SMASH: A Program for Scheduling Memory-Intensive Application Specific Hardware

Pravil Gupta and Alice C. Parker

CENG Technical Report 93-39

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

December 1993

SMASH: A Program for Scheduling Memory-Intensive Application Specific Hardware*

Pravil Gupta and Alice C. Parker
Electrical Engineering – Systems
University of Southern California
Los Angeles, CA 90089-2562

*This work was supported by the 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 Advanced Research Projects Agency or the U.S. Government.

1 Introduction

The topic of the research described in this paper is automatic synthesis of memory-intensive application specific systems, with emphasis on hierarchical storage architecture design. Such systems are commonplace in video signal processing and other real-time applications and they demand high performance datapaths along with efficient storage schemes to support them. Our goal is to design datapaths and storage architectures for such systems. The storage architecture is closely connected to the system datapath, and isolating its synthesis from datapath synthesis may not result in an efficient solution. Our datapath synthesis procedures themselves design the on-chip/off-chip memory hierarchy which is companion to the datapath.

The bulk of research on memory hierarchy design involves theoretical and probabilistic studies for general-purpose computer design where the memory-access pattern varies from application to application. Our tools will be used for systems designed for specific applications, where 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.

We design a hierarchical storage system concurrently with the datapath and also determine the input/output data-transfer schedule between various hierarchies and datapath, as the datapath itself is scheduled. The need for such a combined synthesis is illustrated in the following example, where we have scheduled a simple data flow graph considering various memory-related issues (Figures 1 a, b, and c). In Figure 1 a the datapath scheduling is done

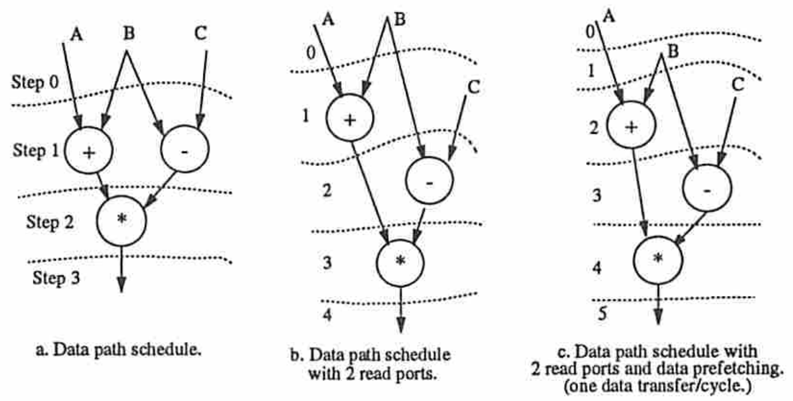
A
B
C
Step 0
Step 1
Step 2
Step 3
+
-
*
a. Data path schedule.
b. Data path schedule with 2 read ports.
c. Data path schedule with 2 read ports and data prefetching. (one data transfer/cycle.)

Figure 1: An example showing scheduling on memory related issues.

without considering any memory-related issues. As a result, all the inputs A, B, and C are required simultaneously in step 1 so, the module(s) storing A, B and C should have 3 read ports and these inputs must be transferred on-chip from outside in one step, demanding a bandwidth of 3 words/cycle. Furthermore, a 3-word buffer is required to store these inputs after the transfer.

In Figure 1 b the scheduling is done with limited read ports. This schedule results in operators requiring at the most two inputs in any step (so, we require only 2 read ports on the storage modules(s) here), but the design still requires a bandwidth of 2-words/cycle. The buffer size required here is reduced to two as A can be overwritten with C.

Finally, Figure 1 d shows the scheduling done with limitation on both the read ports and the bandwidth. This schedule requires only one data transfer per cycle from the external world and two inputs for operators in any step. The storage size required here remains two.

Notice that in a more realistic CDFG, there may be little of no delay in execution due to data transfer by overlapping the execution of other parts of the CDFG with data transfer.

2 Problem statement

An overview of our software is shown in Figure 2. We assume the following information as inputs to our program:

- the behavioral VHDL description of a memory-intensive application-specific system, which may contain inner loops and conditional branches. The loop structure and all the arrays and indexed references are assumed to be already transformed and optimized.
- the module library consisting of (i) functional modules (e.g. adders) with each module characterized by its area, delay and bitwidth, and (ii) storage modules (e.g. registers, single-port/multiport register-files, single-port/multi-port RAMs) characterized by cost per word, number of ports, access time and storage capacity;
- area-performance constraints;
- the clock cycle, which is the duration of each control step in the datapath;

- input/output timing constraints imposed by the external world; and
- memory bandwidth constraints (the number of words that can be transferred onto the chip in one control step).

Our synthesis system produces a two-chip system with

- a datapath consisting of operators and operation schedule,
- size and port configuration for on-chip foreground memory to store input/output and intermediate variables,
- data-transfer schedule between the datapath and on-chip memory,
- size and port configuration off-chip background memory for bulk storage, and
- data-transfer schedule between the on-chip and off-chip memory.

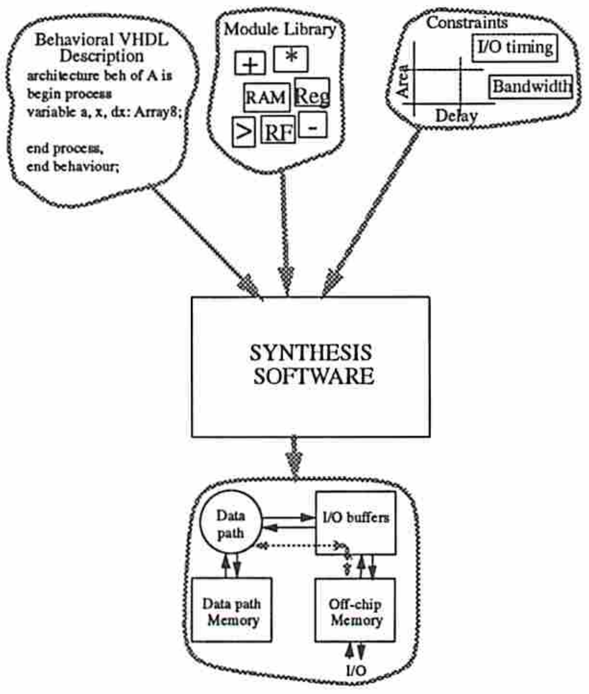

Figure 2: Synthesis of memory-intensive systems.

3 Related research

The original MIMOLA system was the first system to make tradeoffs in the use of multiport memories [7]. Balakrishnan et al. [2] presented an approach to use multi-port memories to implement single isolated registers. Chen [3] explored the design space for multiport memory synthesis. Ahmad and Chen [1] use 0-1 integer-linear programming to group intermediate variables in the datapath into a minimum number of multiport memories depending on their ports and their access pattern. Stok [11] optimizes register files during the synthesis process. Grant et al. [4] suggested an approach to group the memory requirements of various operators using signle-port memory modules such that control and communications may be optimized. In an earlier study the same group studied address generation [5].

Recently Lippens et al. [6] 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 datapath scheduling and allocation. They model multi-dimensional periodic signals as data streams and then manipulate these streams to form a distributed memory structure. They assume a limited number of memory types available to them (1 and 2 port RAMs) and so their approach is to distribute the data among parallel memories. They do not distinguish between background and foreground memory. They do not consider conditional branching in the behavior of the system.

In IMEC's CATHEDRAL-II, efficient storage schemes and memory access techniques were implemented [12]. They compile multi-dimensional data structures into distributed dual-port register files and single-port SRAMs. They also consider multi-dimensional signals and optimize the memory organization by loop transformations. They perform high-level memory management prior to datapath synthesis.

All the above efforts concentrate on separate aspects of memory synthesis. In this paper, we describe a software which combines tradeoffs for datapath synthesis as well as storage architecture synthesis.

4 Target architecture

As mentioned earlier, our target system consists of a **datapath** and a **hierarchical storage architecture** (Figure 3), which includes all the storage modules. In our model, the datapath *per se* does not have any kind of storage capability. All the intermediate variables are stored in datapath memory, which is described later.

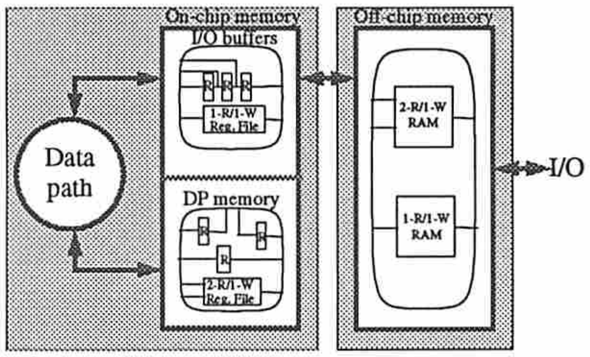

Figure 3: Target architecture.

We have classified the overall storage system into two levels, on-chip foreground memory, and off-chip background memory.

The **on-chip foreground memory** consists of I/O buffers and datapath memory. I/O buffers temporarily store the variables brought in from the off chip or to be written off chip. The on-chip I/O buffers make the appropriate input variables available at every step. The relevant parameters that are being determined by our software are (i) total buffer size, which is determined by the maximum number of inputs/outputs stored in the buffers in any given step, and (ii) the number of read/write ports accessible to the datapath R_{buf}/W_{buf}, which is the maximum number of inputs/outputs the datapath accesses in any given step. The user specifies the bandwidth between the buffers and the off-chip memory (BW_{on-off}), which is the number of inputs/outputs that can be transferred from/to the off-chip memory in one control step. Datapath memory stores the intermediate or temporary variables in the datapath. It is distributed throughout the datapath. The parameters which determine this subpart are derived from the CDFG and schedule and include (i) number of variables, and (ii) the lifetime of these variables. However this kind of tradeoff study has already been performed by other researchers [2, 3, 11] and is not described in this paper.

Off-chip background memory is the bulk storage required for the inputs and outputs. All the I/O data values from/to the external world are stored here. The purpose of the off-chip memory is to provide large cheap storage space, just as in a general purpose computer. The interaction between this memory and the datapath is via the on-chip buffers. The buffers degenerate into a simple set of wires if they are not needed. The parameters relevant to off-chip memory are: (i) the number of read/write ports (given by the user as BW_{on-off})

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

Although the top-level target architecture is fixed, there is a great deal of flexibility for each subpart. For example, the I/O buffers in the on-chip memory can be made up of a heterogeneous combination of registers, single-port/multi-port register-files, and single-port/multi-port RAMs. Datapath memory may consists of registers and single-port/multi-port register-files and may even contain RAMs if the storage requirements are huge enough. The background memory, which is bulk off-chip storage, is constructed using larger modules like RAMs. In an extreme case, a subpart can degenerate to interconnections if there is no need for any storage. Such a part will not have any cost and no storage capability.

In the proposed model, the bandwidth between the on-chip and the off-chip memory imposes cost constraints on the overall design because of (i) the pin constraints on the chips, and (ii) the expense of having multiport memory for off-chip bulk storage. Furthermore, the access time for the off-chip background memory will be greater because (i) it is off-chip, and (ii) it is bigger in size. Since access to the data values in the off-chip memory may be slow as a result, the software schedules the transfer of the input variables from off-chip memory into the I/O buffer of the on-chip memory before they are required in order to avoid delay in the execution. In each control step the required data is loaded onto the buffer-ports and as the datapath reads data from the buffers, the controller adds new data to them from the off-chip memory for further processing. Similarly, the output variables are first stored in the on-chip memory as they are produced and then are transferred to off-chip memory at an appropriate time, overlapped with the execution of the datapath. If the data is used again in future steps and can not be easily retrieved later, it may be retained in the buffers.

5 Design methodology

All the major design steps of our methodology are shown in Figure 4. In this paper we have focussed on the following two steps (highlighted in the figure): First, **datapath synthesis** with operation scheduling is performed combined with scheduling of on-chip data transfers to/from I/O buffers. As a result of this scheduling, constraints are placed on the memory

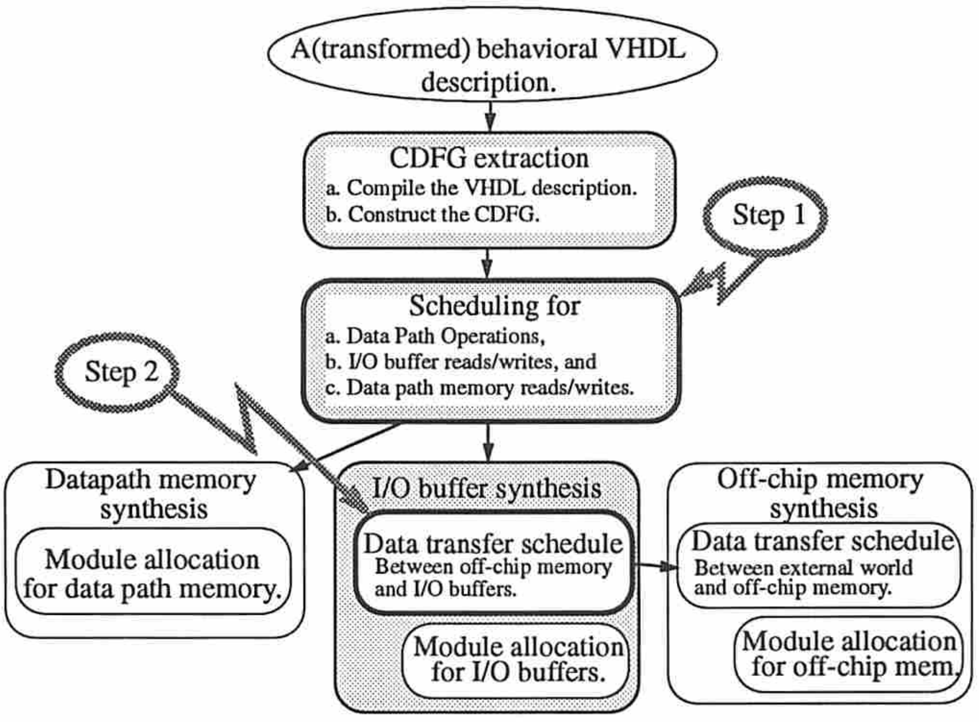
A(transformed) behavioral VHDL description.
CDFG extraction
a. Compile the VHDL description.
b. Construct the CDFG.
Step 1
Scheduling for
a. Data Path Operations,
b. I/O buffer reads/writes, and
c. Data path memory reads/writes.
Step 2
Datapath memory synthesis
Module allocation for data path memory.
I/O buffer synthesis
Data transfer schedule
Between off-chip memory and I/O buffers.
Module allocation for I/O buffers.
Off-chip memory synthesis
Data transfer schedule
Between external world and off-chip memory.
Module allocation for off-chip mem.

Figure 4: Design steps in our methodology.

structure. The second step, the **I/O transfer scheduling**, includes determining the data transfers between the two levels of the memory hierarchy[1]. We ensure that first step of the stepwise construction of the system takes into account the second step by looking ahead so that the second step is not overly constrained. Global design parameters like BW_{on-off} and timing constraints are considered when constructing the partial design in each step, tying the whole synthesis process together.

The datapath scheduling imposes significant constraints on the design; therefore, all the storage-architecture-related tradeoffs must be embedded in the datapath scheduling software itself. The following section describes the area-time tradeoffs in storage architecture which we have incorporated into our software.

5.1 Design tradeoffs in storage architecture

In the design of storage architecture, we have three parameters which vary while making the cost-performance tradeoff, (i) number of ports, which determines BW_{on-off} (ii) size of the storage architecture, and (iii) execution time. The software can tradeoff between (i)

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

more clock cycles (delay the execution) or (larger buffer size) prefetch the data before it is required and store it; (ii) more ports (wider bandwidth) or more clock cycles to transfer words; or (iii) more ports or larger buffer size for data values which are used again, (retrieved them repeatedly whenever needed with wider BW_{on-off} specified by the user or save them for future use which will increase the I/O buffer size). To deal with this 3-way tradeoff, we iterate on BW_{on-off} by repeatedly invoking the synthesis software and for each choice of BW_{on-off} specified by the user, the software trades-off between the size and the execution time. This can be done efficiently because the number of ports is small in practical designs.

5.2 Step 1: Combining datapath scheduling with I/O accesses

We combine the scheduling of I/O reads/writes from/to the on-chip buffers with the scheduling of operators by inserting a read/write (R/W) node, which corresponds to every I/O access to the buffers, into the CDFG. The read node consists of two inputs: (i) array name, and (ii) index and one output: value. The write node has three inputs: (i) array name, (ii) index, and (ii) value and one output: array (modified). A single data value is represented as an array of one element with the index input being a constant 0. These R/W nodes are treated as functional operators performing *data transfer*. During the first design-step (datapath scheduling), whenever an operator involving I/O is scheduled, the corresponding R/W node is also scheduled to the same step, implying an I/O buffer access in that step. Furthermore, all memory-related issues are considered during this scheduling to avoid violating the memory-related constraints imposed by the user (e.g. bandwidth and I/O timings) in the second step. In our approach this issue is resolved by checking if

1. the I/O buffers are able to provide the required number of R/W ports in that step;

$$R_{req}(s)/W_{req}(s) \leq R_{buf}/W_{buf} \text{ for all steps } s$$

2. the input data, if required, could be prefetched into the buffers from the off-chip memory prior to that step and the output data, if any could be transferred back to the

off-chip memory before it was required elsewhere;

$$I(s) \leq BW_{on-off} \times sO(s) \leq BW_{on-off} \times (MaxSteps - s)$$

where $I(s)$ ($O(s)$) is the number of data values required for (produced by) all the operations scheduled before (after) or in control step s for all s.

As a result of this approach, the schedule obtained in the first step guarantees that no bandwidth or I/O timing constraint is violated in the following design-step when the complete I/O transfer schedule between the on-chip and off-chip memory is determined. In addition, this process provides the I/O requirement schedule which is the starting point for the second step.

5.3 Overview of the datapath synthesis software

Inputs to the software: The program accepts as inputs the behavioral description of the system in VHDL, the area-performance constraints, the functional module library with area-delay characteristics, storage module library, timing constraints on I/O data values, BW_{on-off} constraints and clock cycle.

The user can also specify whether priority should be given to performance optimization or area optimization while satisfying the area-performance constraints [9]. When performance optimization is a primary objective, whenever the scheduler is unable to schedule a node, it first attempts to add an extra resource and if it fails adds an additional step (and the other way for area optimization). This feature facilitates the user in obtaining a design which is near-optimal for the desired parameter (area or performance).

The software also takes into account the timing constraints on the input/output data values, whenever imposed by the external world. An operation may not be scheduled in a step before the required inputs are available from the external world.

Assumptions: The synthesis software assumes two-phase clocking. The datapath reads and processes the data in the first phase and writes it back into memory in the second phase. In case the datapath interacts directly with the off-chip memory, it writes the data into the off-chip memory in the second phase.

The VHDL compiler and the synthesis software allow bounded loops in the VHDL description.

We estimate the cost of the I/O buffer using (i) buffer size (i.e. the maximum number of inputs/outputs stored in a step) and (ii) the required number of ports on it by counting the maximum number of I/O accesses in a single step. We have not emphasized minimizing the cost of the datapath memory in our work, as it has been widely researched elsewhere [2, 3, 11].

Scheduling technique: The software uses list scheduling with freedom of the nodes as the urgency factor [9], but any other scheduling technique which performs scheduling under hardware and performance constraints could be used. Distribution graphs of the *read (write)* operations and the datapath operations are used to assign them probabilistically to the most suitable control step, as is done in force-directed scheduling [10].

To improve the total execution time and allow operators with varying execution delays, the nodes in the CDFG are: *chained*, or *distributed over multiple cycles* appropriately. The scheduler allows operator sharing among mutually-exclusive nodes. A predicate bitmap representing the associated predicates is attached to each node, and is used in determining the mutual exclusion among the nodes very efficiently. The VHDL compiler allows only 2-way branching, therefore, multi-way branches are converted into multiple 2-way branches.

I/O for the operations inside the **conditional branches** is handled in the following way: We attempt to schedule the *test* node as early as possible so that the interval between the test node and operation node (requiring the I/O) is large enough to transfer the data into I/O buffers from off-chip memory. The data transfer must be done concurrently with the transfer of other data values being used in other parts of the CDFG. In step two when the actual data transfer is scheduled, the software may or may not be able to make all the required transfers in time. This gives rise to the following two scenarios, (Figure 5 a):

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

2. worst case analysis: in case there is not enough time (or bandwidth) to transfer the data, after the test is executed all the data values are scheduled to be transferred to the buffers irrespective of the outcome of the test, so that the appropriate data value is available during the execution. This results in extra data transfers but avoids delay in execution. Such situations may arise in real-time systems demanding very high performance.

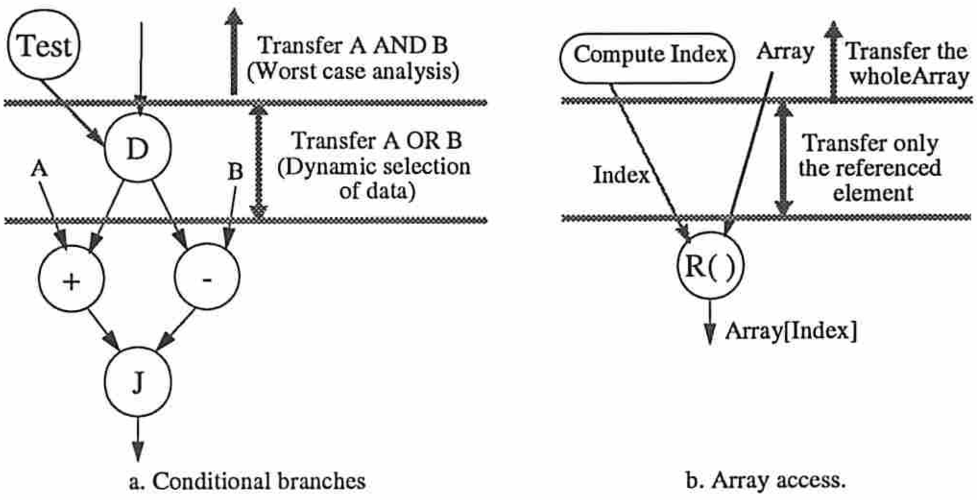
Test
Transfer A AND B (Worst case analysis)
D
A
B
Transfer A OR B (Dynamic selection of data)
+
-
J
a. Conditional branches
Compute Index
Array
Transfer the wholeArray
Index
Transfer only the referenced element
R( )
Array[Index]
b. Array access.

Figure 5: Data transfer into I/O buffers.

Our approach to **array accesses** is very similar to our handling of conditional branches, (Figure 5 b). Here we try to schedule the node computing the array index as early as possible. Again, during the actual data-transfer scheduling we could have

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

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

The **loops** are pipelined in order to achieve higher performance. They are scheduled first and then are treated as one unit for the scheduling of the remaining CDFG. The approach supports multiple loops as well as nested loops, but is yet to be completely tested for multiple and nested loops.

Outputs from the software: For the *datapath* the software determines the number of functional units of each type required for the design and the scheduling of all the operations in the CDFG to appropriate control steps. It also outputs the percentage utilization of each type of module in each step, which helps us determine the quality of the schedule.

5.4 Step 2: I/O transfer between on-chip and off-chip 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 the value to the operators.

Our second step is to schedule data accesses from the off-chip memory to the I/O buffers to satisfy the schedule obtained in the previous step in Section 5.2, with the objective of minimizing the buffer size while meeting timing constraints. The problem is described as follows: given BW_{on-off}, the time interval during which the data is available, and the data requirement (as determined in the previous step); we determine a complete data transfer schedule to and from off-chip memory, read time and write time for all the data points, such that the size of the I/O buffers is minimized, and the buffers provide the data to the datapath as they are required and stores the outputs as they are produced.

Inputs to the software: The software expects the BW_{on-off} and the data requirement, determined in the datapath synthesis step; and the interval during which each data value is available, determined by the external timing constraints.

Overview of the technique: The data-transfer scheduling is analogous to the scheduling problem in datapath synthesis. The software uses *list scheduling*, with a redefinition of the objective function, to schedule on-chip/off-chip memory reads and writes. A list of all the data values to be scheduled is maintained. 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 fetch the inputs as late as possible because we know that presence of data in the buffers will contribute to the size of the buffers. So, unless it is necessary to transfer the inputs into the buffers we will postpone it. From that point of view it is a greedy algorithm. The urgency factor for these values will depend on the following factors: (i) the number of steps remaining to prefetch the data into buffers. 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 datapath then we can lower its priority because all we need to do is to keep it stored in the buffer. Similarly, the outputs transferred back to the off-chip memory as soon as possible in order to avoid their presence on-chip.

Outputs from the software: The output of step two consists of the complete data access schedule for the I/O buffers and the required buffer-size (determined from the access pattern.)

6 Experimental results

We applied our technique to three representative benchmark examples: a second-order differential equation solver [10], an AR filter element [8], and an Elliptic filter element.

Our experiments were designed to demonstrate the following aspects of our research: (i) the capabilities of the software to combine the datapath scheduling with I/O access from the buffers, (ii) validating datapath schedules obtained when considering memory-related constraints, and (iii) existence of the cost-performance tradeoff in the storage architecture (cost being a function of R/W ports, storage size, and BW_{on-off}).

Module name	Area (sq. microns)	Delay (ns)	Module name	Area (sq.microns)	Delay (ns)
Multiplier	2398490	55	Comparator	81558	0
Adder	81558	30	Distribute/Join	0	0
Subtractor	81558	30	Register	96624	3

Table 1: module library.

The module library used is shown in Table 1. The clock cycle used is 30ns. To simplify the experiments we have assumed that the input data is present in an off-chip RAM before processing. The software was executed with priority on performance optimization while meeting the area constraint. We also generated some designs without considering memory-related constraints (BW_{on-off} and R_{buf}/W_{buf}). We implemented several designs with varying parameters for these examples. These implementations are summarized in Table 2. The storage architecture related parameters are shown in parentheses for the designs without memory-related constraints and are manually determined (when they are mapped on our target architecture).

Design number	Input constraints		Software output				
	Functional area (sq.microns)	BW_{on-off} (words/ clock cycle)	R_{buf}	W_{buf}	Buffer size (words)	Functional resources	Execution time (ns)
2nd-order differential equation							
1	7500000	NA (4)	NA (4)	NA (2)	NA (0)	<, +, -, 3 *	240
2	7500000	2	4	2	2	<, +, -, 3 *	240
3	6000000	NA (4)	NA (4)	NA (1)	NA (0)	<, +, -, 2 *	300
4	6000000	2	3	1	2	<, +, -, 2 *	300
5	6000000	1	3	1	3	<, +, -, 2 *	330
6	3000000	2	2	1	1	<, +, -, *	450
7	3000000	1	2	1	3	<, +, -, *	480
AR filter element							
1	10000000	NA (6)	NA (6)	NA (2)	NA (0)	2 +, 4 *	390
2	10000000	4	4	2	0	2 +, 4 *	390
3	10000000	3	4	2	4	2 +, 4 *	420
4	5000000	2	4	2	2	2 +, 2 *	600
Elliptic wave filter							
1	10000000	NA (3)	NA (3)	NA (4)	NA (3)	4 +, 4 *	540
2	10000000	3	3	4	3	4 +, 4 *	540
3	7500000	2	2	2	3	3 +, 3 *	600
4	5500000	2	3	3	2	3+, 2 *	690
5	5000000	2	2	1	1	2 +, 2 *	720

Table 2: Experimental results.

In this section, we analyze the differential equation example in detail. Designs 1 and 3 *vs.* 2 and 4 respectively clearly demonstrates the effectiveness of our technique in reducing the unnecessary storage cost of ports on off-chip memory (BW_{on-off}). Schedules with memory constraints achieved the same performance with the same functional area while reducing BW_{on-off} and R_{buf}/W_{buf}. This was because in our technique we distributed the data requirement evenly and overlapped the data transfer with the execution.

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

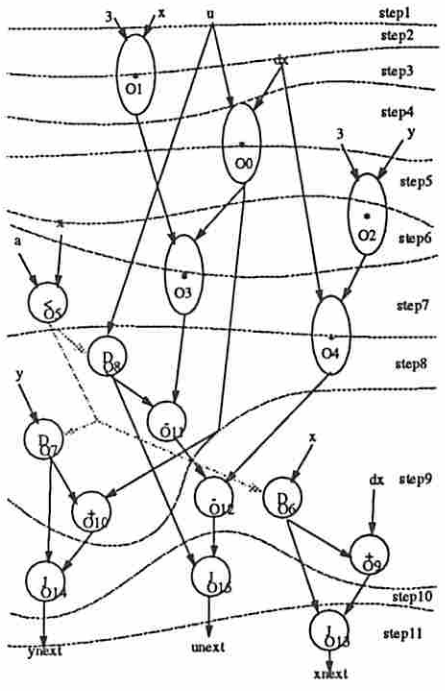

Figure 6: Scheduled CDFG for design 5 (2nd-order differential equation).
Figure 6 shows the datapath schedule in detail for design 5, excluding the R/W nodes. This figure also does not include the on-chip off-chip data transfer schedule, as that task is

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

Table 3: Data transfer schedule for design 5 (2nd-order differential equation).

performed in the second design step. The results obtained from the data transfer scheduling software are summarized in Table 3. Observe how inputs 'x' and 'u' are read twice into the buffer because there was an empty 'slot' available for retransfer before they were required again, and storing them in the buffers would have resulted in an increase in the buffer size. On the other hand, 'dx' was stored in the buffers for future use because its retransfer was not possible before the given time. Note that the inputs which are transferred from the off-chip memory and required by the datapath in the same step need not be stored. Similarly the outputs which are produced and transferred back to off-chip memory in the same step are not stored.

7 Future work

We are in the process of completing the remaining steps of our design methodology. The software system will then be interfaced with Cascade Design Automation's ChipCrafter which will enable us to produce a layout of 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 Int'l Conf. on Computer Aided Design*, pages 276–279, 1991.

[2] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, J.G. Linders, and J.C. Majithia. Allocation of Multiport Memories in Datapath Synthesis. *IEEE Trans. on Computer Aided Design*, 7(4):536–540, April 1988.

[3] C. H. Chen. Allocation of Multiport Memory with Ports of Different Types in Register Transfer Level Synthesis. In *Proc. of the Int'l Conf. on Computer Design*, pages 418–421, 1991.

[4] 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.

[5] D.M. Grant, P.B. Denyer, and I. Finlay. Synthesis of Address Generators. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 116–118, 1989.

[6] 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.

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

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

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

[10] 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.

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

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