A Study of Design Efficiency with a High-Level Language for FPGAs

Zain-ul-Abdin and Bertil Svensson
Centre for Research on Embedded Systems (CERES),
Halmstad University, Halmstad, Sweden.
{Zain-ul-Abdin,Bertil.Svensson}@ide.hh.se

Abstract

Over the years reconfigurable computing devices such as FPGAs have evolved from gate-level glue logic to complex reprogrammable processing architectures. However, the tools used for mapping computations to such architectures still require the knowledge about architectural details of the target device to extract efficiency.

A study of the Mobius language and tools is presented in this paper, with a focus on generated hardware performance. A number of streaming and memory-intensive applications have been developed and the results have been compared with the corresponding implementations in VHDL and a behavioral hardware description language. Based upon experimental evidences, it is concluded that Mobius, a minimal parallel processing language targeted for reconfigurable architectures, enhances productivity in terms of design time and code maintainability without considerably compromising performance and resources.

1. Introduction and Motivation

Field Programmable Gate Arrays (FPGAs) are drawing the attention of an increasingly wide audience because of their double performance advantages, not only providing better price per performance but also consuming far less power per computation, when compared to ordinary CPUs. Another key factor is that, over the last decade, the enhancements in FPGA density and speed have not only followed but somewhat exceeded Moore's law [1]. The possibility of run-time reconfiguration by which FPGAs can be remotely configured has also led to the growing realization of algorithms in silicon instead of using general purpose processors.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

The target applications for FPGAs range from signal, image and video processing to communication, cryptography and pattern matching.

Despite these merits, FPGAs have still achieved limited applicability because of the difficulty in describing complex algorithms in the hardware oriented design methods such as structural Hardware Description Languages (HDLs) and schematic based tools. They require a more rigorous development process, involving manpower training beyond application level, resulting in an increase in development cost and time-to-market. In addition, the clock speed of FPGAs is typically ten times lower than that of general purpose processors, and due to this they need to exploit large scale parallelism in algorithms to compete with these devices. The techniques used to obtain maximum performance include keeping the design synchronous, implementing pipelining, minimizing logic cell fan out, duplicating logic in critical paths and handcrafting the critical sections of the design.

Thus there is a dire need for a higher layer of abstraction, as structural HDL is well below the level of classical application programming, in order to hide the low level hardware complexities from the developer. One such high-level behavioral language is Mobius, which allows the applications to be described as a composition of concurrent processes. The individual processes communicate among each other by using typed channels. This approach hides the intricacies of low-level details from the application programmer. In the rest of the paper we refer to structural HDL as HDL.

In this paper we present an experimental study of the Mobius language and tools, with a focus on the efficiency of the generated HDL code. This work is part of our ongoing research whose initial studies are presented here. A comparison has been made between results obtained from implementations built in Mobius, VHDL and Streams-C.

The rest of the paper is organized as follows: Section 2 presents some of the similar languages. Section 3 provides a review of the Mobius language. Section 4 describes the Mobius design flow and tools. The results of

implementations are discussed in Section 5, and the conclusions drawn from the study are presented in Section 6.

2. High-level Languages for Reconfigurable computing

There have been a number of initiatives taken in both the industry and the academia to address the requirement of high-level languages for reconfigurable silicon devices. A few of them are described here.

Handel-C is a high-level language with ANSI-C like syntax used to program gate level reconfigurable hardware [2]. It supports behavioral descriptions with parallel processing statements (`par`) and constructs to offer communication between parallel elements. Handel-C is being used for compilation for synchronous hardware and inherits sequential behaviors.

Streams-C, a project initiated by Los Alamos National Laboratory, is based on the Communicating Sequential Processes (CSP) model for communication between processes and used for stream-oriented FPGA applications. The Streams-C implementation consists of annotations and library function calls to stream module. The annotations define the process, stream, and signal. The Abstract Syntax Tree (AST) consisting of sequences of the basic and pipelined data path blocks, is generated by the compiler and the compiler analyses the AST for partitioning of control and data flow blocks [3]. Streams-C, which is a subset of ANSI-C, lacks the support for two dimensional arrays.

SA-C, a single assignment synthesizable language is one of the academic projects. SA-C is designed as functional in nature, not intended as a stand-alone language [4]. The compiler constructs the data flow graphs of the application before generating FPGA configurations. SA-C eliminates the need for pointers as used for de-referencing in C, and does not support recursion and while-loops.

SPARK is a high-level synthesis tool, which translates C code to VHDL [5]. The compiler applies a number of optimizations such as loop unrolling and code motion to improve the synthesis results of control intensive computational blocks. SPARK, specialized for multimedia and image processing applications, cannot perform optimizations on input data reuse.

Compiler for Application Specific Hardware (CASH) converts the C source code into Pegasus [6], which is an intermediate representation in the form of a dataflow machine. It exploits instruction level parallelism by applying prediction and speculation. Other optimizations include pipeline balancing and data width resizing.

The reason for selecting Mobius for experimentation is that it is one of the recently emerging languages for reconfigurable computing where concurrency is inherent. This facilitates the use of parallelism to structure programs, thus enhancing code modularization and reuse. Most of the other languages discussed have a C like syntax and expose parallelism by either providing statement-level parallel constructs or relying on the compiler to maximize parallelism.

3. Mobius language

Mobius is a tiny, domain specific, concurrent programming language with CSP-based [7] interprocess communication and synchronization methodologies using handshaking [8]. It has a Pascal like syntax with bit specific and Occam like extensions, suitable for both fine-grained and coarse-grained architectures.

The hierarchical modules in Mobius are composed of procedures and functions. The processes execute concurrently and communicate with each other through message passing unidirectional channels. A channel consists of *req*, *ack* and *data* signals and provides unbuffered, point-to-point communication. The data width can be user defined. An active port can either initiate a read or a write. The data types supported by Mobius are `integer(n)`, `float(n,m)`, `fixed(n,m)`, `Boolean`, `channels`, `arrays`, `records`, `bit-select` and `timers`. The data type can itself be an array in order to construct multi-dimensional arrays and every type conversion is explicit in Mobius. Furthermore, Mobius has built-in support for floating point operators such as add, subtract, multiply, divide, invert, square-root and comparison. The size of mantissa and exponent is parameterizable, depending on the constraints for area conservation or desired precision. There are also constructs to manage the flow control such as `seq` (sequential block), `par` (parallel block), `alt` (alternative communication), `while`, `if/else`, `for` and `repeat`. The non-deterministic `alt` construct can be used to choose between multiple channel events.

4. Mobius Compiler

Figure 1 shows the complete design flow used for developing in Mobius [9]. The tool-chain consists of three components, namely Simulator, SW compiler and the HW compiler.

There is also a prototype debugger derived from the simulator. Mobius SW compiler generates ANSI-C compatible code with inherent run time support. Any application developed in Mobius can be tested at the functional level by using the simulator, which is a transaction modeler lacking timing information.

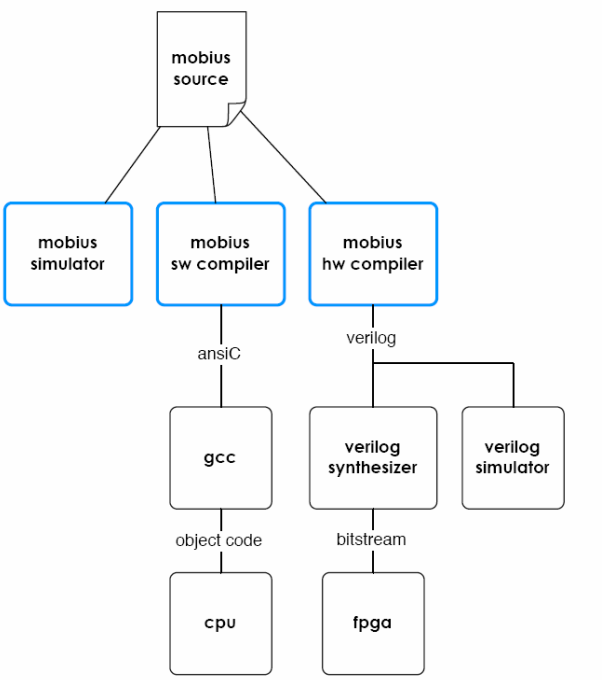

Figure 1 Mobius design flow [9].

However, there are certain discrepancies regarding the generated HDL code, for example, the non-deterministic `alt` scheduling, the fact that the simulator uses double precision format for floating point operations irrespective of what is specified in the source code, and it also enables recursion and hierarchical scoping, not supported in the generated code.

The major focus in this study has been laid on the HW compiler, which generates the synthesizable Verilog/VHDL code with embedded test bench. The corresponding HDL code can then be synthesized or simulated by using any third-party tool to generate the bitstream for the FPGA. The Mobius implementation has been translated to data flow graphs consisting of handshake primitives, which are interconnected by handshaking channels [9]. The data flow graphs give an abstract view of the generated hardware logic. Mobius uses a 4-phase handshaking protocol for data transfers over channels. The data channels are further classified as either push or pull channel and the control channel is only used for synchronization.

The use of the handshaking primitives automatically manages the flow control and provides automatic elastic pipelining. The various handshake primitives along with their respective resultant circuits are shown in Figure 2 . All of these handshake primitives provide events to sequence the actions. Each handshake primitive contains a passive sync activation port and optionally a number of active sync output activation ports. In the handshake graph, the filled dots or arrows represent active ports, whereas the hollow circles indicate passive ports and the arrow heads indicate the direction of the data flow. By examining the equivalent circuits, it can be deduced that a sequential primitive translates to a sequencer with the *ack (a)* signal of the first statement connected to the *req (r)* signal of the following statement. A parallel block constitutes a fork-join where fork allows the concurrent activation of events for the child statements. The next three primitives allow the control channel to synchronize the data transfers. An if-else block is translated to a demultiplexer followed by a multiplexer and the mapping of input values to the choice of output port is based on the condition. A do-loop is decomposed into a mux-demux pair operating in a loop so that as long as the control input of the multiplexer is asserted, it activates the following statement. A transfer primitive is used to implement assignment by transferring a data value from the pull data path and pushing it towards the push data path and is implemented as wires.

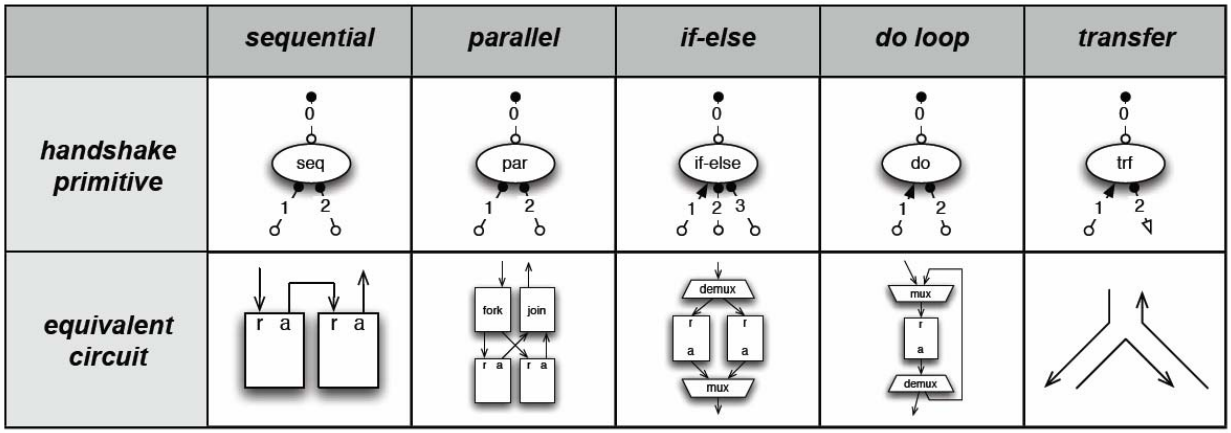

Figure 2 Mobius handshake primitives and circuits [9].

Generally, the Mobius compiler generated VHDL code can be divided into four sections:

- Definition of top-level package
- Definition of Mobius specific primitives
- Definition of application procedures and functions
- Definition of test bench

It is possible for third-party IP cores to interact with the Mobius generated VHDL, provided they are compliant with the Mobius handshaking protocol.

There is so far no automatic resource sharing implemented by the compiler, so it is up to the developer to build a resource sharing architecture. A client-server model can be implemented to share the resources among multiple clients using a time-multiplexing technique. An important factor considered while implementing such a structure is that the shared resource should be large and complex enough to justify the additional logic used.

5. Experimental Results and Discussion

The experimental results presented in this section are compared based on the area utilization and maximum achievable frequency characteristics of the generated code, as well as on the development effort involved.

Comparison of Mobius vs Hand coded implementations

In this section the implementation results for three benchmark applications are presented, as summarized in Table 1. A comparison has been made between the Mobius results and those of manually optimized versions provided by Xilinx [10] [11]. The results have been obtained from Xilinx ISE 8.2i synthesis tool based on the XCV300 device.

The first example is an FPGA based digital filter. These are gaining interest because of their flexibility and performance. The hand-crafted implementation consists of an 8-tap transformed form Finite Impulse Response (FIR) filter with Constant Coefficient Multipliers (KCM), adders, registers and a delay-locked loop [10]. The fixed-point representation used for input samples is 16-bits and that of coefficients is signed 14-bits. The Mobius implementation of the FIR filter consists of inelastic pipelining to achieve better throughput, but the disadvantage is that the first few samples are to be discarded till the pipeline is full. Once the pipeline is filled it provides an output sample per clock cycle. The area utilization results of the Mobius design are better than those of the hand coded, whereas the maximum speed is 30% lower. This can be explained in the context of use of KCMs by the hand coded implementation where the partial products are stored in lookup tables (LUTs), which is highly efficient.

Next, implementation results of the complex Fast Fourier Transform (FFT) are discussed. The implementation consists of a radix-2 decimation in time algorithm. The design in Mobius receives inputs in natural order and provides output samples in natural order too. The bit reversal stages for the samples as well as the weights are embedded in the FFT process. Since the run-time calculation of sine and cosine values are not yet supported in Mobius, the pre-computed weights are passed along with the input data in a stream. The HDL implementation also uses pre-computed weights. In order to conserve the area utilization the algorithm operates on arrays for real and imaginary parts, but the butterfly computations have been parallelized. It can be deduced from the results that the implementation described at the gate level occupies less area and generates better performance because the movement of data into and out of the butterfly processing unit and computation are overlapped by the use of dual port RAM. Thus the FFT unit never stalls while waiting for an I/O operation, thus providing higher throughput rate.

Table 1 Resource Utilization & Code Complexity of Mobius and Hand coded applications.

Benchmarks	No. of Slices	No. of Slice FFs	No. of 4-Input LUTs	No. Of Bonded IOBs	Max. Frequency MHz	Lines of Code
8-tap FIR (Mobius)	559	279	994	38	48.36	61
8-tap FIR (VHDL)	672	811	948	56	68.04	480
8-pt. FFT (Mobius)	785	331	1388	64	48.89	121
8-pt. FFT (VHDL)	533	670	957	67	71.297	1759
2D-DCT (Mobius)	631	190	1117	25	43.97	112
2D-DCT (VHDL)	1573	1608	1876	23	61.02	866

```
for i:=0:(mrows-1) do
seq
        for j:=0:(mcols-1) do
        seq
                tmp2 := 0;
                for k:=0:(mrows-1) do
                seq
                        itmp1:= indtyp((i*mcols)+k);
                        itmp2:= indtyp((k*mcols)+j);
                        tmp1 := xmat[itmp1] * ctmat[itmp2];
                        tmp2 := tmp2 + tmp1
                end;
                itmp3:= indtyp((i*mcols)+j);
                zmat[itmp3] := tmp2
        end
end;
```

Figure 2 Mobius implementation of 1D-DCT using Vector processing.

The third implementation is a Two-Dimensional Discrete Cosine Transform (2D-DCT) employed in video compression [11]. The architecture adopted for implementing 2D-DCT in the HDL version consists of cascaded stages of an 8-point one-dimensional DCT, a double RAM buffer and another 1D-DCT. The 1D-DCT has been implemented by using vector processing which employs parallelized multipliers, giving a regular structure with simple control and interconnect to provide improved performance and area utilization. The 2D-DCT in Mobius has been implemented in the form of matrices by using arrays, as shown in Figure 2. The `indtyp` is a new type, and `xmat`, `zmat` are the array matrices defined previously in the complete code. The cosine and inverse cosine transposed coefficients are stored as constants in arrays. The input matrix is passed through an input channel and vector processing has been performed in two steps operating on the arrays. The results are passed through the output channel to the test bench. The use of arrays results in the instantiation of block RAM, thus conserving the slice resources considerably compared to the hand written implementation, but the maximum speed achieved is lower due to operating in a sequential loop. The Mobius implementation has an initial latency of 234 clock cycles as compared to 92 cycles for the VHDL implementation [11] (the performance mentioned in the application note can not be reproduced by us because of the unavailability of proper simulation tools). The latency of the Mobius implementation can be improved by implementing pipelining, but it will result in increased area resources.

The number of lines of code may be used as a measure of the complexity involved in debugging and development of the implementations. With this measure the relative complexity in a VHDL design is 8 to 14 times higher than that in a Mobius based one.

Comparison of Mobius vs Streams-C

In this section, the synthesis results are compared for implementations performed in Mobius and in the Streams-C [3] language. The results presented are targeted for the Xilinx XCV1000 device and are shown in Table 2.

A Polyphase filter is a multirate filter employed along with FFT in signal detection to extract RF signal subbands from noisy environments [3]. The design of a polyphase filter bank is based on a lowpass FIR filter with symmetric coefficients. The input stream consists of 8-bit unsigned encoded data, while the coefficients are 12-bit unsigned values. A bank of four polyphase filters is mapped on the FPGA. The results show that the Mobius implementation not only acquires less area utilization, the maximum speed of the Mobius design is also much higher. This can be explained by the fact that the Streams-C compiler is unable to pipeline the computational blocks in the algorithm having the conditional control flow. In contrast, fine-grained parallelism has been achieved efficiently in the Mobius design by the use of `par` statements within the conditional block, as shown in Figure 3. It is worth mentioning here that in terms of speed the Mobius results even exceed the hand coded version [3].

The final example is a relatively complex Pixel Purity Index (PPI) algorithm [12]. The PPI algorithm has been used in the analysis of hyper spectral images. A large number of random D-dimensional vectors called skewers are generated.

Table 2 Performance results of Mobius and Streams-C applications.

Benchmarks	Area Slices	Speed MHz	Development Time in Weeks
Polyphase filter (Mobius)	<1%	98.2	1/2
Polyphase filter (Streams-C)	1%	40	1/2
Pixel Purity Index (Mobius)	2%	54	1
Pixel Purity Index (Streams-C)	6%	40	1

```
while true do
seq
    cu ? u;
    par
          m1 := datat(b1*u);
         m2 := datat(b2*u);
         m3 := datat(b3*u);
         m4 := datat(b4*u)
    end;
    if (even=1) then
      par
           ot := e1+m1;
           e1 := e2+m2;
           e2 := e3+m3;
           e3 := m4;
           even := 0
      end
    else
      par
           ot := o1+m4;
           o1 := o2+m3;
           o2 := o3+m2;
           o3 := m1;
           even := 1
      end;
    cy ! ot
end
```

Figure 3 The Polyphase filter bank in Mobius.

For each skewer every data point is projected onto the skewer and the corresponding location along the skewer is noted. The data points corresponding to the extrema are listed and the number of times a specific pixel is placed on the list is also noted. The one with the highest number gives the most pure pixel. The computationally most expensive part is the calculation of the dot products between the skewers and the pixels, and this part has great potential for parallelization. The individual dot products are independent and can be executed concurrently. The Mobius implementation consists of 6144 pixels, 4096 skewers and 512 spectral bands. Thus the complexity of the implemented PPI algorithm is O(8×512×12). It can be deduced from the results that Mobius produces much better resource utilization as well as enhanced performance in terms of speed. The Streams-C generated results lag due to the inability to pipeline loops that have conditional constructs and due to the lack of capability of having arrays of processes on chip [3].

If we take a look at the figures of the development time required for these two applications, it is deduced that one can achieve improved performance comparable to respective hand-crafted applications [3] by using Mobius, whereas the amount of effort put in development is similar to Streams-C.

6. Conclusions

A study has been performed by using a CSP based multi-threaded language. The benchmark applications considered range from streaming applications like FFT and DCT to control flow based applications like PPF and PPI, all including computationally intensive and memory intensive kernels.

The study reveals that Mobius can be regarded as a design and algorithm space exploration tool that enhances the productivity by keeping the control with the programmer. The local control and data flow can be parallelized to achieve better throughput and minimum area. However, the parallelization of compute-intensive blocks of an algorithm of course results in an increase in area utilization. The features like automatic pipelining, parameterization and variable length data types allow the programmer to bridge the gap between algorithms and circuits. In addition the availability of floating point operations opens the possibility of efficient mapping of high performance scientific applications onto the FPGAs. However there is still minimal library support available for developing designs in Mobius.

The comparison of Mobius generated results with Streams-C and hand coded implementations proves that Mobius provides the flexibility of a high level language without compromising the device utilization and performance. In terms of development time the achieved engineer efficiency is almost 10 times better than when working with the corresponding HDL codes.

Despite the fact that so much work has already been conducted to explore new paradigms from higher level

software to circuit design level for reconfigurable computing, still there are opportunities ahead of us. One such possibility is to extend the use of the CSP based model to other more coarse-grained architectures such as PACT's eXtreme Processing Platform (XPP) [13] and from a high level language generate low level Native Mapping Language (NML) [14] code.

ACKNOWLEDGMENT

We would like to thank Dr. Per Ljung from CodeTronix LLC and Lars Sjöberg from EWE AB for making their tools available to us and for their support during this study.

The research has been financed by a grant from the Swedish Knowledge Foundation.

REFERENCES

[1] A. Dellson, G. Sandberg, and S. Möhl, "Turning FPGAs into supercomputers- debunking the myths about FPGAs-based software acceleration", *Proceedings of the 48th Cray User Group Conference*, Lugano, Switzerland, May, 2006.

[2] Handel-C language reference manual, Version 3.1, Celoxica Inc., 2002.

[3] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the Streams-C C-to-FPGA compiler: An application perspective", *Proceedings of the 9th ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 2001.

[4] W. Najjar, W. Bohm, B. Draper, J. Hammes, R. Rinker, R. Beveridge, M. Chawathe, and C. Ross, "High-level language abstraction for reconfigurable computing", *IEEE Computer 36(8)*, Aug. 2003, pp. 63-69.

[5] S. Gupta, N. Dutt, R.Gupta, and A. Niculau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations", *Proceedings of the 16th International Conference on VLSI Design*, New Delhi, India, Jan. 2003.

[6] M. Budiu and S. C. Goldstein, "Compiling application-specific hardware", *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, Montpellier, France, Sep. 2002.

[7] C.A.R. Hoare, "Communicating sequential processes", Prentice-Hall, 1985.

[8] Mobius language manual, Codetronix LLC., May 30, 2006.

[9] P. Ljung, S. Zahrai, W. Snapp, and G. Wenes, "Industrial experience using Mobius for rapid hardware development", *Proceedings of FPGA World Conference*, Stockholm, Sweden. 2005.

[10] XAPP219, "Transposed form FIR filters", Xilinx homepage, 16 Sep. 2006.
http://direct.xilinx.com/bvdocs/appnotes/xapp219.pdf

[11] XAPP610, "Video Compression using DCT", Xilinx homepage, 16 Sep. 2006.
http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf

[12] D. Lavenier, J. Theiler, J. Szymanski, M. Gokhale, and J. Frigo, "FPGA implementation of the pixel purity index algorithm", *SPIE, FPGAs and Reconfigurable Processors for Computing and Applications*, vol 4212, Boston, MA, November 2000.

[13] V. Baumgarte, F. May, A. Nuckel, M. Vorbach, and M. Weinhardt, "PACT XPP – A self-reconfigurable data processing architecture", *Proceedings of 1st Inernational Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA'01)*, Las Vegas, NV, June 2001.

[14] PACT software design system M64 reference manual version 4.0 (PSDS-M64_Reference_Manual.pdf), Mar. 2004.