FPGA HARDWARE SYNTHESIS FROM MATLAB *

Malay Haldar, Anshuman Nayak, Nagraj Shenoy^{NU}, *Alok Choudhary and Prith Banerjee*
MACH Design Systems, Inc.
Schaumberg, IL. USA.

$_{NU}$ *Northwestern University*
Evanston, IL 60208-3118

ABSTRACT

Field Programmable Gate Arrays (FPGAs) have been recently used as an effective platform for implementing many image/signal processing applications. MATLAB is one of the most popular languages to model image/signal processing applications. We present the MATCH compiler that takes MATLAB as input and produces a hardware in RTL VHDL, which can be mapped to an FPGA using commercial CAD tools. This dramatically reduces the time to implement an application on an FPGA. We present results on some image and signal processing algorithms for which hardware was synthesized using our compiler for the Xilinx XC4028 FPGA with an external memory. We also present comparisons with manually designed hardwares for the applications. Our results indicate that FPGA hardware can be generated automatically reducing the design time from days to minutes, with the tradeoff that the automatially generated hardware is 5 times slower than the manually designed hardware.

1. INTRODUCTION

Reconfigurable Computing is characterized by hardware that can be reconfigured to implement specific functionality more suitable for specially tailored hardware than on a simple general purpose uniprocessor. Though the concept of using FPGAs for custom computing evolved in the late 1980's, certain recent advancements in FPGA technology has made reconfigurable computing more feasible. Current trends indicate that FPGAs have a faster growth of transistor density than even general processors. Newer FPGAs such as the Xilinx Vertex series have a density of around one-million gates which is projected to grow to ten million gates in the next few years. This implies that more complex designs can be put on the FPGAs. However, the absence of high level design tools can stretch the design time for complex designs, compromising the benefits of putting an application on FPGAs. This is the principal motivation behind providing a high level language such as MATLAB, C/C++, Java for designing and implementing complex systems in hardware. A lot of work in the research community has demonstrated the potential of achieving high performance from FPGAs for a range of applications [2]. However, performance improvements from such systems depends on the expertise of the hardware designer who must possess an accurate knowledge of the underlying FPGA board architecture. With new FPGA board architectures being proposed almost every year, designs coded for a particular board has to be rewritten for a newer architecture. Also, since it is difficult to understand the cycle-by-cycle behavior of millions of gates spanning multiple FPGA chips, writing such codes itself is very difficult. Though VHDL, which is a widely used hardware description language (HDL), gives us the ability to design hardware at a higher level of abstraction, the lack of effective debugging tools make coding in such HDLs very difficult. Further, a new designer has to go through the extra effort of learning a new language. Hence, a lot of effort has been made both in the industry and in academia to develop a compiler that would help the designer accomplish adequate performance improvements without the need to be involved in complex low level manipulations.

A lot of work [9, 10, 11, 12, 13, 15, 14] has been done in targeting existing programming languages like C, C++ and Java as the next hardware description language. Although C/C++ has been the popular choice as a language for synthesis, our choice of MATLAB is guided by the following factors

1. MATLAB is more intuitive and easier to use and learn. It provides a higher level of abstraction than C, and hence developmental time is orders of magnitude less.
2. MATLAB has a well defined rich set of library functions related to matrix manipulation and signal/image processing. Starting the hardware synthesis process from MATLAB enables the compiler to use optimized designs for these library functions. This gives an easy way to reuse design.
3. Absence of pointers and other complex data structures make optimizations easier and the compiler less error prone.
4. Extracting parallelism from the input program is the most critical step in synthesizing an optimal hardware. Compilers targeting automatic extraction of parallelism from C loops suffer from complex data dependency analysis. Other compilers shift the burden of specifying parallelism to users, and most programmers are not comfortable with parallel programming. On the other hand, most of the computation in MATLAB is expressed as matrix operations which are very amenable to parallelization. Users can specify parallelism by "vectorizing" their MATLAB code, which enables them to specify parallelism within the MATLAB framework.
5. MATLAB is a very popular language within the signal/digital processing community and our belief is that the next generation of embedded systems will be modeled in MATLAB. By providing a synthesis tool from MATLAB to hardware, we aim to close an important gap.

In this paper we present a compiler framework to produce hardware described in RTL VHDL from MATLAB. Section 2 describes the WildChild Architecture on which most of the our curren tsyn thesized hardwares ha ve been tested. Section 3 gives an ov erview of the compilation process. Section 4 and 5 describe the MATLAB AST and the VHDL AST, that are two intermediate representations used b y the compiler. Section 6 gives the details involv ed in transforming the MATLAB AST into a VHDL AST. In Section 7 we presen t some of our experimental results and Section 8 concludes the paper.

2. WILDCHILD ARCHITECTURE

Our MATLAB to VHDL compiler generates code for an y FPGA board architecture, the specifics of which are described in an architecture file. Figure 1 shows the WildChild board from Annapolis Micro Systems and is our target architecture on whic hmost of our curren t applications are tested. A brief description of the WildChild architecture is given belo w.

The WildChild custom computing engine is an integrated system consisting of the WildChild m ulti-FPGAreconfigurable hardware unit and a host that controls it and provides a conven tional soft w are in terface to the complete system. The WildChild FPGA unit is a VME-compatible board that is installed in a standard chassis along with a VME-compatible host computer. The WildChild board consists of 9 FPGAs of the popular Xilinx 4000 family. Eight of these are 4010 FPGAs with 400 CLBs (about 10000 gates) eac h and are identical. The ninth is a 4028 FPGA with 1024 CLBs (about 30000 gates). The nine FPGAs are arranged in a master-slave configuration as shown in Figure 1. Each FPGA on the board is connected to a pseudo dual-ported memory, which can be accessed by both the FPGA and the host. The memories connected to the 4028 master FPGA is 32 bits wide and contains 2^{18} addressable locations, while the memories connected to the 4010 slave FPGAs are 16 bits wide.

The architecture of the system enables high-throughput systolic computation using the 36-bit bus that connects each 4010 FPGA neighbors and also to the on-board FIFOs. There is also a crossbar netw ork that can be used to realize an y arbitrary interconnection of the FPGAs thus enabling irregular computations.

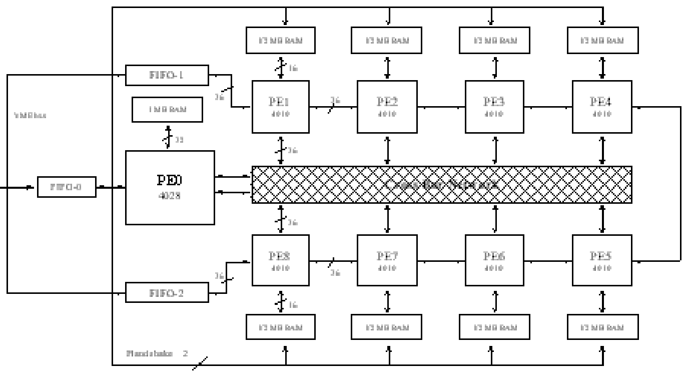


Figure 1. WildChild Board Architecture

3. SYNTHESIS FLOW

Figure 2 shows the overview of the synthesis process. The fron tend parses the input MATLAB program and builds a MATLAB AST. The input code may con tain directives [6]regarding the types, shapes and precision of arrays that cannot be inferred, whic hare attac hed to the AST nodes as annotations. This is followed b y a type-shape inference phase. MATLAB variables have no notion of type or shape. The type-shape phase analyzes the input program to infer the type and shape of the variables present whos type shape is not provided by directiv es. This is followed by a scalarization phase where the operation on matrices are expanded out into loops. In case optimized library functions are available for a particular operation, it is not scalarized and the library function is used instead. Scalarization is follow ed b y lev elization where a complex statement is brok en down into simpler statements. Scalarization facilitates VHDL code generation and optimizations. Most of the hardware related optimization are performed on the VHDL AST. A precision inference sc heme finds the minimum number of bits required to represen t each variable in the AST. Transformations are then performed on the AST to optimize it according to the memory accesses presen t in the program and characteristics of the external memory. This is follow ed by a phase to perform optimizations related to resources present and the opportunities of parallel execution and pipelining available. Finally a traversal of the optimized VHDL AST produces the output code. In this paper we restrict our focus to the basic compiler framework which involv es transforming the input MATLAB code into a hardware description in VHDL. The optimizations are part of our ongoing work.

4. THE MATLAB AST

The input MATLAB code is parsed based on a formal grammar and an abstract syntax tree (AST) is generated. The detailed grammar used for MATLAB 5.2 is giv en in [7]. Figure 3 shows a graphical view of the hierarchy captured b y the grammar. An example MATLAB code and the corresponding AST is also shown. Certain type-shape information can be input to the compiler with the help of directives. The information presented by the directives are parsed by the compiler and is used to annotate the AST. After the AST is constructed, the compiler invokes a series of phases. Each phase processes the AST either b y modifying it or annotating it with more information.

5. THE VHDL AST

VHDL serves as a hardware description language for simulation as well as synthesis. Most commercial tools today support only a subset of VHDL for synthesis. For example constructs like file operations, assertion statements, timing constructs are not supported by most synthesis tools. Also, certain tools require certain specific coding styles if accurate hardware is expected to be generated. F or example, *BehavioralCompiler*^{TM} [4] from Synopsys expects the user to specify the clock boundaries using specific *wait* statements while *Synplify*^{TM} [5] from Synplicity does not require it. Since we are interested in portability and expect the VHDL code generated by our compiler to be compatible with the good commercially available high-level syn thesis tools, w e operate on a subset of VHDL which is sensibly synthesized by most of the tools.

6. GENERATION OF VHDL AST FROM MATLAB AST

It was necessary to have a VHDL AST in spite of having an AST based on a MATLAB grammar so that not only the final VHDL code generation is simplified, but also that certain hardware related optimizations like memory pipelining are made possible. These optimizations require the notion of clock cycles which cannot be easily introduced in the MATLAB AST. While generating the VHDL AST, we

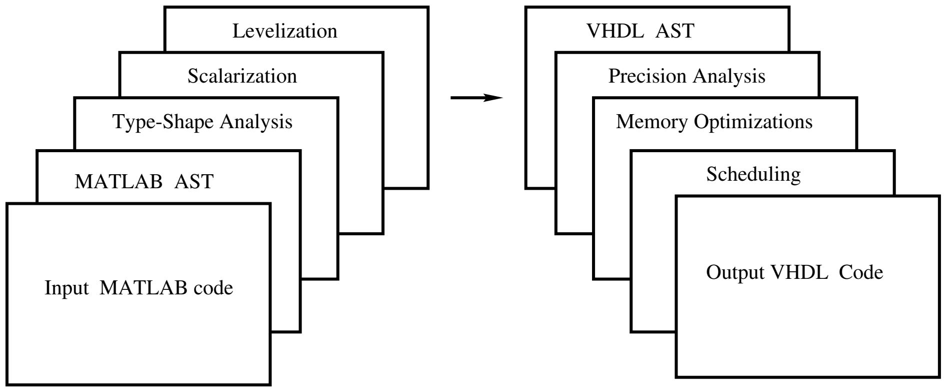
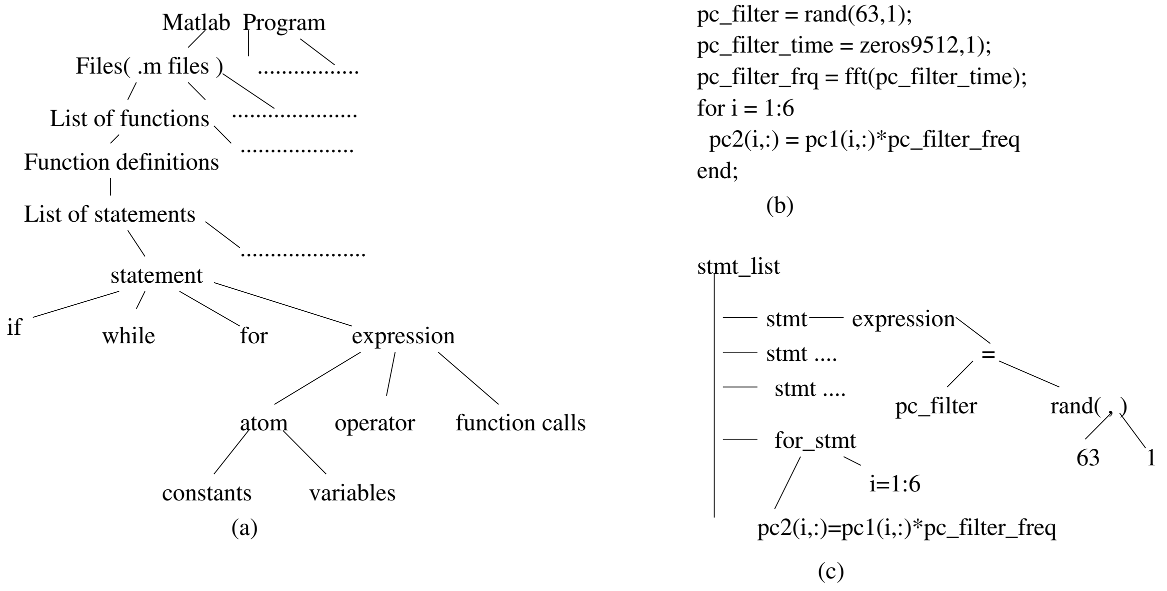


Figure 2. Synthesis flow.

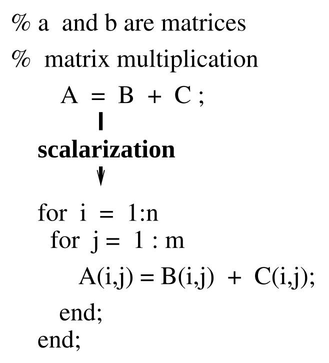


```
pc_filter = rand(63,1);
pc_filter_time = zeros9512,1);
pc_filter_frq = fft(pc_filter_time);
for i = 1:6
  pc2(i,:) = pc1(i,:)*pc_filter_freq
end;
```

Figure 3. Abstract Syntax T ree: (a) The hierarch y captured b y the formal grammar (b) A sample code snippet (c) Abridged syntax tree for the code snippet.

assume that the corresponding MATLAB AST is already scalarized. This is necessary as MATLAB is an array-based language.

6.1. Scalarization of the MATLAB AST



```
% a  and b are matrices
% matrix multiplication
     A = B + C ;
       |
   scalarization
       ↓
   for i = 1:n
     for j = 1 : m
        A(i,j) = B(i,j) + C(i,j);
     end;
   end;
```

Figure 4. Example of Scalarization

Scalarization of the MATLAB AST is necessary when the objective is to perform a source-to-source transformation to a target language that is statically typed and which only supports elemental operations. MATLAB is an array-based language with lots of built-in functions to support array operations. Hence, to generate a VHDL AST, it is necessary that the corresponding MATLAB AST is scalarized. But, scalarizing MATLAB vector constructs requires an accurate knowledge of shape and size of the arrays. MATLAB being a dynamically typed language, does not carry explicit basic data type and shape declarations. Hence, inferring an array's shape becomes a vital step. An example of scalarization is shown in Figure 4, where a two dimensional matrix addition is scalarized. To produce the scalarized code it is necessary to know the dimensions of matrices A, B and C and also their sizes (the values of m and n).

We have established a static inferencing mechanism to establish at compile-time, what these attributes will be at run-time. In [8] a framework is described for symbolically describing the shape of a MATLAB expression at compile-time using a methodology based on systematic matrix formulations. The representation exactly captures the shape that the expression assumes during execution.

6.2. Levelization of the MATLAB AST

The scalarized MATLAB AST is subjected to a levelization phase which modifies the AST to have statements with only three operand format (Figure 5). The advantage of levelization is that the different operators can be scheduled independently, guided by the optimization algorithms. For example the operations can be spread across different states, so that an optimal clock frequency can be obtained.

Levelization also enables optimizations like pipelining, operator chaining, loop body scheduling to enhance the performance of the output hardware. Also, since statements having large number of operations are broken down to a series of statements having only one operation, resources can be reused as these smaller statements can be distributed across different states.

Details of the mechanism of generating state machine descriptions from loops, conditionals and function calls are discussed in [1].

6.3. Handling Memory Accesses

Currently the compiler maps all arrays to an external memory and instantiates registers on the FPGA for scalars. The

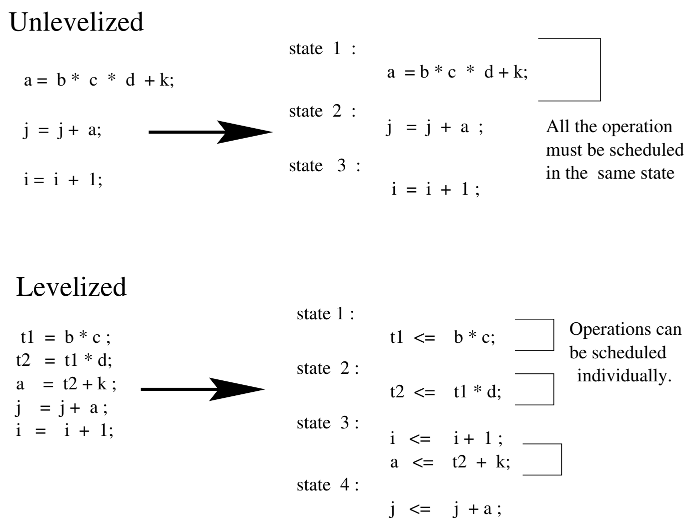
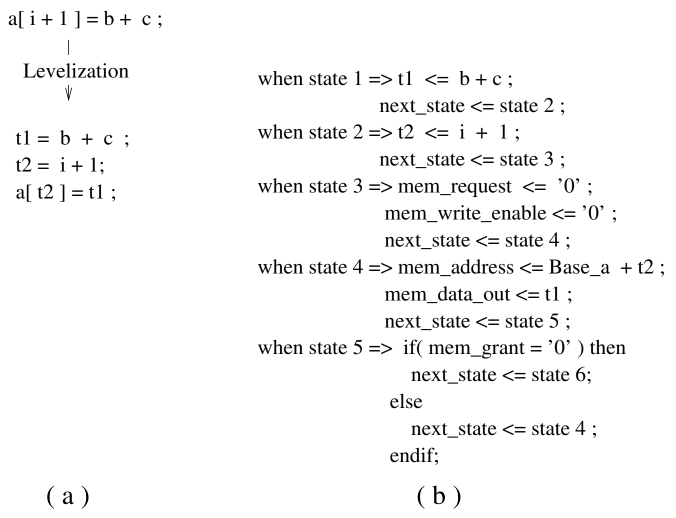


Figure 5. Example of Levelization .

levelization phase ensures that a statement has at most one memory access with no other associated operation. The exact mechanism and signals involved in accessing the memory is specified in a file that is read by the compiler. The compiler then uses this information to produce the states necessary to read/write memory corresponding to each array access that appears in the levelized and scalarized MATLAB code. Figure 6 shows an example.



```
a[ i + 1 ] = b + c ;
     |
 Levelization
     ↓
t1 = b + c ;
t2 = i + 1;
a[ t2 ] = t1 ;

      ( a )
```

```
when state 1 => t1 <= b + c ;
                next_state <= state 2 ;
when state 2 => t2 <= i + 1 ;
                next_state <= state 3 ;
when state 3 => mem_request <= '0' ;
                mem_write_enable <= '0' ;
                next_state <= state 4 ;
when state 4 => mem_address <= Base_a + t2 ;
                mem_data_out <= t1 ;
                next_state <= state 5 ;
when state 5 => if( mem_grant = '0' ) then
                   next_state <= state 6;
                else
                   next_state <= state 4 ;
                endif;

      ( b )
```

Figure 6. (a) Array statement in MATLAB and it's levelization (b) VHDL corresponding to the MATLAB code. The signals $mem\_request, mem\_data\_out, mem\_grant, mem\_write\_enable$ **and their particular states and assignments are specified in an external file read by the compiler.** $Base\_a$ **is a constant denoting the starting address of the array** a **in memory.**

A simple tree traversal of the VHDL AST then produces the output VHDL code.

7. EXPERIMENTAL RESULTS

In this section we present some of our experimental results. The benchmarks presented include matrix multiplication, FIR filter, sobel edge detection algorithm, average filter and

Table 1. Experimental Results : T - Execution time in secs. F - Synthesized frequency in MHz. S - Number of states in the inner most loop. Manual : Hand optimized designs. Compiler : Compiler generated designs.

	Matrix Mult.			Sobel			FIR			Motion Est.			Av erage Filter		
	T	F	S	T	F	S	T	F	S	T	F	S	T	F	S
Manual	4.6	19.9	8	0.06	18.6	14	16.35	20.1	10	3.28	15.0	11	14168.3	13.1	6
Compiler	19.3	9.8	23	0.72	9.9	116	51.98	9.8	23	6.79	17.1	29	59132.7	14.2	32

the motion estimation algorithm. The matrix multiplication algorithm multiplies t w o input matrices. Matrix multiplication is at the core of many signal and image processing algorithms. FIR (Finite Impulse Response) filter is very important in the signal processing domain as it suggests a system that passes certain frequency components and rejects all other frequencies. Sobel edge detection algorithm takes an input image and performs a tw o dimensional linear convolution with 3×3 kernels. The av erage filter takes an input image and for each pixel of the image computes the average v alues of the pixels in the neighborhood. A comparison is then made betw een the pixel and the average value. The motion estimation algorithm is used to low er the bandwidth requirement in video transmission b y comparing blocks of the current frame with blocks of the previous frame and selecting a best match. We have chosen these applications as benc hmarks as they are represen tativ e of the applications that are most suitable for implementation in hardware.

F or each benc hmark, a description of the algorithm in MATLAB w as written and passed through the MATCH compiler. The hardware generated was v erified in t w o ways

- By simulating the generated hardware using a VHDL simulator and correlating the results with the output of the algorithm executed in the MATLAB interpreter.
- By manually designing hardwares for eac h of the benchmarks and correlating the output of the compiler generated hardware with the manually designed ones. For this purpose the hardwares w ere run b y configuring the X C4028 FPGA.

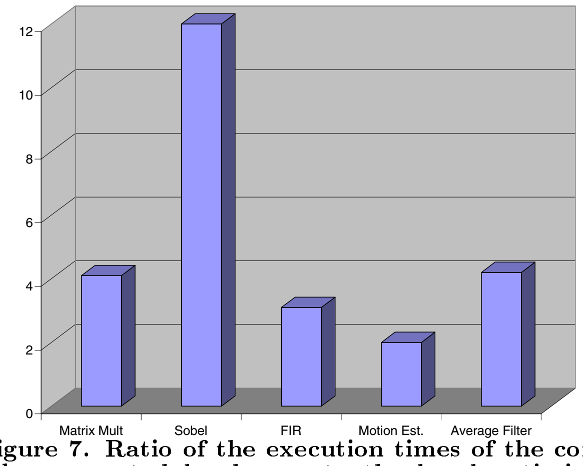


Figure 7. Ratio of the execution times of the compiler generated hardwares to the hand optimized hardwares.

The manually designed hardwares were highly optimized for performance. A comparison of the performance of the compiler generated hardware with the manually designed hardware indicates the tradeoff in volved betw een design time and design qualit y. It also gives an idea about the performance gains expected by implementing optimization techniques in the compiler. Table 1 shows a summary of the experimental results. Figure 7 shows the ratio of the execution times of the compiler generated hardwares with the manually designed ones. As can be seen, the compiler generated hardwares w ere on the average 5 times slower than the manually optimized designs. The total execution time to complete a function is given by the number of state transitions required to complete the function times the clock period. The total n umber of state transitions required to complete a function is dominated by the number of states in the loop bodies, as most of the time is spent in executing loops. Figure 8 shows the ratio of the number of states in the inner most loop for the compiler generated and the manually designed hardware for each benchmark. On the av erage, the number of states produced by the compiler were 3 times more when compared to manually designed hardware. The largest con tribution to increased states came from memory access statements. The compiler generated a sequence to 4 statements for every memory read and a sequence of 3 statements for every memory write. On the other hand, the memory accesses in the manually designed hardwares w ere pipelined, effectiv ely making memory accesses tak e only 1 cycle. Also some of the memory accesses w ere eliminated in manually designed hardwares b y reusing data accessed in previous iterations. The effect is most pronounced in the *sobel* benchmark, whic h has lots of memory reads. Automatically pipelining the memory accesses and caching data fetched require sophisticated data and control flow analysis and is the focus of our future research.

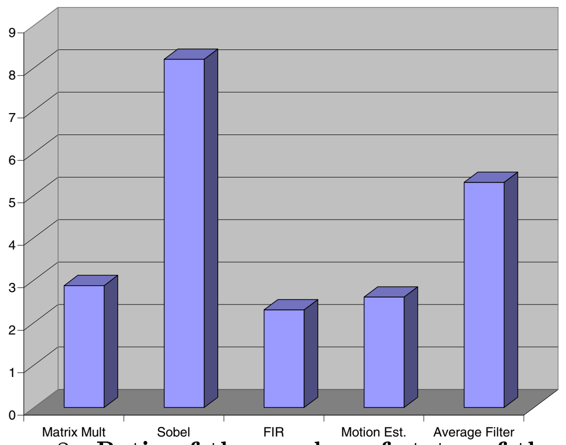


Figure 8. Ratio of the n umber of states of the innermost loop of the compiler generated hardwares to the hand optimized hardwares.

The second factor which contributes to the increased execution time of compiler generated hardware is the larger clock period. The clock period is dictated by the place and route CAD tools. The synthesized clock period cannot be directly controlled b y the compiler, it can only be set as a target. Ho w ever, the amount of resources used on the

FPGA has a high correlation with the synthesized clock period, which can be influenced by the compiler. On an average, the compiler generated hardware used twice the resources used by manually designed hardware. Figure 9 shows the synthesized clock periods for the compiler generated and manually designed hardware for each of the benchmarks. The lower clock period (and lower resource utilization) for the manually designed hardware was due to the fact that manually designed hardware included clever reuse of computation and resources. Again, some complex compiler analysis is required to match the manually designed hardwares.

The main advantage of the compiler is that it reduces design time from months to minutes, and enables much more complex designs by providing a higher level of abstraction. The focus of our future work is to reduce the number of clocks required and increase the synthesized frequency so that the execution time is comparable to manually optimized designs.

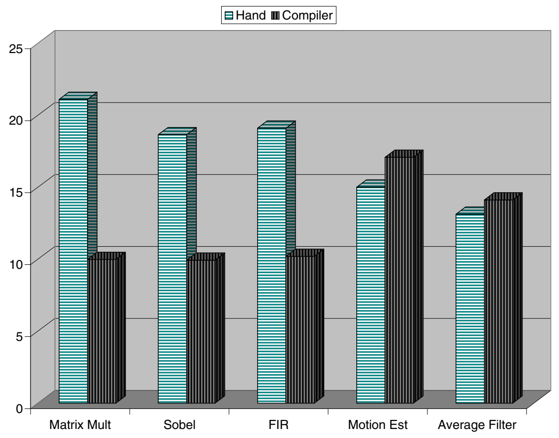


Figure 9. The synthesized frequency from the place and route tools. The frequencies are in MHz.

8. CONCLUSIONS AND FUTURE WORK

In conclusion, we have presented a compiler that takes algorithms described in MATLAB as input, and produces a hardware description in VHDL for the algorithm, suitable to run on an FPGA with an external memory. We have presented methodologies to generate hardware corresponding to high level constructs present in MATLAB such as loops, conditional statements and function calls. We have demonstrated the effectiveness of our compiler by generating hardware for some signal and image processing algorithms described in MATLAB. The generated hardwares have been verified functionally by running on an Xilinx XC4028 FPGA with an external memory.

The primary focus of our ongoing research is to improve the quality of the designs produced by the compiler. In this direction we are targeting some critical optimizations :

- Pipelining : Our attempt is to pipeline the loops in the input program and pipeline the accesses to the external memory.
- Caching : Synthesize small on-chip caches on the FPGA to reduce traffic to the external memory.
- Memory Packing : All accesses to the memory fetches 32 bits. For data types that require 16 or less bits, pack more than one data accesses into one memory access.
- Precision Analysis : Find the minimum number of bits necessary to represent a variable and produce hardware accordingly.
- Parallelizing : Automatically parallelizing the input program and map it to multiple FPGAs.

With these optimizations we expect that the hardware generated by the compiler will be very close to manually designed hardware in performance.

REFERENCES

[1] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary and P. Banerjee, *FPGA Hardware Synthesis from MATLAB*, Techinal Report, Northwesetrn University, September 2000.

[2] Villasenor, J., Mangione-Smith, W. H. *Configurable Computing*, Scientific American, June 1997, pp. 66-71

[3] Peter J. Ashenden *The Designers's Guide To VHDL*

[4] *Behavioral Compiler Coding Styles*, Synopsis Inc.

[5] *Synplify and Synplify Pro Reference Manual*, Synplicity Inc.

[6] Prith Banerjee, Nagraj Shenoy, Alok Choudhary, Scott Hauck, Chris Bachmann, Malay Haldar, Pramod Joisha, Alex Jones, Abhay Kanhare, Anshuman Nayak, Suresh Periyacheri, Mike Walkden and David Zaretsky, *A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems*, Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'00, April 2000.

[7] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U.Nagaraj Shenoy and Alok Choudhary, *The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler*, Technical Report, Center for Parallel and Distributed Computing, Northwestern University, CPDC-TR-9909-017, Sep. 1999.

[8] P.G. Joisha, A.Kanhere, U.N. Shenoy, A. Choudhary, P. Banerjee *An Algebraic Framework for Array Shape Inferencing in MATLAB*

[9] A. Duncan, D. Hendry, and P. Gray, *An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems*, Proc. Field-Programmable Custom Computing Machines, April 1998.

[10] J. Babb, M. Rinard, C.A. Moritz, W. Lee, M. Frank, R. Barua, S. Amarasinghe *Parallelizing Applications into Silicon*, FCCM 1999

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

[12] J. Hammes, B. Rinker, W. Bohm and W. Najjar, *Cameron: High Level Language Compilation for Reconfigurable Systems*, Proc. Parallel Architectures and Compilation Techniques (PACT'99), October 1999.

[13] M. Gokhale, J. Stone, J. Arnold and M. Kalinowski, *Stream-Oriented FPGA Computing in the Streams-C High Level Language*, Proc. Field-Programmable Custom Computing Machines, April 2000.

[14] R. Helaihel and K. Olukotun, *Java as a Specification Language for Hardware-Software Systems*, Proc. International Conference on Computer-Aided Design, pp. 690-697. November 1997.

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