Synthesis of Pipelined Memory Access Controllers for Streamed Data Applications on FPGA-based Computing Engines

Joonseok Park and Pedro C. Diniz
University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{joonseok, pedro}@isi.edu

ABSTRACT

Commercially available behavioral synthesis tools do not adequately support FPGA vendor-specific external memory interfaces making it extremely difficult to exploit pipelined memory access modes as well as application specific memory operations scheduling critical for high-performance solutions. This lack of support substantially increases the complexity and the burden on designers in the mapping of applications to FPGA-based computing engines. In this paper we address the problem of external memory interfacing and aggressive scheduling of memory operations by proposing a decoupled architecture with two components - one component captures the specific target architecture timing while the other component uses application specific memory access pattern information. Our results support the claim that it is possible to exploit application specific information and integrate that knowledge into custom schedulers that mix pipelined and non-pipelined access modes aimed at reducing the overhead associated with external memory accesses. The results also reveal that the additional design complexity of the scheduler, and its impact in the overall design is minimal.

Keywords: FPGA-based configurable computing; Scheduling of Memory Accesses; Hardware Interfaces and Customizable Memory Controllers.

1. INTRODUCTION

Commercially available synthesis tools for FPGAs have mostly ignored system level issues when dealing with external memories. While some tools now incorporate internal RAM modules and the mapping of array variables to them, they have avoided all external memory interfacing issues. Even for internal memories, the current tools require specific syntax and synthesis expertise precluding the interfacing with third party vendors for which timing information is either unavailable or cannot fit into the timing model of the tool. In addition most, if not all, behavioral synthesis tools do not support pipelined modes further restricting an important source of performance improvement. As a result designers must engage in laborious and error-prone matching and synthesis of interface signals.

Configurable architectures offer a unique opportunity to address the memory access and interfacing issues via customization. Improvements can be achieved by creating specialized hardware components for generating addresses and packing and unpacking data items. Compiler analysis can provide application knowledge as to the memory access patterns for pipelining of data references corresponding to array references across multiple iterations of loops. Furthermore the compiler can derive information about the relative rate among various array references and embed that knowledge into the scheduling of memory operations.

In this paper we focus on the design, implementation and validation of external memory interfacing modules that are generated by a compiler and behavioral synthesis tool that translate high-level computations directly to FPGA hardware. For maximum generality, we have separated the memory interface into two components. The first component is target-dependent and captures the specific target architecture timing requirements for accessing memory. The second component is architecture-independent and provides a set of channels and memory access modes abstractions for the application to store and retrieve data from memory. In particular the designs we have implemented allow compiler-generated designs to exploit pipelined access modes and mix both pipelined and non-pipelined memory accesses modes.

We used the interfaces described in this paper to develop application specific scheduling strategies for a set of four digital image-processing applications. The experimental results reveal that while the designs do not exploit all of the available external memory bandwidth, they achieve respectable performance at a reasonable implementation and generality cost.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada,

This paper makes the following contributions:

- Describes an abstract memory channel architecture with the concepts of channels, FIFOs and schedulers for managing memory accesses. This architecture decouples the target-dependent from the application-specific implementation knowledge thereby decreasing the complexity of interfacing with commercially available target architectures.

- Describes a suite of simple memory access schedulers for both pipelined and non-pipelined access modes for statically schedule streamed data access patterns. The schedulers are parameterizable and interface with behavioral code for the core datapath to generate complete designs.

- Describes an optimization in the definition of the memory controller scheduler that exploits application specific knowledge. This optimization reduces the number of controller cycles improving the overall design's performance.

- Presents experimental results of the application of distinct scheduling strategies for a small set of kernel applications on a commercial FPGA-based computing engine. For these kernels pipelined access mode improves the performance up to twofold while the controller FSM optimization further improves up to 10%.

The results presented in this paper support the claim that it is possible to exploit application specific information and integrate that knowledge in a custom scheduler for reducing the overhead associated with external memory accesses. The results also reveal that the additional design complexity of the scheduler, and its impact in the overall design is minimal.

While generality and performance are often conflicting goals, the complexity reduction, faster and more robust mapping of applications to FPGA-based hardware, are non-negligible factors in fast-prototyping settings. We believe that the simple abstract memory architecture with the interfaces presented in this paper are fundamental concepts that allow the easy integration of high-level analysis and program transformations in the mapping of applications to configurable computing architectures.

The rest of this paper is organized as follows. In the next section we describe the proposed external memory interface and its underlying abstractions. Next we describe the opportunities for application specific memory channel scheduling strategies. In section 3 we describe the basic analysis our compiler uses to generate the memory interfaces by analysis of the application programs. Section 4 presents a set of experimental results. We survey related work in section 5 and conclude in section 6.

2. EXTERNAL MEMORY INTERFACING

The proposed external memory interface is defined around a target-dependent interface and a target-independent interface. The target-dependent interface takes into account the specific timing requirements of the target hardware architecture and for practical reasons is tightly coupled to the vendor-supplied memory interface. This interface includes strict requirements that every memory access must meet such as the relative timing of memory operations; clock cycle counts for each type of access; and binding of pins in the design. The target-independent interface allows the core datapath that implements the computation in hardware to interface with input/output FIFO queues using a simple, timing-independent protocol.

2.1 Streamed Execution Model

To support the operation of streamed applications we have defined several memory access abstractions to exploit application-specific knowledge and advanced memory access modes, respectively, an **address generation unit (AGU)**, a **memory access scheduler (MAS)** and **FIFO queues** (see Figure 1).

The core of the computation is carried out by a datapath with input and output ports. These ports are physically bound to FIFO queues to retrieve and store data from memory. As computation progresses the datapath issues requests for data from the FIFO queues associated with its ports. These requests are then translated into low-level signals via the target-dependent interface. A simple handshaking protocol allows the design control to be informed about when the data has been deposited into the FIFO queue.

2.2 Memory Architecture Abstractions

Associated with each FIFO queue there is the notion of a **channel** and a **data stream**. A data stream defines a sequence of addresses and is identified by the tuple *(base_address, offset, stride*). A channel binds a port, the corresponding FIFO queue in time, by the contents of specific hardware resources in the address generation unit (AGU). The AGU consists of a register table with one entry per data stream supported. Each entry has a set of programmable fields to keep physical address data, an arithmetic unit to update and generate physical addresses, and a FSM to generate control signals. The details of the implementation are described in [4].

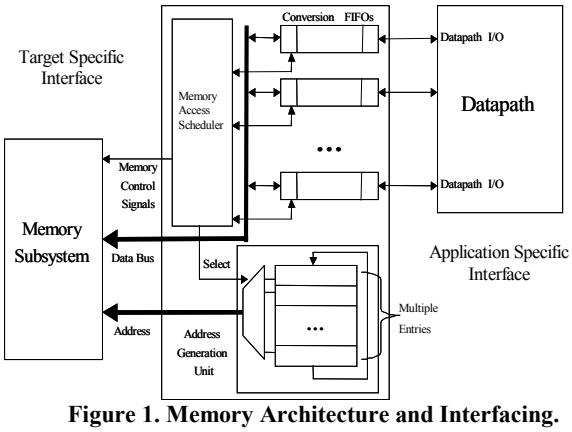

Figure 1. Memory Architecture and Interfacing.

We have defined and implemented both interfaces and hardware abstractions to support the data stream execution model of this approach for designs with a single computation task and with both single and multiple external memories.

2.3 Application Specific Scheduling

In the simpler case where the memory accesses can be determined statically, the compiler can **schedule** the order in which the accesses should occur. The scheduler, implemented in hardware via a FSM, should avoid deadlocking due to internal state transition conditions of multiple outstanding memory operations that can be dependent on each other. The memory access scheduler is defined by three hardware processes. A process checks which of the channels needing service by checking if the corresponding FIFO is empty (or full); a process to issue memory requests and a process to synchronize the transactions of the other two processes.

A simple scheduling calls for the handling of each memory channel corresponding to the datapath input/output ports in a **round-robin** fashion. For each computation, typically an iteration of a loop, the scheduler checks whether or not a given channel needs to access memory by inspecting the FIFO queue associated with it. To be able to handle conditional memory accesses we convert all conditional accesses to non-conditional accesses by speculatively fetching all of the potentially required. This approach, while wasteful of memory bandwidth allows for a much simpler memory controller. An alternative would be to implement a memory controller with the possibility of aborting memory accesses. This approach however would require timing the signals that dictate which memory references to abort, in the core datapath with the memory controller at the appropriate time. In the presence of pipelined execution modes this design would clearly become more sophisticated most likely leading to a reduced overall design performance.

Despite its apparent simplicity, this strategy interacts with other techniques common in synthesis such as loop pipelining. Using pipelining, loop prologues and loop epilogues do not exhibit all of the memory operations of the regular, steady state bodies of the loop. Typically the prologue generates no write operations and the epilogue requires no additional input data. The information about the scheduling of prologues and epilogues must be encoded in the scheduler's FSM, therefore increasing its complexity. In addition, round-robin works well if all of the channels have the same rate, that is, they produce/consume the same number of data items per cycle of the computation. If different channels have distinct rates, the scheduler must check for every computation, whether or not a memory operation is required. Again, because the FSM uses a larger number of predicates to determine its appropriate action, the implementation complexity increases.

A way to eliminate the overhead associated with the checking of which channels need to access memory is to embed information about several channels whose data access exhibit the same pattern. For example, if at every iteration of the computation there are three channels that need to access memory, testing for a single channel, is equivalent to test for any of them. Using this approach the number of FSM states, and therefore, cycles dedicated to hardware checks is substantially reduced. We call this approach a round-robin **group** scheduling strategy. Clearly, this group strategy can only be applied when the compiler can determine the exact relationship between memory accesses of the channels.

3. COMPILER ANALYSIS AND SUPPORT

We have integrated the synthesis of pipelined memory access controllers described in the previous sections in the DEFACTO compilation and synthesis system [3]. The current implementation analyzes source C programs in the intermediate SUIF representation format [10] and identifies opportunities for simple data reuse using tapped-delay lines. The analysis focuses on loop nests with array references using affine index functions (e.g., pixel[2*i + j +1]). While the analysis, as described in detail in [4] is capable of handling indexed for loops with compile-time unknown bounds, the current code generation phase must rely on known bounds to perform loop unrolling of inner that are to be mapped to hardware.

Given a set of array references in a loop nest that exhibit temporal reuse the compiler analyzes the reference index functions and the loop bounds to determine the *base address*, *offset* of the array being accessed as well as the *stride* of consecutive accesses. Figure 2 below illustrates the extraction of a single data stream corresponding to the accesses and data reuse of the array references in a loop.

```
for(i=1; i <= N; i++){                  // Compiler extracts
  tmp = 2*a[i] + a[i+1] + 2*a[i+2];     // stream 0 = (a, 1, 1) in
  if(tmp < threshold){
     e[2i-2] = 0;                       // stream 1 = (e,0,2) out
  } else {
     e[2i-2] = 1;
  }
}
```

Figure 2. Temporal Data Reuse and Stream Extraction.

Using the information about data streams the compiler uses library functions to define a data stream and programs an entry in the address generation unit (AGU) that will generate the sequence of addresses that correspond to the consecutive array accesses. The compiler also determines, using a very simple algorithm, the placement of the data in the FPGA's external memories, so that it can generate the base addresses for each of the data streams. In addition the compiler generates source C code, via a predefined set of library functions, that map the data to and from the host processor on the target FPGA's external memories.

As part of the DEFACTO code generation phase the compiler emits structural VHDL code that implements all of the functionality of the memory interface and corresponding abstractions using predefined VHDL parameterizable templates. This structural code is then "linked" automatically with the behavioral code that describes the code of the datapath computation by via a naming scheme for the individual datapath ports and corresponding streams.

4. EXPERIMENTAL RESULTS

We now present experimental results for the application of distinct memory access channel scheduling strategies as described in Section 2. We first describe the methodology used in these experiments and then present the results obtained for three image processing kernel computations running on a real FPGA-based computing board.

4.1 Methodology

We have mapped three (3) computation kernels from C to VHDL using the DEFACTO design system [3]. The DEFACTO mapping process generates behavioral VHDL specifications for computations in the body of loop nests it finds to be profitable to execute on an FPGA-based computing engine – the Annapolis WildStar™ [13] FPGA-based board.

Next, we have manually modified the memory channel interface for the target architecture to allow the implementation of two distinct flavors of the round-robin memory access scheduling strategy, namely **naïve (N)**, **pipelined(P), group (G)** and **pipelined with grouping(P+G)** scheduling. We then compare the performance of the designs using the different strategies. This performance comparison was carried out in a functional simulator, ModelSim™[7], where we are able to extract more precise clock cycle counts. We also confirmed the performance improvements via real executions on the WildStar™ board.

The WildStar™ board has three Xilinx® [15] Virtex 1000BG560 parts each of which is connected to two SRAM memory modules with 4Mbytes capacity. We have used Synplicity® Synplify® 5.1.5 and Xilinx® M1 place & route tool to generate the bitmap file for the Virtex parts. To exacerbate the problems of memory access scheduling, we mapped all of the applications data onto a single memory module. This approach allowed us to determine the severity of the memory scheduling issue. Techniques such as stripping data arrays onto different memory banks (see for example [5]) are orthogonal to the scheduling approach presented in this paper.

4.2 Applications

Sobel Edge Detection (SOBEL)

This application implements a simple eight point computations based on a 3-by-3 window of a 128-by-128 image of pixels. Each iteration of the inner most loop of the code's main loop nest consumes 8 data items from the input array variable and produces 1 item of the output array variable. The number of operations can be reduced significantly to 3 read and 1 write stream operations exploiting data reuse using tapped-delay lines across iterations of the main computation loop.

Automatic Target Recognition (ATR)

This application computes a binary image correlation using a binary mask array variable and a scalar accumulation variable. Because the compiler has aggressively exploited loop unrolling it generates a design with vast amount of parallelism and data reuse. Due to the unrolling, however, the number of data memory accesses per iteration of the inner loop is substantial. To study the impact of the number of memory accesses and hardware support for a varying number of channels we have implemented two variants of this application. One variant called ATR-4 uses 4 input channels and corresponds to using a fixed mask variable for the correlation computation. The other variant, called ATR-8 uses 8 input channels, 4 channels for the input image variable and another 4 for the binary mask.

Multiply-Accumulate-Zero (MAZ)

This computation computes the sums of the product of every pair of adjacent array values if the first value of the pair is smaller then the current running sum. For every addition, the computation zeros out the first element of the pair in the array. Because the accumulation and multiplication (16 bits output) is conditional and data dependent on the running sum of the computation, we call this computation multiply-accumulate-zero (MAZ). From the standpoint of the memory controller this computation exhibits a more irregular memory access pattern as some of the write operations are dependent not on an input values but rather on the value of an internal computation.

4.3 Results

We begin this discussion by first characterizing the execution of each of the applications using the default round-robin naive memory access scheduling strategy. Figure 3 below presents a breakdown of the execution time for the steady state of the main computation loop in each of these applications for a single memory bank implementation.

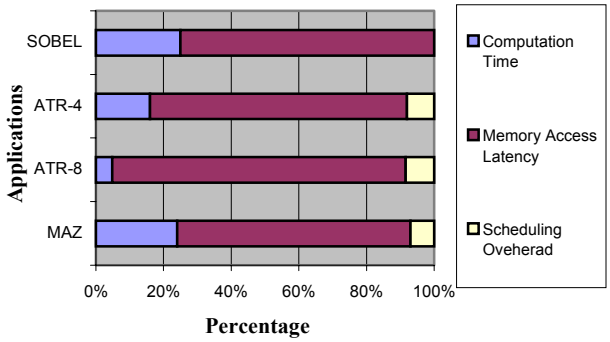

Figure 3. Execution Time Breakdown for Tested Applications.

As expected, and given that in these experiments memory accesses are blocking, the bulk of the execution time (60% to 80%) is spent stalling on memory accesses. Approximately 8% to 9% is spent checking the status of the input/output FIFO queues.

Table 1 shows the performance results for all applications for the different scheduling strategies. These results exclude the initial data loading and final data retrieval from the board. We report the overall design size in terms of FPGA slices; the maximum allowed clock rate for the design; the simulated execution time using a 25 MHz clock and the speedup measured as the ratio of the execution time of each version with respect to the computation using the **naïve** scheduling strategy.

Table 1 reveals that all designs are small (12.5% maximum FPGA occupancy) and therefore exhibit good performance characteristics in terms of maximum attainable clock rates. Table 1 also reveals the performance advantages of pipeline with an average speedup of 1.9 over the four tested kernels. Group scheduling by itself yields modest performance improvement with an average speedup of 1.1. When combined with pipelined, group scheduling boosts the average speedup to 2.05. This improvement

is most noticeable for ATR-8 where the number of channels with the same input/output behavior is the largest.

Table 1. Synthesis and Timing Results.

Applications		Slices (out of 12,288)	Max. Freq. (MHz)	Simulation Time (nsecs)	Speedup
SOBEL	**N**	1,144	30.1	1,312,020	1.00
	P	1,061	31.5	738,540	1.78
	G	1,160	31.7	1,312,140	1.00
	P+G	1,068	31.6	697,660	**1.88**
ATR-4	**N**	1,968	25.9	120,040	1.00
	P	1,980	25.6	66,600	1.82
	G	1,974	33.9	102,280	1.17
	P+G	1,984	26.9	59,600	**2.00**
ATR-8	**N**	2,771	25.9	188,440	1.00
	P	2,707	25.9	71,840	2.62
	G	2,718	30.8	163,440	1.15
	P+G	2,730	25.9	69,480	**2.71**
MAZ	**N**	1,027	30.4	85,760	1.00
	P	1,191	36.2	62,360	1.38
	G	1,226	31.5	78,680	1.09
	P+G	1,003	29.6	55,520	**1.55**

Table 2 shows the synthesis metrics for the synthesis of the channel controllers for each design. Overall the more sophisticated group-scheduling controller has clock rates in the 100MHz range and therefore appears not to impact the critical path of the whole design. By itself, the implementation of the group-scheduling controller requires no more than 21 additional slices than the simpler naïve controller does for a total a maximum of 75 slices barely 5% of the designs.

Table 2 Synthesis Metrics for Channel Controller (N: Naïve, P: Pipelined, G: Group P+G: Pipelined with Grouping).

Applications		CLBs	Gates	Clock Rate (MHz)
SOBEL	**N**	29	381	140.1
	P	25	333	130.3
	G	35	431	87.5
	P+G	35	448	134,7
ATR-4	**N**	35	458	127.3
	P	42	568	125.0
	G	40	531	105.2
	P+G	46	635	120.7
ATR-8	**N**	54	679	106.2
	P	74	937	82.2
	G	57	758	77.9
	P+G	75	1,010	69.4
MAZ	**N**	30	383	120.5
	P	34	450	112.8
	G	36	450	115.6
	P+G	45	559	116.4

4.4 Discussion

The experimental results, not surprisingly, reveal that pipelining techniques substantially improve the overall design performance. The implementation of group-scheduling techniques marginally increases the performance for the whole design with negligible impact in terms of area and very little influence on the maximum clock rate.

While we are able to eliminate almost all the memory overhead by pipelining and aggressive group scheduling there are several techniques that have been explored in other contexts and could be explore for the context of FPGA-based designs, namely:

- Reducing the sharing of physical bus channels will reduce the memory latency.
- Assigning multiple memory modules to disjoint input array for concurrent accesses.
- Aggressive pre-fetching and overlapping memory accesses with computations.

In this work we have focused exclusively on application level techniques that impact the design of the memory controller, rather than on architecture related approaches for reducing memory latency. We focused on the scheduling of memory accesses within a single computational task where memory accesses are statically scheduled. The scheduling in the context of multiple tasks may require a more flexible run-time scheduling strategy to minimize memory access contention. In the future we plan to address the implementation of dynamic, run-time scheduling implementation techniques where a schedule is setup only at run-time rather than statically for both single and multiple tasks.

Given the trade-off between generality and performance we have estimated the performance gap between the currently automated applications in this empirical study and what a designer could achieve exploiting the overlapping of computation in the core datapath with the communication with external memory. In Table 3 we compare the performance of the generated designs against an optimal solution where the memory accesses are perfectly scheduled and are fully overlapped with the computation in a zero latency scenario.

Table 3. Performance Expectation for Hand Designs (P+G+O: Pipelined with Grouping and Overlapping, OPT: Optimal Scheduling Design).

Applications		Speedup	Applications		Speedup
SOBEL	**N**	1.00	**ATR-8**	**N**	1.00
	P+G	**1.88**		**P+G**	**2.71**
	P+G+O	3.60		**P+G+O**	4.41
	OPT	7.99		**OPT**	7.00
ATR-4	**N**	1.00	**MAZ**	**N**	1.00
	P+G	**2.00**		**P+G**	**1.55**
	P+G+O	3.56		**P+G+O**	2.91
	OPT	7.50		**OPT**	6.48

While Table 3 reveals there is still a substantial performance gap between the automatically generated codes and the possibly infeasible optimal version, the effort and time investment for a hand design is still substantial, in particular for a novice programmer. While our designs take a few seconds to generate and about 30 minutes to synthesize and download onto the board, a hand design can take days if not weeks to design and verify its correctness.

5. RELATED WORK

Other researchers have addressed the issues memory operations scheduling in the context of application specific implementations.

Weinhardt and Luk developed memory access optimizations for pipeline vectorization in RAM interface [12] using a scheme to reduce consecutive memory accesses of array data using shift registers but accessing only on-chip RAM modules. Gokhale and Stone [5] proposed an automatic array allocation compile-time algorithm for multi-level memory subsystem. They attempt to allocate array variables to memories based on the memory latency, data access frequency and execution schedule. Schmit [11] developed a mapping scheme for mapping datapaths with memory operations directly to hardware. His approach uses a centralized memory controller scheme where the scheduling of the operations is done in conjunction with the execution of the datapath computation. Panda et. al. [8,9] refined this approach by defining a time-constrained based specification of the a centralized scheduler for handling external memory operations. Catthoor, Balasa et. al. developed and evaluated memory optimizations for embedded systems for a particular application set [1,2,6]. This research focuses on optimizations to minimize memory area and power consumption. Catthoor also proposed a data packing scheme to reduce memory bandwidth requirements for dynamic data structure. Wuytack et al. suggested minimizing memory bandwidth requirements [14] by mapping highly accessed array variables to fast hierarchy storage.

The research described in this paper differs from these efforts in several aspects. The common approach to memory scheduling operations has been to develop, based on a specification for the dependences of the operations, a centralized memory scheduler that fetches the data from an external memory into the datapath. Our approach provides an architecture independent view of the datapath and a memory controller that is decoupled from the datapath execution controller. This scheme has several advantages over the centralized control schemes proposed by previous researchers. First, in designs where pipelined memory accesses are desirable, the implementation complexity of a centralized scheme substantially increases. The scheduler needs to generate a controller that obeys the timing constrains for all the memory accesses in the pipeline. Second, our approach makes it easier to incorporate several tasks into the same design allowing them to share physical memory ports. A centralized memory controller scheme would require the description of a joint behavior for all the tasks a proposition that is hardly extensible and scalable.

6. CONCLUSION

The performance of FPGA-based computing platforms depends critically on the performance of external memory interface. In this paper we have described an approach to decouple the concerns of memory interfacing and static scheduling of possible memory accesses for applications with streamed data. We described the application of simple scheduling strategies to three kernel computations. The experimental results reveal that we are able to significantly increase the performance (speedup of two) of the overall design by using pipelined memory accesses. We further improve the performance by eliminating almost all of the latency due internal memory controller optimization resulting in a further 10% performance increase. The experiments reveal that the added complexity of the resulting designs is minimal.

7. REFERENCES

[1] F. Balasa, F. Catthoor, and H. De Man “"Dataflow-driven Memory Allocation for Multi-dimensional Signal Processing Systems", Proceedings of the IEEE International Conference on Computer Aided Design, Santa Jose, Calif., Nov. 1994, pp. 31-34.

[2] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan "Global communication and memory optimizing transformations for low power signal processing systems", IEEE workshop on VLSI signal processing, La Jolla, Calif., Oct. 1994.

[3] P. Diniz, M. Hall, J. Park, B. So and H. Ziegler, “Bridging the gap between Compilation and Behavioral Synthesis in the DEFACTO System”, To appear in the Proc. of the workshop on Languages and Compilers for Parallel Computing, (LCPC’2001).

[4] P. Diniz and J. Park, "Automatic Synthesis of Data Storage and Control Structures for FPGA-based Computing Machines", In Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines (FCCM'00), IEEE Computer Society Press, Los Alamitos, Calif., Oct. 2000, pp. 91-100.

[5] M. Gokhale and J. Stone “Automatic Allocation of Arrays to Memories in FPGA Processors With Multiple Memory Banks” Proc. of IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’99), IEEE Computer Society Press, Los Alamitos, Calif. Oct. 1999, pp. 63-69.

[6] M. Miranda, F.Catthoor, M. Janssen and H. DeMan, “High-level address optimization and synthesis techniques for data-transfer-intensive applications”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 6(4), Dec. 1998, pp. 677 –686.

[7] Monet™ User’s and Reference Manual Software Release R42, Mentor Graphics Inc., 1999.

[8] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle and P. Kjeldsberg. “Data and memory optimization techniques for embedded systems”, ACM Transactions on Design Automation of Electronic System, 6(2), Apr. 2001.

[9] P. Panda, N. Dutt and A. Nicolau, “Exploiting off-chip memory access modes in high-level synthesis” Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design (ICCAD’97), 1997, pp. 333 – 340.

[10] “The Stanford SUIF Compilation System”, version 1.1.2 Public domain software and documentation available at http://suif.stanford.edu.

[11] H. Schmit and D. Thomas, "Synthesis of Applications-Specific Memory Designs," IEEE Transactions on VLSI Systems, 5(1), Mar. 1997, pp. 101-111.

[12] M. Weinhardt and W. Luk “Memory Access Optimization and RAM interface for Pipeline Vectorization”, In Proc. of Symp. on Field Programmable Logic (FPL’99), Springer-Verlag, 1999, pp. 61-70.

[13] WildStar™ Reference Manual revision 4.0, Annapolis MicroSystems Inc., 1999.

[14] S.Wuytack, F.Catthoor, G.De Jong, and H.De Man. “Minimizing the required memory bandwidth in VLSI system realizations”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 7(4), Dec. 1999, pp. 433 –441.

[15] Xilinx, Inc. Virtex™ 2.5V Filed Programmable Gate Arrays Product Specification. DS003(v2.4), 2000.