AUTOMATED SYNTHESIS OF PIPELINED DESIGNS ON FPGAS FOR SIGNAL AND IMAGE PROCESSING APPLICATIONS DESCRIBED IN MATLAB®

Malay Haldar, Anshuman Nayak, Alok Choudhary and Prith Banerjee
MACH Design Systems, Inc.
Schaumberg, IL. USA.
www.MachDesignSystems.com

ABSTRACT

We present a compiler that takes high level algorithms described in MATLAB and generates an optimized hardware for an FPGA with external memory. A framework is described to detect and exploit opportunities to pipeline loops in an optimal way. Effectiveness of the framework is demonstrated by synthesizing some image and signal processing applications. Starting from the MATLAB description of the applications, hardware is synthesized that runs on a Xilinx XC4028. The synthesized designs are equivalent to manually optimized designs in performance.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have been recently used as an effective platform for implementing many image/signal processing applications. Though the concept of using FPGAs for custom computing evolved in the late 1980's, availability of commercial synthesis and placement tools for FPGAs has made reconfigurable computing more feasible. Current trends indicate that FPGAs have a faster growth of transistor density than even general processors. The implication of this is that there will be sufficient transistor budget for larger and more complex applications to be implemented on FPGAs. This has prompted many researchers to provide a hardware synthesis path from general purpose languages such as C/C++ [9], Java [8] and Matlab[1] [6]. Our choice of the MATLAB language is guided by the following facts - **(1)** MATLAB is extremely popular with the signal/image processing community and is easier and more intuitive to use than C/C++ **(2)** MATLAB has a rich set of libraries for signal/image processing functions which can be directly mapped to efficient libraries, thus making MATLAB very conducive to design reuse. **(3)** Large amounts of parallelism can be extracted from MATLAB programs with little or no dependency analysis, as opposed to complex dependency analysis required by languages like C/C++. In addition to meeting time-to-market pressures, the hardware synthesis frameworks must meet the required performance constraints. Many of the synthesis frameworks leverage off the optimization frameworks of the compilers of the target high level language. In particular, the Modula Pipeline [5] compiler focuses on using vectorizing compiler analysis of loops to synthesize efficient pipelined implementation in hardware. Their framework is based on the SUIF compiler framework and they take algorithms described in C. The effect of various loop optimizations including pipelining have been reported. In this paper, by pipelining we refer to pipelining loops in the hardware descriptions, either at the high level language or at the low level VHDL generated. This is different from pipelining hardware resources like adders and multipliers which involves using the particular resource for multiple computations in different stages. Pipelining loops is about using the hardware resources to compute multiple *iterations* in different stages. The main difference between the Modula compiler and our work is that while the Modula compiler attempts to pipeline at the level of the C statements, our work focuses on pipelining the VHDL generated from the MATLAB statements. This brings the optimizations closer to the hardware generated. On one hand this opens up more opportunities of optimizations and on the other hand one has to deal with the complexities of the hardware.

The contributions of this paper can be summarized as follows

- We present a framework for generating pipelined hardware on FPGAs from algorithms described in MATLAB.
- We present a pipelining algorithm that produces optimal designs for typical signal/image processing applications described in MATLAB.
- We demonstrate the effectiveness of our methodology by synthesizing hardware for some signal/image processing applications with the quality of the generated hardware comparable to manually designed hardware.

The paper is organized as follows. Section 2 gives an overview of our compiler framework and target architecture. Section 3 motivates the need to pipeline. Section 4 describes our pipelining algorithm. Section 5 presents our experimental results with the compiler and Section 6 talks about our conclusions.

2. OVERVIEW

The work presented in this paper is part of a MATLAB compiler for heterogeneous systems consisting of general purpose processors, embedded processors and FPGAs. Our compiler takes the description of a system in MATLAB and partitions it into software to be executed on general purpose and embedded processors and hardware to be mapped to FPGAs. In this paper we address the issues involved in generating an efficient hardware once the front-end of the compiler has partitioned the system into hardware and software. In particular we focus on the pipelining framework that does dependency analysis on the input algorithm and produces a pipelined design if possible. Figure 1 shows an overview of the framework. First the input MATLAB code is parsed to construct an Abstract Syntax Tree (AST). Since MATLAB variables do not have any notion of type-shape, a compiler phase infers the types of the variables and the dimensions of the matrices. The AST is then scalarized, where the operations on matrices are expanded out into loops. The AST is then levelized, where complex expressions are broken down into simpler expressions containing at most three operands. A dependency analysis phase infers the control and data dependency present in the AST. Finally a pipelining phase analyzes the loops to check if pipelining is possible, determines the initiation rate and produces a corresponding pipelined hardware description in VHDL. The output register transfer level (RTL) VHDL code is then passed through commercial synthesis and physical design tools to generate a netlist and bit-stream to configure the FPGA.

[1] MATLAB® is a registered trademark of Mathworks, Inc.

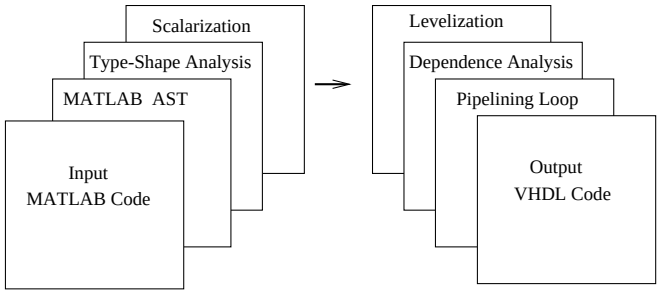

Figure 1. Overview of the synthesis framework.

Input : Dependence graph of loop body G(V,E).
Output : A pipelined hardware, if possible.
Algorithm :
M : Number of memory references in the loop body.
I : Initiation Rate.
MM : Next modulo memory number to be assigned.
CS : Current state number to be assigned.
IF reverse dependence edge then RETURN.
$M \leftarrow$ Number of memory references.
$I \leftarrow M$.
$MM \leftarrow 0, CS \leftarrow 0$
FOR all vertices $v \in V$, $state[v] \leftarrow \infty$
UNTIL $\exists$ vertex $v \in V$, s.t $state[v] = \infty$
  $state\_advanced \leftarrow false$
  FOR all vertices $v \in V$ not memory references
    IF(predecessor ready AND $state[v] = \infty$)
      $state[v] \leftarrow CS$
    ENDIF
  ENDFOR
  FOR all vertices $v \in V$ memory reference
    IF(predecessor ready AND $state[v] = \infty$)
      $state\_advanced \leftarrow true$
      IF($CS\%M \geq MM$)
        $state[v] \leftarrow CS + M - (CS\%M - MM)$;
      ELSE
        $state[v] \leftarrow CS + MM - CS\%M$
      ENDIF;
      $MM \leftarrow MM + 1$
      $CS \leftarrow state[v] + 1$
    ENDIF
  ENDFOR
  IF($state\_advanced = false$) $CS \leftarrow CS + 1$
ENDUNTIL
Produce code for prologue,epilogue and steady state
Calculate loop bounds
Produce VHDL code

Figure 2. Algorithm to find pipeline schedule of a loop body

2.1. Overview of the $WildChild^{TM}$ Architecture

Our compiler is designed to produce code for most current FPGA architectures. The compiler reads in a description of the FPGA board architecture and mechanisms to access the external memory which is used to produce the RTL VHDL for the particular FPGA board. Our current experimental results are on the $WildChild^{TM}$ FPGA board from Annapolis Micro Systems. It is a VME compatible board with eight $Xilinx4010$ FPGAs and one $Xilinx4028$ FPGA. In this paper we focus on our work regarding the generation of hardware for a single FPGA with an external memory. For that purpose we have used the $Xilinx4028$ FPGA having 1024 CLBs. The memory connected to the FPGA is 32 bit wide and contains 2^{18} addressable locations.

3. MOTIVATION FOR PIPELINING

The compiler generated hardware (without pipelining) involve a simple hardware synthesis mechanism without any optimizations. On an average, the compiler generated hardware are about five times slower than the manually optimized designs (Table 1). One of the critical optimizations that makes the manually optimized designs perform better is pipelining. All the benchmarks involve heavy accesses to memory as they operate on matrices that are mapped to an external memory. This is a typical characteristic of MATLAB programs, and signal/image processing applications in particular. In the compiler generated hardware each memory read requires five states, while a memory write requires three states. However, in manually designed hardware, most of the memory accesses are pipelined, effectively taking only one cycle per memory access. Most of these memory accesses occur in a loop, while performing operations on matrices. The motivation behind incorporating a pipelining phase in our compiler is to pipeline these loops containing memory accesses, such that the performance of the output hardware matches the performance of a manually designed hardware.

Table 1. Comparison of execution times for compiler generated and manually designed hardware. Times are in seconds.

	Matrix Mult.	Sobel	FIR	Motion Estimt.	Average Filter
Compiler	29.9	0.49	51.9	6.7	591
Manual	4.6	0.06	16.3	3.2	141

4. PIPELINING ALGORITHM

Given a pipelined loop body that is executed for I iterations and has s states and execution of an iteration is started every T cycles, the number of clock cycles required to complete the execution of the loop is given by $s + (I - 1) * T$. T is known as the initiation rate of the pipeline. For most practical loops in signal and image processing applications, s and T are much smaller than I. Thus the total number of clocks required to execute the loop is $\simeq I * T$. The goal of the pipelining phase is to determine a schedule with the smallest possible T so that the loop takes minimum clock cycles to complete.

4.1. Dependency Analysis

Given a series of nested loops, the pipelining algorithm requires the information whether the innermost loop body has any loop carried dependency from a previous iteration from a statement that appears after it in the loop body. For example, see Figure 3.

Dependencies of the form shown in Figure 3(d) prevent the execution of our pipelining algorithm. Fortunately, most of the innermost loops are obtained by scalarization of the MATLAB code, wherein matrix operations are expanded out into loops having only basic arithmetic operations. None of the matrix operations such as addition, multiplication, accumulation etc have such dependencies. Thus while scalarizing, this information is simply annotated to the loops, and no dependency analysis is required. However, for loops that were not obtained by scalarization, the GCD [1] test is performed, and the information is annotated to the loop.

4.2. Memory Constrained Initiation Rate

Apart from precedence constraints, the pipelined schedule must also satisfy resource constraints. If the loop body is pipelined with

```
for  i  = 1 : 10
  a[ i ]  =  b[ i ] + c[ i ];
end ;
```
(a)

```
for i = 1 : 10
  a[ i + 1 ] = c[ i] + 2 ;
  b[ i ] = a[ i ] + 3 ;
end ;
```
(b)

```
for i = 1 : 10
  s =  s  +  a[i] ;
end ;
```
(c)

```
for i = 1 : 10
 b[ i ] =  a[ i ]  + 3 ;
 a[ i + 1 ]  = c [ i ] + 2;
end ;
```
(d)

Figure 3. Examples of dependencies. The pipelining algorithm is applicable to cases (a),(b) and (c). However when dependencies of the (d) kind are present, the pipelining algorithm cannot be applied. (a) No loop carried dependencies. (b) Loop carried dependency, but with a statement that appears before in the loop body. (c) Loop carried dependency with the same statement in the loop body. (d) Loop carried dependency with a statement that appears after in the loop body.

an initiation rate of T clocks, then the states that are T clocks apart are executed simultaneously in the steady state of the pipeline and hence must have enough resources to execute concurrently. For example, if the initiation rate is 3, then the states $1, 4, 7, ..$ of the loop body execute together and so does the states $2, 5, 8, ...$ The general pipelining problem of finding the minimum initiation rate given a set of resources and a set of dependence relations of the input code is NP-complete [3]. Typical algorithms employ some heuristics that try to minimize the initiation interval iteratively. We present an algorithm that takes into account particular characteristics of the image/signal processing applications and the target architecture under consideration and produces optimum schedules without iterative searches. The critical assumptions that make the algorithm possible are

1. The arrays in the program are mapped onto an external memory. This assumption is valid as typical arrays that appear in signal and image processing domain are too large to fit on current FPGAs (even large ones) and have to be stored on an external memory.
2. The loops have references to these arrays and limited amount of computations. This is again a valid assumption as typical signal and image processing applications do relatively simple computations inside the loops.
3. The FPGA can access only one external memory at a time through a single external memory port. Most of the FPGA board architectures today fall under this category. In the case where multiple memory ports are available, the problem can be handled as a parallelization problem with distributed memory for which considerable literature is available.
4. There are no loop carried reverse dependency edges in the dependency graph of the loop body, i.e, a statement of the loop body does not have any loop carried dependence to a statement that appears after it. This condition holds for most of the image and signal processing benchmarks we have come across. Moreover, pipelining is a performance optimization. In the rare event where the loop body does contain reverse dependence edges, we cannot pipeline by the algorithm described here and we shall loose performance. But the output hardware will be always functionally correct.

The implication of these assumptions is that the initiation rate of the pipeline schedule is dominated by memory references. Since there can be only one memory reference in a state, if a loop body has R memory references, then the smallest possible initiation rate is R. This is the smallest possible initiation rate required so that none of the R memory references happen concurrently in the steady state of the pipeline. Given the assumptions stated above are satisfied, we can devise a simple algorithm to find a pipeline schedule which has initiation rate equal to the number of memory references, and hence is optimum. By optimum we refer to the fact that the pipeline schedule has the best initiation rate possible. The algorithm in Figure 2 is a variant of the ASAP scheduling algorithm [2].

Statements that are not memory references are scheduled as soon as possible. Statements that are memory references are placed as soon as possible, provided they do not conflict with any already placed memory referencing statement. This is done by assigning the states corresponding to memory reference statements such that their state numbers modulo the initiation rate are in increasing order and hence different. The algorithm in Figure 2 can be efficiently implemented by considering the vertices in topologically sorted order. Traditional pipelining techniques consider arbitrary dependence graphs and limited resources. Such a scenario is geared towards handling arbitrary constructs (like those present in C code) on a VLIW machine with predetermined number of resources (like adders and multipliers). As mentioned earlier, pipelining in such a scenario is NP-complete and typically heuristics are applied. In image and signal processing applications written in MATLAB, such arbitrary dependencies do not exist. Furthermore, FPGAs give the flexibility to decide the resources at compile time. This leads to great simplifications and we can devise a simple optimum algorithm in that case.

4.3. Other Issues

4.3.1. Overlapping live variables

The pipelining algorithm overlaps different iterations, causing an overlap of potentially live variables. To solve this we use the modulo variable expansion technique proposed in [4].

4.3.2. Conditional statements

Traditionally conditional statements evaluate a condition and jump to a state depending on the outcome of the condition. However, in a pipelined loop such jumps to statements cannot be permitted as the next state assignment is governed by the pipeline. Hence to pipeline loop bodies having conditional statements, the conditional statements are converted into a series of statements with predicates. This transforms the control dependence to data dependence, and the statements can be scheduled by the pipelining algorithm described in Section 4.2. This however poses the restriction that the condition variable cannot be modified inside the body of the conditional statement. This is true in most cases, and the compiler checks this condition before pipelining.

4.3.3. Turn around delay in $WildChild^{TM}$ Architecture

On the $WildChild^{TM}$ FPGA board, there must be a delay of at least one cycle between reads and writes to the memory. To handle this constraint, first a pipeline schedule is produced by applying the algorithm of Section 4.2. Then dummy data dependencies are introduced between successively scheduled memory accesses to keep their ordering intact. Then dummy memory access statements are introduced wherever a read is followed by a write, or vice versa. Finally, the pipelining algorithm is applied for the second time to get the final schedule. The dummy memory access statements serve as a gap between the reads and the writes.

4.3.4. Memory access cycle

Strictly speaking, the number of memory cycles needed to complete a read or write are indeterminate and depends on the handshake signals with the memory and the bus. Access to the mem-

ory can be arbitrated, and depending on the arbitration, a memory access can take any number of cycles. This makes pipelining multiple iterations with memory accesses very complex. To perform correctly, the pipeline must have facility to flush and recover, a mechanism similar to the pipelines in general-purpose processors. However, producing such pipelines is extremely complicated. Moreover, most users hand over the control of the memory to the FPGA board before starting an application on the board. Under such conditions, the access to memory take a predictable number of cycles. We also assume that the access to memory is under the FPGA board's control and memory accesses cannot be stalled in between. If an initiated memory access does not complete in the pre-determined number of clock cycles, the hardware raises an error flag.

5. EXPERIMENTAL RESULTS

In this section we present our experimental results. The benchmarks include matrix multiplication, FIR filter, Sobel edge detection algorithm, average filter and the motion estimation algorithm. For each benchmark, first a description of the algorithm in MATLAB was passed through our compiler without the pipelining optimization. Secondly, the description of the algorithm in MATLAB was passed through the compiler with the pipelining optimization. All the hardware were mapped to the Xilinx $XC4028$ FPGA with an external memory on the $WildChild^{TM}$ board from Annapolis Micro Systems. The results were compared with manually designed hardware for the benchmarks. Figure 4 shows the execution times for the three versions of each benchmark. For each

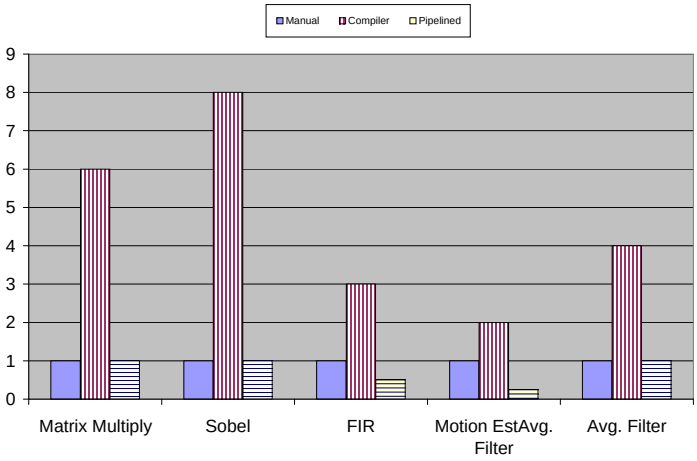

Figure 4. Ratio of execution times for compiler generated unpipelined hardware, compiler generated pipelined hardware, manually designed hardware (normalized to execution times of manually generated hardware).

benchmark, the execution times have been normalized against the execution time for the manually designed hardware. On the average, the execution time for the compiler generated hardware without pipelining is about 5 times slower than the manually designed hardware. And for all the benchmarks, the compiler generated hardware with the pipelining optimization matches the execution times of the manually designed hardware. The principal reasons behind the success of the pipelining algorithm are

1. Most of the benchmarks perform simple operations on matrices, which are relatively large. Hence the benchmarks are data dominated.
2. Each memory read takes five cycles to complete and each memory write takes three cycles. On the other hand operations such as addition, multiplication, shift etc take only one cycle.

Both these factors make the execution times dominated by the number of memory access. Our pipelining algorithm is designed to maximize the number of memory accesses per clock cycle, thus matching the performance of manually designed hardware for all the benchmarks. In fact for the motion estimation and FIR benchmarks we outperform the manually designed hardware by a factor of four and two respectively. The reason is that the innermost loop of the motion estimation algorithm is very simple with very little scope of *intra* -iteration optimizations and the number of states in the innermost loop is eight. On the other hand the compiler performs the pipelining optimization *across* the iterations, and since the innermost loop has only two memory accesses, it produces a loop with only two states, giving the performance a boost of four times. A similar reasoning holds for the FIR benchmark. For the motion estimation benchmark, the pipelined design generated by the compiler overlaps six iterations, with the states inside the loop having over 150 VHDL statements in a single state. We believe such complexity can only be handled by an automated tool.

6. CONCLUSIONS

In conclusion, we have presented a framework for producing pipelined designs on FPGAs from signal/image processing applications described in MATLAB. Although the general pipelining problem is NP-complete, taking into account particular characteristics of the applications and target architecture, we have devised a simple algorithm that produces optimum designs. We have proven the strength of the compiler by synthesizing hardware for a couple of image/signal processing algorithms that match the performance of manually designed hardware.

REFERENCES

[1] Steven S. Muchnick *Compiler Design Implementation*, Morgan Kaufmann Publishers, Inc.

[2] Giovanni De Micheli *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc.

[3] V. H. Allan, R. B. Jones, R. M. Lee and S. J. Allan,*Software Pipelining*, ACM Computing Surveys, Vol.27,No.3, September 1995.

[4] M. Lam,*Software Pipelining: An Effective Scheduling Technique for VLIW Machines* , Proc. Programming Language Design and Implementation, June 1988.

[5] M. Weinhardt and W. Luk, *Pipeline Vectorization for Reconfigurable Systems*, Proc. Field-Programmable Custom Computing Machines, April 1999.

[6] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *FPGA Hardware Synthesis from MATLAB*, 14th Intl. Conf. VLSI Design, Jan 2001.

[7] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *Scheduling Algorithms for Automated Synthesis of Pipelined Designs on FPGAs for Applications described in MATLAB*, Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems, CASES'2000, November 2000.

[8] B. L. Hutchings and B. E. Nelson,*Using General-Purpose Programming Languages for FPGA Design*, Proc. 37th Design Automation Conference, June 2000.

[9] G. De Micheli, *Hardware Synthesis from C/C++ Models*, Proc. Design, Automation and Test in Europe Conference and Exhibition, March 1999.