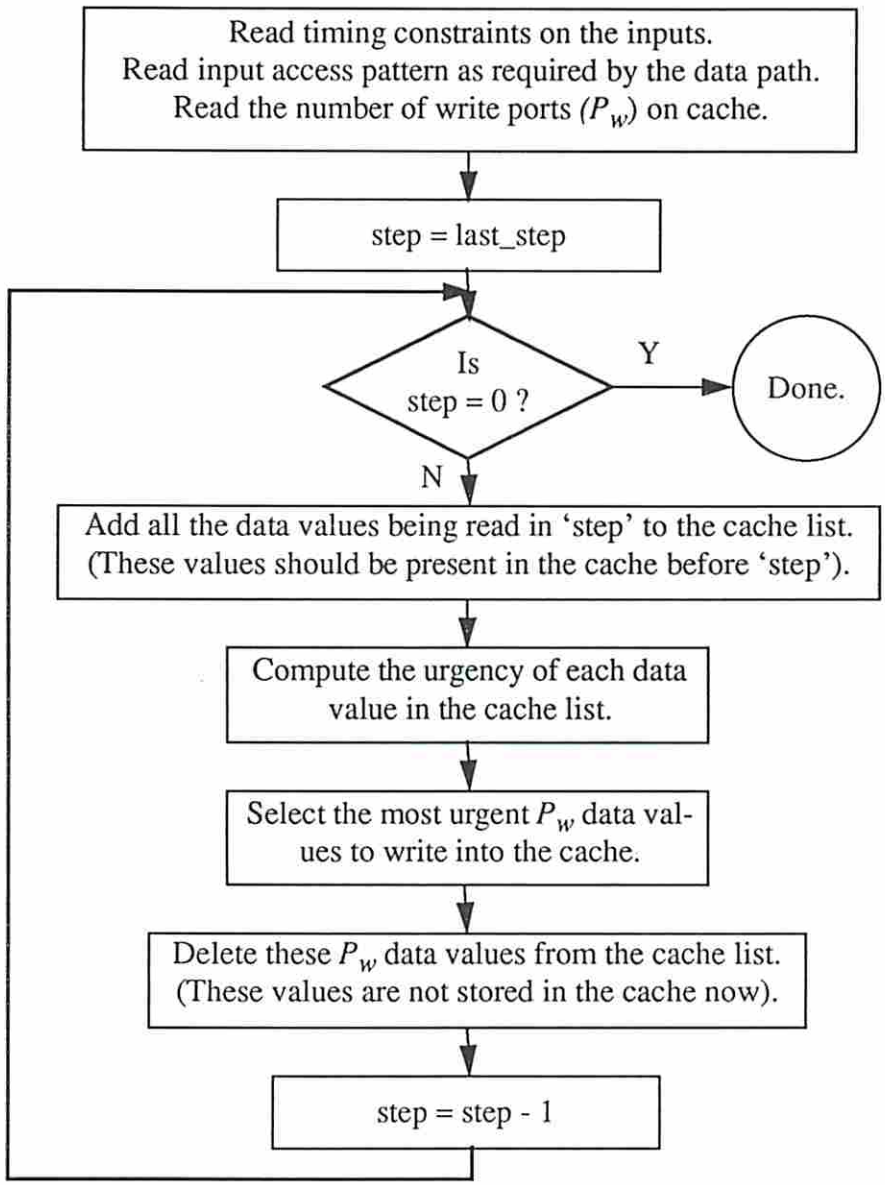

Figure 5: Data transfer scheduling for cache in HISS.

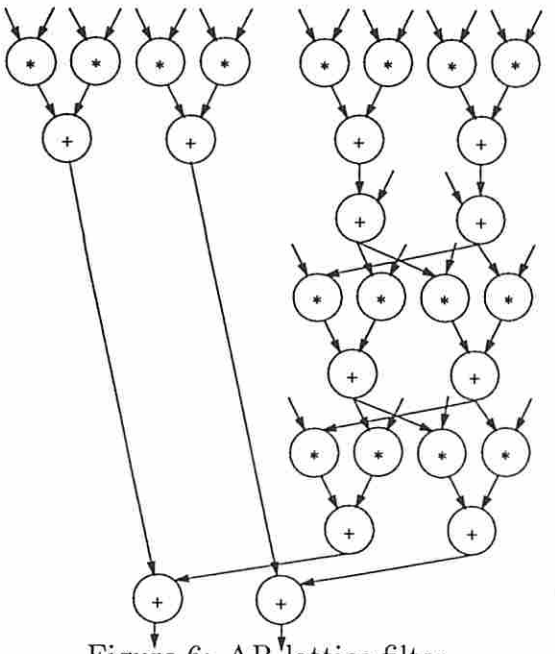

Figure 6: AR lattice filter.

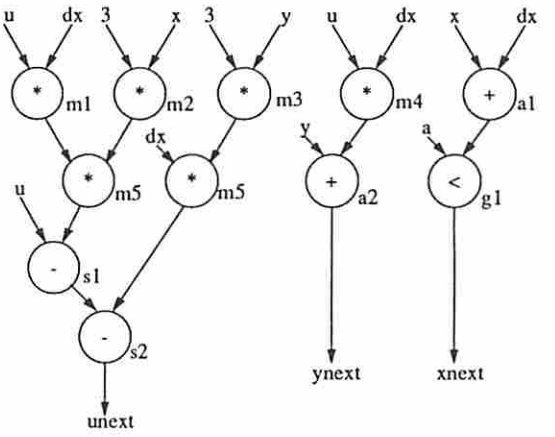

Figure 7: Second-order differential equation example.

as it doesn't appear again in the memory contents in step 4, while 'dx' does, implying that 'dx' was saved for future use.

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 cache at any time. In examples 1 and 3, since the bandwidth between the main memory and the cache is very restricted, we prefetch the data and store it in cache before it is demanded by the data path. If the data value is not required again we can overwrite it with new data and thus reuse the memory location. If it is required again we check if we can transfer it again to the cache from the main memory. Depending on that we either keep it stored in the cache or overwrite it with a new value in order to save memory. For example, in the data transfer schedule for design 3 (shown in Figure 8) *const3* is used in steps 2 and 6. It is transferred into the cache from the main memory in step 2 and we could have just saved it in the cache for step 6. But we transfer it again in step 5, because we realize that before it is required in step 6, we can rewrite it into the cache in step 5. This way we could overwrite the memory location used by *const3* after step 2.

In these examples we had a very stringent memory bandwidth constraint. The long

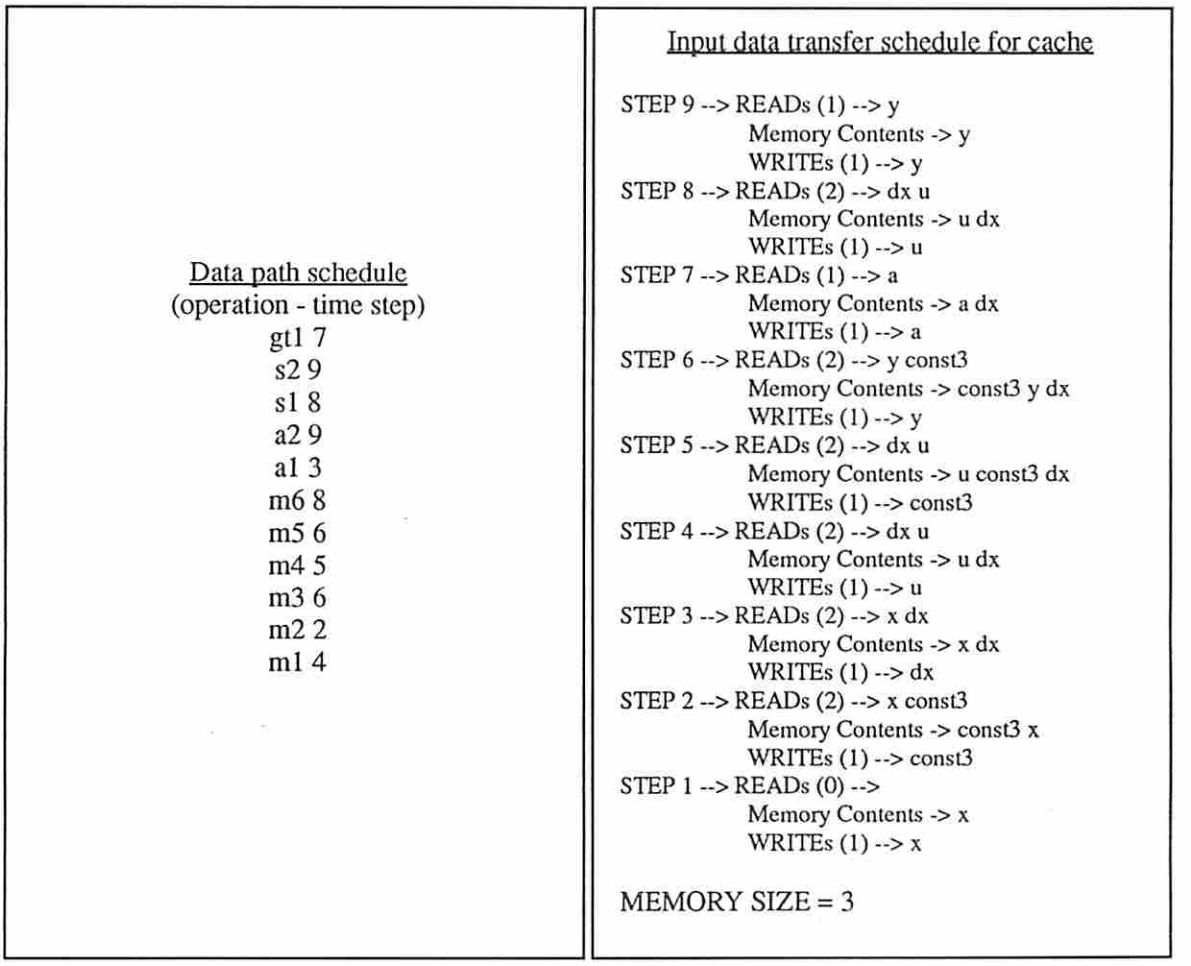

Figure 8: Output for design 3.

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, 2 mults., 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.

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 the second-order differential equation example a larger bandwidth between the cache and the data path can improve the execution time more effectively. Each of these examples ran in less than 100 ms. on a SUN-SPARC.

10 Conclusion

In this paper we demonstrated an approach to perform data path synthesis combined with the required hierarchical storage structures synthesis for ASICs. Our prototype synthesis package, HISS, takes in a behavioral description of a system along with the module library,

hardware and timing constraints and synthesizes a system with storage hierarchies. We demonstrated the capabilities of HISS by synthesizing two examples with varying design parameters. We are in the process of interfacing HISS with our ADAM framework and Cascade Design Automation's ChipCrafter. That will enable us to layout the whole design from the behavioral description within an acceptable time limit.

References

[1] I. Ahmad and C. Y. Roger Chen. Post-Processor For Data Path Synthesis Using Multiport Memories. In *Proc. of the International Conference on Computer Aided Design*, pages 276–279, 1991.

[2] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, and J.G. Linders. Allocation of Multiport Memories in Datapath Synthesis. In *Proc. of the International Conference on Computer Aided Design*, pages 266–269, 1987.

[3] C. H. Chen. Allocation of Multiport Memory with Ports of Different Type in Register Transfer Level Synthesis. In *International Conference on Computer Design*, pages 418–421, 1991.

[4] C. H. Chen and G. E. Sobelman. Singleport/Multiport Memory Synthesis in Data Path Design. In *Proc. of the IEEE International Symposium on Circuits and Systems*, pages 1110–1112, 1990.

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

[6] D.M. Grant, P.B. Denyer, and I. Finlay. Synthesis of Address Generators. In *Proc. of the International Conference on Computer Aided Design*, pages 116–118, 1989.

[7] 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 Conference*, pages 11.7.1–11.7.4, May 1991.

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

[9] 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.

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

[11] A. C. Parker, Pravil Gupta, and Agha Hussain. The Effects of Physical Design Characteristics on the Area - Performance Tradeoff Curve. In *Proc. of the 28th Design Automation Conference*, pages 530–534, June 1991.

[12] 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.

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

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