HWML: RTL/Structural Hardware Description using ML

Technical Report 2006

Sven Woop
Computer Graphics Lab
Saarland University
Germany
woop@cs.uni-sb.de

Erik Brunvand
Computer Science Department
University of Utah
USA
elb@cs.utah.edu

Philipp Slusallek
Computer Graphics Lab
Saarland University
Germany
slusallek@cs.uni-sb.de

Abstract

This paper describes how functional programming techniques can be used to obtain simple, compact and highly expressive hardware descriptions even for complex designs. We use the functional programming language ML to describe the hardware structure of a system using a library for basic circuit construction and transformation tasks. The result is an RTL/structural description of the desired circuit that is then easily optimized and mapped using conventional synthesis tools. Using this library embedded in a functional programming environment enables efficient high-level abstractions such as higher order components, abstract polymorphic wires, data streams, automatic pipelining, multi-ported memories, recursive structural definitions, and hierarchy tagging for the resulting RTL/structural description. These features combine to make a powerful and natural environment for describing hardware implementations.

We have used the HWML system in a large hardware project and successfully implement a custom multi-threaded graphics processor for ray tracing in about 4500 lines of code. The resulting circuit has approximately four million non-memory transistors and 2M bits of memory. On an FPGA the circuit already delivers realtime graphics performance and has also been mapped to ASIC standard cells in a 130nm CMOS process.

1 Introduction

When choosing a set of hardware description tools, it is important to find tools that balance the ease of system description with the ability to achieve the desired performance in the resulting system. Tools that describe behavior at a very high level may not result in the best performing systems, while tools that take a very low-level approach to describing circuit structure (schematics, for example) may not be suitable for describing large and complex systems. For high-performance systems a designer often wants relatively transparent control of the resulting implementation. That is, changes to the high-level design description should have understandable effects on the implementation and performance. This argues for at least some control over the structure of the circuit in the high-level description.

In this spirit we have developed the HWML system (hardware meta language) based on functional programming techniques in ML for describing an RTL/structural view of a system rather than an abstract behavioral view.

This increases the transparency of the mapping from the high-level description to the circuit, but still allows powerful high-level abstractions of the circuit structure. These abstractions are at a higher level than are possible in standard HDLs like VHDL and Verilog, but still describe structure rather than pure behavior. The HWML description can be simulated and debugged at a high level in the ML environment before moving to more detailed circuit-level simulations. The result of compiling the HWML description is an RTL/structural VHDL description that is easily mapped to a target technology such as an FPGA or ASIC using standard synthesis tools. The HWML system was developed during the design of a complex multithreaded graphics processor for ray tracing (the Ray Processing Unit (RPU) [7]), and proved to be very suitable for describing this system.

Of course, leveraging an existing programming language to describe hardware systems is not a new idea. Several hardware description libraries for existing programming languages have been developed. Some examples include: System C [8] for C, JHDL [5] for Java, and Lava [4] for Haskel. The capabilities of the mother language strongly determines the style of describing the hardware using these libraries. There are also some interesting extensions to the industry standards VHDL and Verilog, such as Bluespec System Verilog [2]. Functional approaches to structural hardware description have been explored by many researchers [9, 3] but examples of using these languages for large complex systems are few. An advanced language for functional structural hardware description is Lava [4]. Main differences of our approach is, that we support automatic pipelining of arbitrary non-cyclic circuits, and typed multi-ported memories.

Because our project required a high-performance implementation, we chose to focus on a high-level structural descriptive style. This paper describes how to perform high level structural hardware abstractions using an ML library. The focus of this paper is not only to describe a new functional hardware description language, which we believe has significant advantages over existing functional approaches, but also to show the advantages of functional hardware description for large scale designs in practice through its use in the design of a large and complex system. We believe that by carefully choosing the abstractions, a structural description can be quite compact and as a consequence quite readable and understandable by the designer.

The HWML hardware description library described in this paper is a quite small basic library implementing functionality like structural circuit construction, and support for pipelining and simulation. It is implemented in the functional programming language ML. We use the Moscow ML dialect but mappings to different ML dialects should be trivial. Some basic knowledge of ML programming is advantageous to understanding some of the following sections. For a brief introduction to ML see [6].

We used the abstraction concepts presented in this paper to build a large scale ray tracing graphics processor. The complete system has been implemented in about 4500 lines of HWML code in about 6 man months concurrently with the development of the HWML system. This shows that compact and expressive HWML descriptions can result in complex working hardware realizations.

The following sections of the paper describe the low level structural library, and how that library is used to create abstractions for automatic pipelining, abstract polymorphic wires, data streams, multiported memories, recursive structural definitions, and hierarchy tagging for the resulting RTL/structural description. We then describe the results of using these techniques on the Ray Processing Unit (RPU).

2 Low Level Structural Library

On order to use ML as a hardware description language we have created a basic library that supports the description of structural circuits. Our structural library implements basic primitive Boolean functions such as `And`, `Or`, `Not`, `Mux` and others to describe the structure of combinational circuits. A delay function `Reg` introduces the possibility of creating synchronous circuits using a register clocked by a global clock signal. The introduction of registers also requires the ability to build cyclic circuits. To make this possible in the ML framework one can create a fresh wire and then assign a different wire to it later, thus closing the loop.

For the later inclusion of external components, such as memories or highly optimized floating point units, an atomic *pipeline element* with specified latency and a simple *black box* are also supported in the low level library. To these elements one can give semantics for simulation by attaching some behavioral ML simulation code.

2.1 Circuit Creation

In order to understand how an ML program using the library maps to a circuit we give a simple example.

```
fun reg_en en in =
  let val out = Wire ()
      val _ = Assign(out,Reg (Mux (en,out,in)))
  in out end
```

This ML code defines a register `reg_en` with clock enable that gets a clock enable signal `en` as its first and a wire `in` as its second argument. In order to construct a register with clock enable out of a standard register we require a loop, thus we first create a fresh wire `out` which will be the output. Assigned to this wire is a register that clocks the input to the output if `en` is true or it holds the output if `en` is false. From these simple lines of ML code the library creates a graph representation whose nodes are labeled with the names of the logical functions (not, and, or, mux, reg, ...). This has a very straightforward mapping from the ML description to the hardware structure. The ML program is simply executed by the ML system, and the library calls directly generate graph nodes with the corresponding label.

This graph representation can be written out to a VHDL file containing behavioral statements for each combinational library function, and instantiated components for some special black boxes like memories. The resulting VHDL code for `reg_en` is:

```
architecture reg_en of reg_en is
    signal w0,w1,w2,w3,w4: std_logic;
begin
    w3 <=  port1; w4 <=  port0;
    w2 <= ((not w3) and w0) or (w3 and w4);
    reg_w1 : process (lclk) begin
      if lclk'event and lclk = '1' then
        w1 <= w2;
      end if;
    end process reg_w1;
    w0 <=  w1; port2 <=  w0;
end reg_en;
```

Fine grained technology mapping into a specific target library such as that for an FPGA or an ASIC standard cell library is done later by standard synthesis tools. Mapping for large memory structures is done by our tool and results in FPGA memory instantiations or specifications for an ASIC memory compiler to generate black box memories.

2.2 Automatic Pipelining

The low level HWML library can introduce pipelining into the circuit by taking a non-cyclic circuit, that may already contain pipeline elements, and creating a balanced pipeline that is adjusted to a specified delay per pipeline stage.

The pipelining algorithm is separated into three steps.

1. A simple constant propagation algorithm is applied which replaces subgraphs that contain a constant wire by its reduced form, like replacing $And(x, 1)$ by x. This optimization can reduce the depth of the circuit and help prevent the algorithm from inserting too many pipeline stages.

2. The depth to each of the cells in the circuit is computed. All atomic cells (individual Boolean gates, for example) have a specified delay. The depth of a node in the graph representation is now defined as its own delay plus the maximum of the depth of its input nodes. By applying this definition recursively onto the circuit, one can assign a depth to each graph node.

3. The algorithm now walks the graph and inserts pipeline stages at depth $0 \cdot step, 1 \cdot step, 2 \cdot step, \cdots$ until it reaches the maximum depth of the circuit. The stepping size is set to the specified delay per pipeline stage.

The automatic pipelining to a specified delay per pipeline stage is an important function that enables the following abstractions to work more easily. Because the high level abstractions view the circuit at the dataflow level, circuits may have varying latencies or may be described recursively. Manually inserting register stages into a recursively defined multiplier, for example, would be quite complicated, as this type of modification does not map well to the recursive definition scheme. Instead, the functions are described using whatever HWML structures make sense, and the pipeline registers are inserted automatically.

2.3 Simulation

The HWML library can perform a cycle accurate simulation of the generated graph representation. To feed the simulation with data, an ML test bench applies stimuli to the input wires in each cycle of the simulation. The library propagates the circuit values in each cycle for the primitive gates and calls user callback functions to simulate the behavior of the black boxes such as memories.

3 High Level Abstractions

Based on the low level functionality one can now use the HWML framework to define higher level abstractions to simplify the hardware description process. Besides the automatic pipelining, memory abstraction, and hierarchy tagging, the techniques described in the following would also be possible on top of Lava by using the functional capabilities of Haskel.

3.1 Components as Functions

The first abstraction is to represent hardware components by ML functions. Compared to standard structural components this allows higher level semantics. Thus a function can by polymorphic, which means it can implement different structural behavior depending on the type of input wires, as explained in the next section. The function arguments and results do not necessarily correspond to input wires and output wires. By passing a fresh wire to a function, it can return a result to this argument wire by assigning a different wire to it. Similarly one can use a result wire as input by returning a fresh wire and already using it in the computations.

3.2 Abstract Wires

The low level library contains a wire type `bit` whose usage is quite limited. Assume you like to multiplex a tuple of two wires, then you have to build a special function `mux2` to select between both tuples.

```
fun mux2 sel ((a0,b0),(a1,b1)) =
  (Mux sel (a0,b0), Mux sel (a1,b1))
```

For each type of wire you would need a separate multiplexer function to operate on it. A more elegant method would be to have only a single function that automatically chooses a different multiplexer based on the type of the wire, which is called polymorphism. Such a function should be useable on all wire types supported in HWML: bits, integer wires, floating point wires, or combinations of them. To make this possible we use the datatype construct of the ML language and define our own wire datatype as follows:

```
datatype Wire =
  B of bit                        boolean wire
| I of bit list                   integer wire
| F of bit*bit list*bit list      floating point
| L of Wire list                  list of wires
```

This datatype specification can easily be extended for special needs such as more general fixed point arithmetic. This datatype defines a new type called `Wire` that can either be a simple boolean wire `B`, an integer wire represented by a bit list I, a floating point wire F consisting of the sign, exponent and mantissa, and a list L of wires. The constructors B, I, F, and L can be used to build abstract wires and using ML pattern matching them can be decomposed again.

These abstract wires are internally similar handled as VHDL bit vectors but for the hardware designer them contain additional information about the structure and type of the contained bits. Furthermore they allow to write polymorphic functions over this abstract wire datatype. For example, the following multiplexer can multiplex arbitrary inputs of the same type.

```
fun mux (B sel) (B a, B b) =
    B(Mux sel (a,b))
  | mux (B sel) (I a, I b) =
    I(map2 (Mux sel) (a,b))
  | mux (B sel) (F(s0,e0,m0), F(s1,e1,m1)) =
    F(Mux sel (s0,s1),
      map2 (Mux sel) (e0,e1),
      map2 (Mux sel) (m0,m1))
  | mux sel (L a,L b)=L(map2 (mux sel) (a,b))
  | mux _ _ = raise Error " ... "
```

This multiplexer function is a polymorphic function over the wires that can select between tuples of arbitrary wires of the same structure. There are four cases in the declaration, one for boolean, integer, floating point and list of wires. The map2 function call maps the multiplexer recursively onto two lists of wires. If there is a structure mismatch in the two wires the function fails with a runtime type error.

The ML language allows the programmer to define new operators which, when used carefully, increases the readability of the code. In the following the operator A && B will stand for logical and, A || B for logical or, A ++ B for addition, and A <- B for the assignment of wire B to the fresh wire A. HWML defines a number of other operators including multiplication but these are sufficient to describe the concepts in the rest of the paper. All the operators are defined to be polymorphic over the supported data types.

3.3 Stream Abstraction

A datastream is a basic concept of hardware architecture defining data flowing along some paths in a data flow graph. Communication schemes are often implemented using datastreams. For example, a function call to a component could consist of an input datastream for the argument and an output datastream for the result of that function call.

A datastream that can hold arbitrary data can be abstracted by a triple of three wires: a valid signal that indicates if there is currently data on the stream, the data itself, and a busy control signal that indicates the stream should stop. The data on the stream can of course be of any complex type, while the valid and busy signals must be of boolean type. Note that the busy signal indicates that the previous unit in the stream should stop thus its direction is the opposite than the valid and data signals.

Using this stream abstraction, we can define functions that operate on streams. For example, we can define a stream statement to create a stream, an assign statement to close a cyclic dataflow, stream multiplexers, a demultiplexer that directs the datasteam to one of two output streams, fifos, a stop statement to stop a datastream, a delay statement to delay the stream element by one cycle, a semaphore that only allows *n* stream elements to be present on a region, a merge statement that merges two streams, and a split statement that splits a stream into two streams (the inverse of the merge operation). Note that each of these operations can work on streams that hold arbitrary data.

As a simple example we show here the implementation of the merge operation that merges two streams into one.

```
fun merge (val0,data0,busy0)
          (val1,data1,busy1) =
let
 val busy = Wire TyB
 val _ = busy0 <- val0 && not valid1 || busy
 val _ = busy1 <- val1 && not valid0 || busy
in (val0 && val1,L[data0,data1],busy) end
```

This function takes a data element from stream 0 and stream 1 and creates a new stream with those two data elements combined together. Note that one input stream is stopped if it contains data while there is no data on the other stream or if the output stream is stopped. By combining the high level stream modifiers HWML can generate each type of control flow path at a very high level without dealing with low level control automata.

Such a stream merge operation can be used, for instance, if the two arguments for an addition are computed at an unknown time. Then one can synchronize the input stream of the first argument and the stream of the second argument with a merge function and put the output stream to the adder.

```
fun stream_add (valid,L[a,b],busy) =
    (valid,a ++ b,busy)

val sum_steam =
    stream_add (merge stream0 stream1)
```

One can see here how well the function abstraction works together with the stream abstraction. The input streams (which consist of wires of different directions) are given as the argument to a function and the resulting output stream can then be processed further.

For debugging purposes one can pass special watch functions to the low level library that are called after each clock cycle. These functions can interpret the values on a stream and e.g. print them to the standard output. Thus in debugging one can easily follow the computation at arbitrary points by adding a small watch statement to a stream.

3.4 Pipelines as Stream Elements

As values pass through a datapath they are often operated on by functions, like the addition in the previous example. In general these computations might be quite complex thus not be executable in a single cycle. But it is desirable to write these combinational computations in an as abstract form as possible without spending time to manually adjust the circuit to an appropriate latency. Automatic pipelining transforms combinational circuits into efficient pipelines by using the automatic pipelining of the low level library extended with some streaming control.

The following example reads two vectors from the two input streams and performs a dot product computation that is pipelined to match a target frequency.

```
fun dot (L[L[x0,y0,z0],L[x1,y1,z1]]) =
         (x0 ** x1 ++ y0 ** y1) ++ (z0 ** z1)
val out_steam =
    pipeline dot (merge stream0 stream1)
```

The pipeline function transforms the pipelined result to a stream by adding the valid and busy control signals to it.

3.5 Multiported Memory Abstraction

Nearly every hardware design needs memories in which to store computation state. Often these memories also need multiple ports for instance to implement main register files that read multiple arguments per cycle. Such multiported memories can be abstracted quite simply in our HWML framework. One can define a function that creates a typed memory with multiple ports and specified address and data type. The ports of the memory are returned as functions, that can be called to connect to the memory ports.

In the low level library, each port is represented by a pipeline element of cycle latency 1 and can be used only at one position in the circuit to avoid resource conflicts. The basic library knows nothing about the semantics of the memory, except the latency of the read and write ports, thus it can pipeline circuits that contain memories.

For simulation one has to apply some behavioral semantics to the memory components. This is done by providing ML callback functions for each memory. This function will be called during simulation to provide the behavior of that memory.

To export a circuit containing memories the memory creation function also must be able to synthesize the abstract memory to the atomic memories available in the target platform. Depending on the available memories this might mean that the memory needs to be split horizontally (reduce data width), vertically (reduce address width) or duplicated to emulate more ports. The valid atomic memories are then created as black boxes in the low level library with some circuitry around to connect them. As this synthesis step is at a high level it is efficient and it is guaranteed that memory blocks of the target platform are really used. One might export the same circuit description to a Xilinx FPGA taking efficient usage of the memory hierarchy available there, to other FPGAs, or to an ASIC process by automatically building memories using an appropriate memory compiler.

The advantages of our memory abstraction over the manual use of structural atomar memory blocks of the target platform (such as Xilinx block memories) is that we are platform independent, as this mapping is performed automatically. Futhermore the memory maintains the type of the wire, reducing the possiblity for design errors. Some synthesis tools support mapping behavioral VHDL memory descriptions to atomar memory blocks (such as the Xilinx synthesis tools), but only if the hardware designer uses a strict coding style. Here again portability is an issue, as a direct synthesis to an ASIC target platform (with its own memory compiler tools) is not possible.

3.6 Recursion and Higher Level Functions

Recursive circuit descriptions play an important role if designing arithmetic units. For instance a recursive parallel prefix function is required to define a carry lookahead adder, or a recursive balanced reduction operator to compute the CRC checksum of a bit vector in logarithmic time. The recursion must of course terminate at a statically determined maximal depth as otherwise an infinite circuit is specified. For example, a recursive definition of a reduction operator that generates a balanced reduction tree over an arbitrary reduction function `f`, terminates if the length of its input list reaches 1 after some recursive splits.

```
fun reduce f nil = raise Error "..."
  | reduce f [x] = x
  | reduce f lst =
    let val left = take (lst,length lst / 2)
        val right = drop (lst,length lst / 2)
    in f(reduce f left,reduce f right) end
```

This function first splits the input list into two parts of (nearly) equal length and applies the reduce operation recursively on these parts. Then, the two sub-reductions are combined with the function `f` again. Note that the reduce function is a higher order function that gets an (associative) function `f` as input. If we set this function to the xor operator the reduction computes the CRC checksum of its input vector: `val CRC = reduce xor [B0(),B1(),B0(),B0()]`.

One can use the same reduction operation to sum a list of integer wires for instance, by simply writing `reduce add [a,b,c,d]`. Obviously, such an adder tree might have a high delay so writing such a line seems of limited use. But, as we support automatic pipelining one can simply create such a tree and pipeline it later if that is what the designer wants.

3.7 Hierarchy Tagging

For optimal automatic or manual placement on an ASIC or FPGA a physical hierarchy of the design is very important. A good hierarchy, required for physical layout, does not always match the logical style in which a circuit is specified. We can introduce hierarchy to the structural circuit by using the opening `down hierarchy_name` and closing `up()` functions, that can also be applied recursively. All primitive components instantiated between these function calls are packed to the hierarchy `hierarchy_name`. Opening the same hierarchy again, one can always add additional gates to it regardless of the HWML program hierarchy. It is possible, for example, to define a new reduction operator that packs together operations at the same level to a hierarchy. Thus one can later place an arbitrary reduction level by level to get an optimal placement.

4 Results

The HWML system was developed concurrently with the development of a custom processor RPU for generating computer graphic images using ray tracing [7]. As a result, the design decisions for HWML were made in the context of a large hardware project. Automatic pipelining, streaming, bundled wires of complex types, and hierarchy tagging were used extensively in the RPU design. Also, recursive arithmetic unit definitions were used for the custom floating point units.

4.1 Ray Processing Unit (RPU) Graphics Chip

Using HWML we have developed a complete multi-threaded ray tracing processor (RPU), including four GPU-like shading processors and dedicated hardware units for fast traversal of rays through spatial index structures [7]. The RPU is a highly multithreaded, 4-way SIMD processor with synchronous parallel execution of bundles of threads for shading connected to a dedicated unit to perform ray traversal. The memory connection uses several on-chip caches for the scene information. The architecture is fully programmable (except the custom traversal unit), and implements a complete GPU processor instruction set with additional support for recursion and fast shooting of rays. The RPU is fully functional on an FPGA prototype, and has also been targeted to an ASIC in a 130nm CMOS standard cell design for comparison. Ray tracing performance achieved with the FPGA based prototype clocked at only 66 MHz is similar to that of traditional CPU based ray tracing implementations even on modern multi-GHz CPUs [7]. The ASIC version would achieve about 16 times a single PC CPU's ray tracing performance if four RPUs were integrated on a single chip. The size of a 4-RPU chip would be around $121mm^2$ in a 130nm CMOS process.

As far as we know, this design is the first large scale project that has been implemented exclusively with a functional hardware description language. This approach made it possible to implement the complete (single RPU) chip in about 4500 lines of HWML code in about 6 man months concurrent with the development of HWML. This shows that compact and expressive hardware descriptions can be created with the HWML functional abstractions.

The design is parameterizable to a large degree. By only changing single configuration values of the HWML specification one can modify the number of threads supported in the system, the number of threads contained in a synchronous bundle, the computation accuracy (floating point data path width, which also modifies data path widths, RF size, etc.), the pipelining depth, and many more parameters. These degrees of freedom allow design studies by trying different configurations and testing them for efficiency, or generating simple area estimates find a suitable configuration. For example, the FPGA implementation is limited to 24bit floating point precision by the limitations of the specific Xilinx FPGA used whereas the ASIC version uses 32bit floating point.

The stream abstraction was used throughout the RPU design and was essential to the fast development of the system. Rather than develop control automata directly, the stream abstraction generated the control automata as a consequence of the stream structures. Wire abstractions allowed much more compact and readable code as complex wires are bundled together and polymorphic structural elements automatically adjust for the new wire types. Because we are targeting two very different technologies (FPGA and ASIC), the memory abstraction that generates different memory structures based on the target was also essential.

The statistics of the FPGA and ASIC version of the completed RPU are:

FPGA Version: Implemented in a Xilinx Virtex-II 6000-4 FPGA [11] hosted on an Alpha Data ADM-XRC-II PCI board [1]. The RPU uses the Xilinx FPGA to its limits as 99% of all slices are occupied and 88% of the block RAMs used. The design contains 48 24-bit floating point units which use the available 18-bit block multipliers on the FPGA, support for 128 hardware threads and 4 rays per synchronous thread bundle. The FPGA runs at 66MHz typical case.

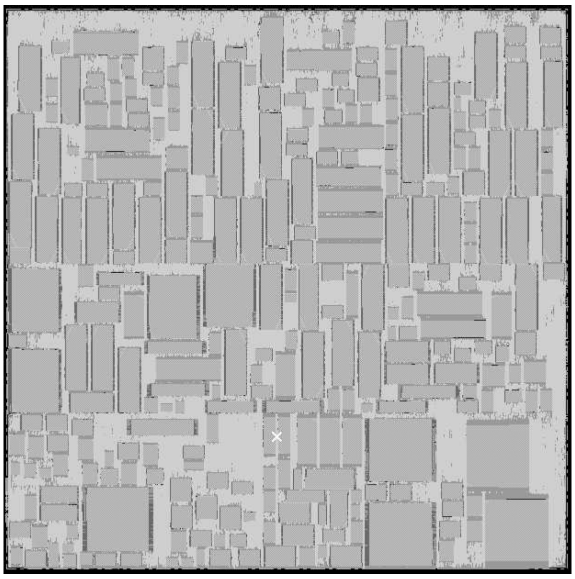

Figure 1. The ASIC version of the RPU shown with only four of the six levels of metal wiring.

ASIC Version: Implemented in 130nm CMOS using a standard cell library from UMC [10]. A single RPU (one traversal unit, four shading processors, plus memory) is 5.5mm X 5.5mm and contains 48 32bit floating point units, support for 128 hardware threads, and 4 rays per synchronous thread bundle. The total design has approximately 4 million non-memory transistors and 2 million memory bits distributed over 301 separate memory blocks. Post place and route timing estimates for the current version are 150MHz worst case and 315MHz typical case which is 70% of the maximum potential execution speed limited by the speed of the memories.

4.2 Floating Point Unit Comparison

To evaluate the ability of HWML in describing detailed structural components we designed our floating point units in HWML and compared them to floating point units synthesized using a commercial datapath synthesis tool, both targeting the UMC 130nm standard cell library. The HWML version leveraged the ability of HWML to describe recursive hardware structure by first generating the partial products, then sum them using a 3-to-2 adder tree, adding the exponents and finally normalizing the result. For a 32-bit floating point multiplier, pipelined to a latency of three cycles, the post place and route extracted timing for both versions of the circuit are nearly identical 520 MHz worst case. The HWML-generated version has a size of $108,768\ \mu m^2$ whereas the multiplier synthesized by the commercial tool is slightly smaller at $82,311\ \mu m^2$, most likely due to that tool's ability to use more complex cells in its synthesis, but the comparison shows that even detailed circuits compiled from HWML can be competitive with those generated by commercial tools.

5 Conclusion

In this paper we describe a low level hardware description library for ML called HWML which enables a number of high level abstractions and leverages a number of features of the ML functional programming environment. These powerful abstractions include automatic pipelining, abstract polymorphic wires, data streams, multiported memories, recursive structural definitions, and hierarchy tagging for the resulting RTL/structural description. The result of compiling the HWML description is collection of RTL/structural VHDL files that can be mapped to a target technology using standard commercial synthesis tools. All these features together allow us to write compact, expressive and reusable code which results in high-performance circuit implementations. We have demonstrated the HWML design flow by implementing a large scale graphics chip with these high level abstraction techniques. The designer's productivity is increased using these abstractions, but the structure of the circuit is always visible to the designer which leads to good transparency of the design process and allows the designer to make informed decisions about how to optimize the design.

References

[1] Alpha-Data. ADM-XRC-II. *http://www.alphadata.uk.co*, 2003.

[2] Bluespec. Bluespec webpage, http://www.bluespec.com.

[3] A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48, 2000.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *ICFP '98 in Baltimore (Maryland, USA)*, 1998.

[5] Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In *IEEE. Symposium on FPGAs for Custom Computing Machines*, pages 175–184. IEEE Computer Society Press, 1998.

[6] Scott Smith. Introduction to the ML programming language, www.cs.jhu.edu/ scott/cw/lectures/sml-intro.html.

[7] Sven Woop, Jörg Schmittler and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *SIGGRAPH 2005 Conference Proceedings*, 2005.

[8] SystemC Community. SystemC, www.systemc.org.

[9] Tom Hawkins. Confluence tutorial and reference manual. www.launchbird.com.

[10] United Microelectronics Corporation. http://www.umc.com, 2005.

[11] Xilinx. Virtex-II. *http://www.xilinx.com*, 2003.