Hardware Compilation of Sequential Ada

M. Ward
Dept. Computer Science
University of York
York, UK
michael.ward@cs.york.ac.uk

N. C. Audsley
Dept. Computer Science
University of York
York, UK
neil.audsley@cs.york.ac.uk

ABSTRACT

Normal implementations of real-time systems on conventional processors are becoming much more difficult to prove correct to their timing specification. This is due to the complexity of modern processors (e.g. the worst case execution time of a program becomes hard to calculate in the presence of CPU speed up features such as caches and pipelines).

Field Programmable Gate Arrays (FPGAs) provide a way to ease this problem by providing an implementation medium that has a simple timing model. However there is no support for real-time languages on FPGAs.

This paper describes a compiler for a sequential subset of Ada95, concentrating upon compilation of subprograms and statements. It is shown how the resulting circuits give simple timing analysis. Extensions to the current compiler are explored to give support for a larger range of types and a predictable subset of the Ada95 concurrency model.

1. INTRODUCTION

1.1 Real-Time Background

Real-Time Systems are those where the timing of an output is as important as the accuracy of its value. Dependable real-time systems are used to control high cost of failure systems, for example flight control software on a plane, or nuclear reactor control systems. These types of system have to be proved correct, both functionally and timely. It is the correctness of timing that we are concerned with.

Confidence in the timing of the system is achieved through the analysis of the program. This is done in two parts:

1. *Worst-Case Execution Time (WCET) Analysis*
   Each basic block of the program is examined for its execution time. The longest path through the program is then found, and the worst case execution time for each process can be calculated.

2. *Scheduling Analysis*
   Since a number of the processes in the system will run on a single processor, it has to be shown that they will all meet their deadlines despite interference from the other processes.

CASES'01, November 16-17, 2001, Atlanta, Georgia, USA.

Calculating the WCET of a basic block can be done simply by adding together the execution times of the individual instructions. This gives an accurate WCET only when every single instruction takes the time given in the data book, and there are no other delays. This only occurs with legacy processors (e.g. Z80 or 68000) which have no architectural speed up features. Any structure within the processor that can vary the timing of an instruction causes problems with this analysis process. This is because they can alter the execution time of an instruction. Pipelines tend to reduce the instruction timing, by up to 80% for a 5-stage pipeline[22], but a cache can increase the execution time significantly, from a single cycle for a hit, to 40 or more cycles for a miss to main memory[13].

Treating each memory access as a cache miss, and ignoring the effects of the pipeline will still produce a WCET that the process will never exceed, but the difference between the calculated and actual WCETs will be vary large. This difference is referred to as pessimism, and a lot of research has gone into reducing the level of pessimism through the use of more accurate modelling of the target processor[12][17]. However, with the latest generation of processors (Intel Pentium 4 etc) not having their technical details made public, making a correct model will be difficult, and levels of pessimism will go up. Increases in pessimism are bad; a processor has only a fixed amount of performance, pessimism reduces the amount of performance that can be guaranteed. It has been suggested that pessimism can reduce the level of guaranteed performance as architectural features are added despite increasing clock speeds.

It is this pessimism that we are trying to remove through the use of FPGAs as the implementation target of real-time systems. FPGAs are preferred over ASICs as they can be re-programmed in a short time, simplifying the maintenance of systems.

1.2 Hardware Compilation Background

There are two forms of language available for hardware compilation:

- *Hardware Description Language (HDL)*
  A typical HDL (e.g. VHDL [24], Pebble [18] and SL [15]) allows a hierarchical description of a circuit. Each block in the circuit can be described in terms of lower level blocks, and the connections between them, or a

functional description of the block, possibly with other blocks used in this description.

- *High Level Language (HLL)*
  A HLL allows the programmer to describe the functionality of the circuit using normal programming syntax. All translation to and arrangement of the final circuit is done by the compiler.

Unfortunately, HDLs are unsuited to the implementation of real-time systems. They have no communication constructs (though VHDL++ [10] proposes these), which are an important part of real-time system [8]. Timing facilities (e.g. delays and clocks) are also important to real-time systems, but as with communications, HDLs do not provide language based support for this. They do, however, have good coarse grained parallelism support through the circuit block structure.

There have been several implementations of high-level languages for hardware compilation. Most of these are based on the C language (e.g. Handel-C [6], Transmogrifier C [11] and BRASS [9]), though Occam (Handel [20]) and Java (Xilinx Forge [25]) have been used.

One of the main C-based languages is Handel-C, produced by Celoxica. This is based on a subset of the C language with added parallelism and CSP [14] based communication constructs. Even with these additions Handel-C is unsuitable for real-time systems implementation. The communication channels provided are too simplistic to provide the style of synchronisation and communication needed in real-time systems. Also, the parallel structures provided are better suited to fine grained parallelism than the coarser grained parallelism used in real-time systems.

The Xilinx Forge-J implementation of Java uses the compiled byte-code of a Java program to produce a circuit in FPGAs. This makes full support of the language easy as there are only a limited range of byte-codes. Java does provide support for the concurrency and communications needs of real-time systems, through use of the 'thread' class. However, the object-oriented nature of the language does cause a lot of analysis problems. There is little support for scheduling amongst the different threads, making it hard to ensure that they all meet their deadlines. The garbage collection memory model also causes problems as the collection of garbage occurs at unknown times for unknown periods. The proposed real-time extension to Java [5] addresses some of these issues, but the fundamental nature of Java's dynamic object creation and class loading make timing analysis difficult.

Hardware compilation for real-time systems requires the support of a true real-time language by a hardware compiler. A real-time language has a number of requirements[8] including, syntactic support for the concurrency, communication and timing needs of a real time system. For this work, an established language was used, having the advantage that legacy systems can be easily migrated to a new implementation when the current platform has become obsolete. Ada95[2] is a language that has been used extensively in real-time systems implementations, having defined and understood semantics. The SPARK[4] and Ravenscar[7] subsets of Ada are used in dependable system development, and due to their simplification of the Ada language, a combination of these subsets was used in this work.

2. THE ADA LANGUAGE

Ada was first standardised in 1983 [1] after a US DoD request for a common language suitable for the implementation of all DoD systems. The standard was revised in 1995 [2] to its current state. The standards define a core syntax and a number of annexes. Each annex extends the language to deal with a specific type of system (e.g. real-time, distributed or dependable), by defining how the compiler should operate for these systems. The annexes do not add any syntax to the language.

Ada has a block structured and procedural syntax. It is heavily type checked, requiring explicit conversions between different types. All standard types are provided (integers, floats, characters, arrays and records) as well as some more complex ones (e.g. variant records and tagged types, which provide object-oriented facilities). Pointers are also provided but they are severely type restricted, only being allowed to point to objects of their base type.

Two types of sub-program are provided. Procedures take a number of parameters, perform operations on them, and return with some of the parameters potentially changed. Functions take a number of parameters, perform operations on them and return a value. The parameters and results of sub-program calls are passed by the most appropriate method (usually value for simple types, and reference for complex types).

Programming in the large is supported through several means:

- Separate compilation is supported at the language level, each program unit having a specification (interface definition) and a body (the code itself). Any body unit can be compiled alone so long as the specifications for all units it uses are available. This allows easy testing of each unit.
- Information hiding is supported through the provision of packages and protected objects.
  - Packages allow all operations on a data type (and the structure of the type) to be hidden from the rest of the system, only providing an interface to this type.
  - Protected objects wrap up data and restrict access to it through defined operations, in a concurrent system protected objects also provide mutual-exclusive access to the data.
- Concurrency is supported by the language level syntax of tasks, together with well-defined inter-task communication and synchronisation.

SPARK

The SPARK sequential subset of Ada95 is described by Barnes in [4]. It is a mathematically analysable subset of the Ada language, allowing programs to be proved (partially) correct. To become analysable, all dynamic constructs are removed. The main type removed is the access (pointer) type, which also removes the variant records used to give Ada object-oriented capabilities. Sub-programs are no longer allowed to recurse, nor can procedure pointers be used. These restrictions make all subprogram call trees known at compile-time, and all variable references resolve to only one instance. Being sequential, SPARK does not

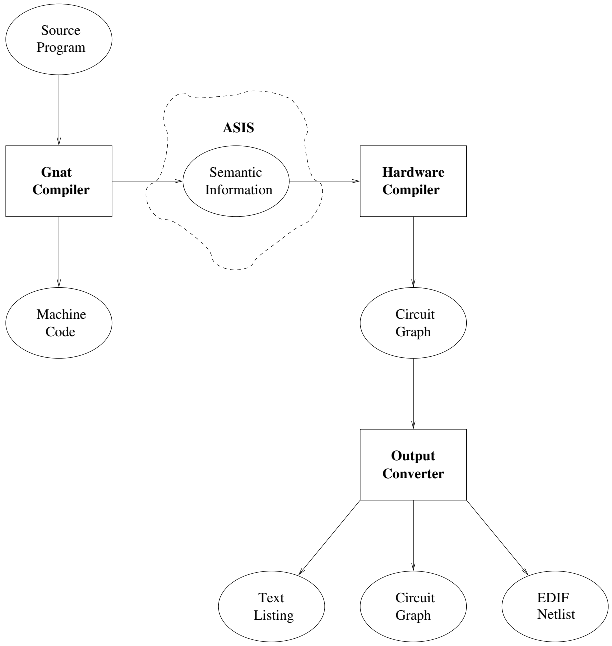

Figure 1: Hardware compiler structure

allow any form of concurrency, so tasks and their associated communications and synchronisations are not part of the SPARK language.

Ravenscar

The Ravenscar tasking profile[7] restricts the freedom of concurrency and communication provided in the language specification. It is a statically analysable tasking subset that removes all run-time dynamicism from the tasking system. Hence, unlike full Ada95, Ravenscar compliant code is predictable in its timing behaviour and resource usage.

The Ravenscar profile makes no comment on the sequential part of the language, so to provide an analysable, concurrent language, we combine the Ravenscar profile with the SPARK language.

3. ADA95 HARDWARE COMPILER

Figure 1 shows the structure of the hardware compiler. The front end parsing of the GNAT compiler[1] is accessed through the Ada Semantic Interface Specification (ASIS) [3]. The ASIS interface allows access to the syntactic and semantic information produced by an existing compiler during the compilation process, obviating the need to implement a parser and analyser specifically for this compiler. The hardware compiler takes four passes:

1. *Parsing and Analysis*
   The source program is run compiled by the GNAT Ada compiler, which generates conventional code as well as the semantic information used by the ASIS interface. This may involve several passes internal to the compiler.

[1]The GNAT Ada compiler is a fully validated Ada95 compiler, the public domain version is available from `www.gnat.com`

2. *Variable Analysis*
   All the source code is scanned for the usage of variables. This ensures that the implementation of a variable does not interfere with its use in expressions, for example a single ported array cannot be used twice in a single expression.

3. *Circuit Graph Generation*
   The program is converted to circuit form through template instantiation of the statement forms in the program. The circuit generated is an intermediate representation of a graph containing all the cells and interconnections in the final circuit.

4. *Output Generation*
   The circuit graph is converted to one of three forms, either textual, graphical, or a netlist which can be converted to a configuration file for use on an FPGA.

This paper concentrates on the process of circuit generation from each procedure, which is described in section 3.2. An overview of the parsing and analysis passes is given in section 3.1. Section 3.3 gives an overview of the generation of the three possible output forms.

3.1 Program Parsing and Analysis

The compiler front end is implemented through the ASIS interface, reusing the existing parsing and semantic analysis phases of the GNAT Ada compiler. ASIS is accessed through the use of queries, which either return references to new items, or to an item semantically or syntactically linked to a given item. ASIS deals with two types of item, compilation units and elements. Compilation units represent the files in the source program, as well as the compilation units (packages, sub-programs, tasks, etc.) in the source. Elements are used to represent below compilation unit structures, for example, definitions and declarations, statements or expressions. It is interesting to note that a subprogram definition or declaration can be represented as both a compilation unit and an element, however, queries will only work with one or other representation.

When started, the hardware compiler looks for all the compilation units that make up the program and each of these is considered in turn. Each compilation unit is scanned for usage of variables within it, particularly the maximum number of uses of each array in an expression so that the array can be generated with the correct number of ports. Following this scan for semantic information on the use of variables, the compilation process can begin.

3.2 Procedure Compilation

Each procedure produces a circuit, which contains the control and data manipulation circuitry associated with the procedure. All circuitry relating to the variables and formal parameters of the procedure are stored in the context associated with each circuit. These combine to be the graph which represents the entire circuit for the program. Compiling a procedure therefore requires that three parts of the circuit are produced. Using the procedure in figure 2 these parts are:

- *Data storage*
  Local variables and the formal parameters of the procedure are defined in the declaration block at the start

```
procedure trisum (n : in integer;
                  sum : out integer) is
      -- declarations
      tri : integer;
   begin
      -- Procedure body code
      --   transforms to data and control structures
      sum := 0;
      for i in 1..n loop
         tri := 0;
         for j in 1..i loop
            tri := tri + j;
         end loop;
         sum := sum + tri;
      end loop;
   end trisum;
```

Figure 2: Example Ada procedure. This calculates the sum of all triangular numbers up to the given parameter.

of each procedure. There are five local variables in the example, `tri` is a locally declared variable, `n` and `sum` are formal parameters to the procedure, and `i` and `j` are local variables that are inferred from the for loops. These are converted to cells in the final circuit as described below. The declaration block can also contain the declaration and definition of procedures, so these need to be dealt with as well.

- *Data Manipulation*
  The data manipulation circuitry is mostly generated from the expressions within the procedure (the rest is generated within statements, e.g. `for` loop counters). For example, the expressions `tri + j` and `sum + tri` are explicit in the example program, whereas the expressions `i <= n` and `j <= i` are implicit in the for loops as termination control of the loops. Expression trees are translated into a combinatorial hierarchy of operator cells.

- *Control structures*
  Statements, which control the flow of program execution, provide almost all the control circuitry (some control circuitry is required at the start and end of a procedure). Statements are compiled using a syntax directed template instantiation.

The final circuit for a procedure is a one-hot state machine[2], each statement compiling to one or more of these states. The current state is the one with a high output on the state flip-flop. This was chosen, over the high input to the current state used by Handel and other compilers, as it allows the saving of one state flip-flop in decision statements when full advantage is taken of parallelism. At present the compiler does not do this due to complexities of detecting and resolving access conflicts in parallel expressions. The procedure circuit has two signals interfacing into it, a start signal, indicating that the procedure should start executing, and a finish signal, indicating that it has completed. There are also data and control lines for copying parameters into and out of the procedure.

[2]A one-hot state-machine uses a register for each state, only one of which is active at any time (hence one-hot). This contrasts to an encoded state machine where a counter is used to count through the states, and a decoder used to indicate which state is current. The former method uses a lot of registers, but little other logic, the latter, few registers, but much more logic.

Declaration compilation

Firstly, the declarations are dealt with, the several types of declarations processed by the compiler are described below:

1. *Variable definitions*
   A variable, of which only integer types are dealt with at present, is represented as a register within the output circuit. This register requires additional circuitry to allow for multiple assignments to the variable. A data selector is required pick the correct value to be stored in the register, and an OR gate is needed to allow multiple write-enable signals to be connected to the register. These parts of circuit are then added, with a reference to the variable, to the context.

2. *Array definitions*
   An array is implemented as a block of memory in the circuit. According to the results of the first pass, the memory is created with sufficient ports to cope with all the accesses in an expression. Like the variables, additional circuitry is required, a data selector and an OR gate for the same reasons as above, but also an address selector for each port, to pick the appropriate address expression. As with variables, these are then added to the context. The compiler can only cope with single dimensional arrays of integers at present. Multiple dimensional arrays can be implemented (at greater cost of resources) by appending the address lines from each index to create a larger address. This way a 3-dimensional 5-bit indexed array would be implemented as a 15 address-line memory. A decision needs to be taken by the compiler as to the location (on- or off-chip) of the memory used by an array.

3. *Procedure declarations*
   A procedure declaration needs to be processed to produce the variables for its arguments, as well as allowing for the connection of the procedure to its calls. To this end the variables are created as above, and stored, along with lists of the call and return ports that the procedure needs to be connected to, in part of the current context.

4. *Procedure definitions*
   A procedure definition can be one of two forms: a completion of a previous declaration; or a declaration and definition. In the later case, the definition is treated as a declaration as above, and then as a definition. Processing of the definition requires the addition of the formal parameters to the current context, and then compilation of declarations and statements within the procedure, this is done by a recursive call to the compile-procedure routine. Once the procedure has been compiled to a circuit, it is stored in a list, and the start and end ports are stored back into the original procedure record. If there are any connections to procedure calls required, these are then made.

At present, type definitions are recognised, but cannot be used as the base type of a variable.

Expression compilation

Expressions are compiled in a simple, recursive fashion. The result of an expression compilation is the output port of the last operation. Since the expression tree is easily retrieved from the ASIS interface, the conversion of an expression to a circuit is done by case analysis on the type of the root node of the expression. There are four types of node that the compiler can deal with:

1. *Constants*
   A constant is constructed by tying the appropriate signals high or low. These connections often get optimised away by the mapping process.

2. *Variables*
   A simple variable is introduced into an expression by making a connection from the register that represents it to the expression tree. Array variables are connected in similar way, but they also need the index expressions to be supplied to the memory. This means that the index expression needs to be compiled, then connected to the appropriate address port.

3. *Operators*
   All operators are compiled to a block that performs that operation, this is connected to the output of the expression trees that form the arguments to this operator. Most operators are implemented in a single cycle, some need to be implemented as multiple cycles, and this can be achieved by adding more states to the state machine, and using these states to drive the registers storing partial results.

4. *Functions*
   A function call is implemented in a similar way to a procedure call. Since functions can take more than a single cycle to execute, they cannot be treated as a simple operator on their arguments.

At present, expressions are implemented to take a single cycle, however the structure of the implementation is such that multi-cycle expressions can be implemented.

Statement compilation

Statements are compiled into a set form, using template instantiation techniques, with the expression graphs from the statement being added to specialise the form. Each statements' circuit representation has the same interface, to allow for easy appending of statements. This interface consists of two signals, a 'start' signal, and a 'finish' signal. At present only assignments, while and for loops, if decisions and procedure calls have been implemented, though other statement forms can be easily included. Figure 3 shows the templates used for circuit generation, these are described below.

1. *Assignment*
   Assignment is the simplest statement to compile. The control structure consists of the single flip-flop required to keep the one-hot encoding of the state. The output of this flip-flop is connected to the data select for the appropriate value into the final register, and the write

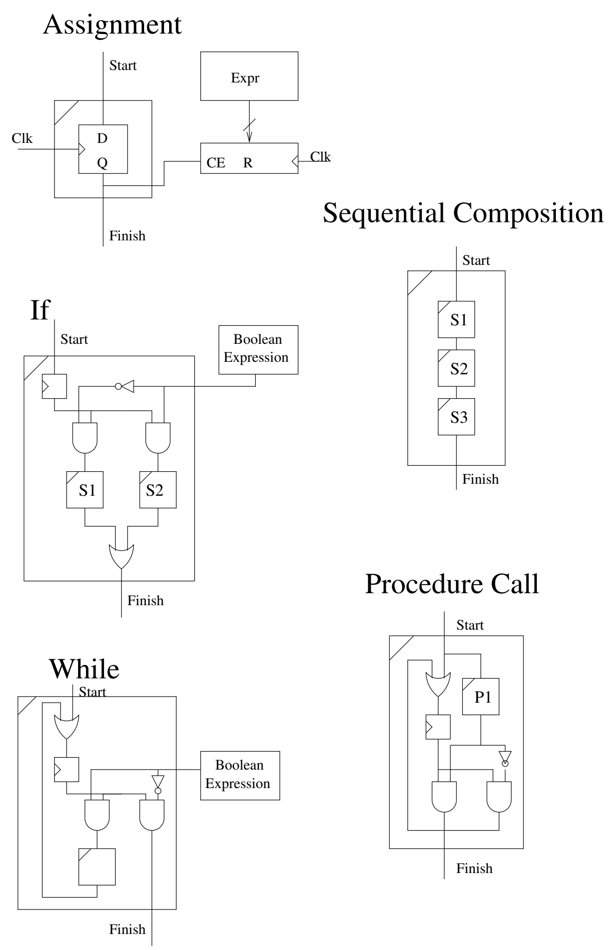

Figure 3: Circuit templates for statement compilation.

enable for that register. The output of the expression is connected to the same port of the selector. For multiple cycle expressions, additional flip-flops can be inserted, each controlling the appropriate parts of the expression, with the final one controlling access to the variable register.

2. *If decision*
   The IF statement implements a simple decision on the result of a boolean expression. The template introduces only one more flip-flop to the circuit, to allow for the calculation of the expression, the true and else paths of the statement are compiled separately and interfaced through their start and finish signals. An AND pair selects, according to the result of the expression, the path to be followed. The final OR gate allows a finish signal from either path to propagate to the next statement. At present, elsif paths are treated as a separate if statement.

3. *While Loop*
   The while loop template continually executes a sequence of statements until the controlling expression becomes false. Like the if statement, a single state flip-flop is added to allow calculation of the expression. An AND pair is then used to select whether the loop body or the following statement is started. In order to allow for the loop to be repeated, the finish signal of the loop body is ORed with the statement start signal before the expression calculation flip-flop.

4. *For Loop*
   The FOR loop template is largely based on the while loop template, since a for loop is a restriction of a while loop. There are two additions to the while loop. Firstly, there is an assignment prior to the OR gate of the while, which places the lower loop bound in the loop variable register. The other addition as another assignment, between the end of the loop body and the OR gate. This time, the loop variable is incremented. The loop expression is a comparison of the variable against the upper loop bound.

5. *Procedure call*
   During a procedure call, the calling procedure must wait for the called procedure to signal completion before continuing. Since recursion is not permitted in SPARK, the first completion of the called procedure will allow the caller to proceed. Before control is passed to the procedure, all the arguments to the call have to be assigned to the formal parameter registers. On the completion of the last of these assignments, the called procedure is passed a start signal. The calling procedure then enters a loop around a flip-flop, which is left through an AND pair when the finish signal is returned from the called procedure. The values of any out parameters are then copied back into local variables.

3.3 Circuit Generation

The graphical representation of the circuit allows the final output of the compiler to take on many forms, from a list of components and connections to a netlist that can be used in the production of a configuration file for a FPGA. The graph contains all the information required to produce the final circuit description. Whilst in the graph form, the circuit can be manipulated and optimised as necessary, before being converted to some form of output.

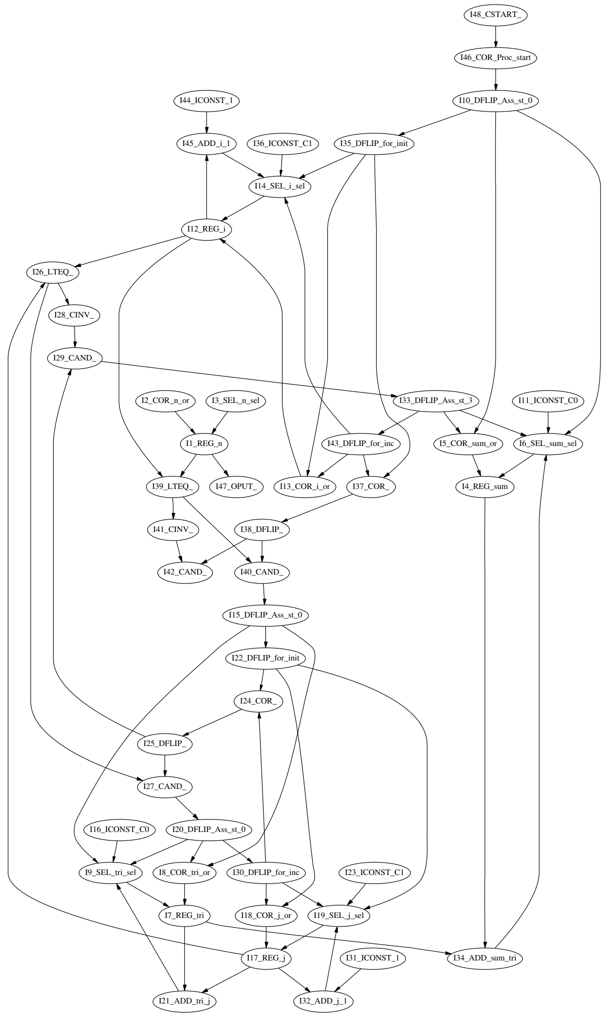

Figure 4: The circuit graph of the example procedure in figure 2.

Three output forms have been implemented:

1. *Text*
   This is a simple listing of components and their connections. This was used initially to check the correctness of graph generation, however, as circuit size rises, this rapidly becomes too complex.

2. *Graphical Output*
   This uses a graphing package to transform a textual description of the graph into a directed graph. An example of the style of graph produced is shown in figure 4, this is the circuit graph produced by compiling the example given in figure 2.

3. *EDIF Output*
   The circuit graph produced during compilation is converted to an EDIF netlist. The circuit is converted to a set of cell instances and connections between them.

Analysis	FPGA	Normal
Naive	1302	341,666
Model	1302	77,766
Path	732	44,602
Measured	732	13,122

Table 1: The WCET timings of FPGA and conventional implementations of the example procedure given in figure 2.

Each cell is also described in terms of logic primitives in the same way. EDIF was chosen as the output as it can be targeted at all the FPGAs currently available, so is not vendor or device specific, but still allows significant control of the placement of logic.

4. TIMING ANALYSIS

This section evaluates hardware compilation of the sequential subset of Ada in terms of the off-line timing analysis that has to be performed for real-time systems in order to prove their timing behaviour meets the specifications. This is done by comparing the WCET analysis of the FPGA and conventional implementations. For this evaluation we consider the compilation of the Ada procedure given in figure 2. It is assumed, for the purpose of this evaluation, that the value of `n` is never more than 20, enabling the maximum number of loop iterations to be known.

The comparison will be done across a number of timing analyses:

- *Naive*
  A naive worst case analysis assumes that the worst possible timings will always happen (e.g. every memory access is a cache miss, every loop executes the maximum number of times).
- *Model*
  A modelled worst case applies knowledge about the architecture to the timing analysis. This results in the effects of caches and pipelines being included in the final result.
- *Path*
  Semantic analysis of the program will result in improved knowledge of loop bounds and paths through the program. Together with modelling the architecture, semantic analysis produces a great improvement in the analysed WCET.
- *Measured*
  This is the average measured execution time of the longest path. This figure can also be analysed by taking a less pessimistic view of caching and pipeline effects.

The results of these analysis for both implementations are shown in table 1.

4.1 FPGA WCET Analysis

The state-machine style of implementation allows easy analysis of the WCET of the final circuit. This is due to the fact that the state-machine will always move onto the next state with each clock tick (unlike a processor that will take a variable amount of time to move onto the next instruction). In contrast to a normal processor the WCET of the compiled circuit is almost exactly the same as its average case (the difference comes from certain paths in if statements taking longer than others). This ignores the pessimism introduced from loop bounds not being known.

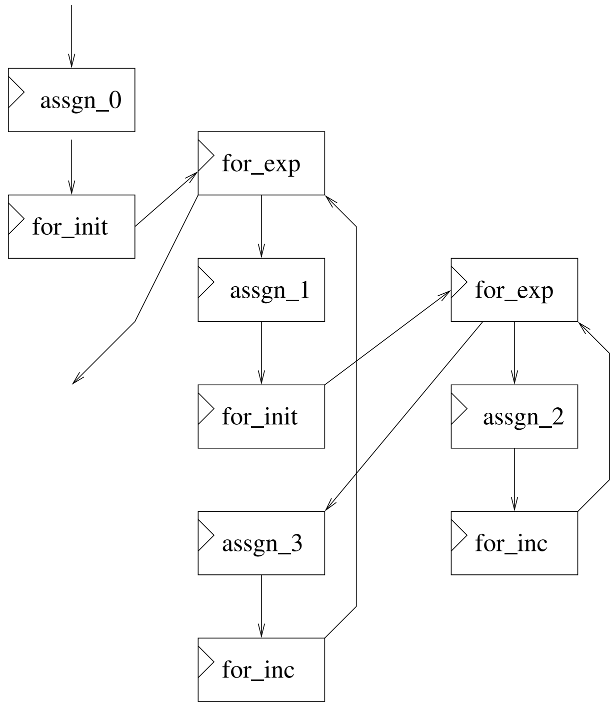

Figure 5: Reduced graph for the example procedure. Only the control flip-flops which regulate the timing of the circuit are shown.

Considering the procedure in figure 2, we can calculate the worst case execution time of the procedure in the following way. The circuit given in figure 4 is quite complicated to follow, so a simplified version, shown in figure 5, will be used instead. This has removed all but the clocked states of the circuit. These are arranged in three columns: the states related to the procedure, the states of the outer for loop, and those for the inner loop.

It can be seen that the inner loop will take three clock cycles per iteration (as there are three states in the inner loop. The outer loop will take five cycles, plus the time taken for the inner loop (which will be three times the number of iterations). The procedure itself will take 2 cycles, plus the time for the outer loop to complete its iterations. Given the assumption that `n` is never more than 20, the maximum number of cycles for the procedure to complete can be calculated as $2 + 20 * (5 + 20 * 3)$, to produce the value in the table. The semantics of the loop (the inner loop doesn't iterate 20 times each time, instead it iterates once, then twice...) can be, through path analysis [21][23], used to reduce this number to 732 cycles.

4.2 Conventional WCET Analysis

This can be compared to analysing a conventional implementation where the clock cycle count reached by adding up the times for each instruction will result in a figure that then has to be modified to take account of the architectural speed-up features of the target processor. Using the Intel 80486 architecture as an example, the inner loop takes 58 cycles, the outer loop 45 cycles, and the code outside the loop 46 cycles. A naive assumption to make is that no instructions are pre-fetched, and that every memory access is a cache miss. Assuming a cache miss cost of 30 cycles, a WCET of 341,666 cycles is reached.

A more realistic WCET can be arrived at through modelling the effects of the pipeline and cache. The size of the code means that there would be very few instruction cache misses, but data accesses cannot be modelled as accurately. Recent work at Princeton [16] and by Lundqvist[19] has shown that data cache accesses can be modelled; Lundqvist reported a hit rate of approximately 60% for most software. The number of jumps in the machine code negates the effects of the pipeline (as every jump requires the pipeline to be refilled) producing the result of 77,766 cycles. Including path analysis produces a further reduction in the WCET to 44,602. The measured WCET was calculated with the assumption that all the data values were in the cache, as were all the instructions, and the pipeline was kept full by (mostly) accurate branch prediction.

4.3 Discussion

Comparing the figures for the two different implementation methods we can see that there is a 60 fold increase in worst case clock cycles for the worst case path. Most of this apparent speed up is lost in the differences in clock period. Modern processors run at speeds of several hundred megahertz, the compiled FPGA circuits run at 20 - 30 MHz, depending on expression size and overall circuit size, approximately 25 times slower than the processors. The remaining 240% improvement is due to the much lower pessimism in the FPGA implementation. This shows that the performance of the FPGA can be accessed much more easily than that in a conventional processor for a real-time system.

Comparing the measured execution times it can be seen that the conventional implementation uses 18 times as many clock cycles as the FPGA implementation. This results in the faster execution of the conventional code (around 1.4 times faster). When concurrent programming facilities are implemented, then even implementing only 2 processes will result in the FPGA providing more performance.

It should be noted that the calculation of WCET for the circuit is done in terms of clock cycles, but the minimum size of the clock is dependent on the circuit. Large expressions cause the clock period to have to increase. Allowing the compiler to split up large expressions, making them multicycle will reduce the clock period, at the cost of making large expressions take two (or more) clock cycles to execute. The extra cycles these expressions will take to execute should be offset by the reduction in cycle time speeding up other parts of the circuit.

5. FUTURE DEVELOPMENTS

The current version of the Ada compiler supports integer and array types, and only integers as parameters to procedures. The sequential syntax that it implements is only a part of the SPARC subset. There are a number of extensions that can be made to the compiler:

1. *Types*
   SPARK has a range of types (integers, floats, characters, arrays and records) but the compiler currently only implements integers and arrays. The scalar types can be implemented easily, only recognition of types needs to be incorporated in the expression compilation to allow use of the correct type of the operator. Records can be easily implemented by treating each field as a variable in its own right.

2. *Sequential Constructs*
   There are two constructs missing from the compiler at present. Case statements can be implemented easily using a decode of the value of the cased expression. Exit statements provide an alternative route for control to leave a loop, this can be implemented with an extra connection and a couple of cells for each one.

3. *Tasks*
   At present, the compiler does not support compilation of tasks. Non-communicating tasks can be implemented by compiling each as a separate program, and putting all the circuits on one FPGA. Compiler support is needed to give communication and synchronisation between tasks. A task is essentially just a procedure, which already can be compiled, all that needs to be done is to implement the communications and synchronisations between tasks, and design a way to start tasks at the right time.

4. *Time*
   Delays are a large part of real-time systems programming. Their implementation requires a system clock to keep track of the passage of time (this could be just a counter counting at a known rate). Each delay would then use this central clock to time when it should continue.

6. CONCLUSIONS

This paper has introduced the need for a full language implementation of a real-time language on FPGAs. We have shown that a significant subset can be implemented, and that the rest of the SPARC subset can be added to the compiler without much modification. The analysable concurrency offered by the Ravenscar profile can also be added to the compiler without too much difficulty.

The timing analysis of the final implementation of a program has been shown to be much simpler than that required for a conventional processor implementation. Despite this, the available performance of the FPGA is much higher due to the much lower levels of pessimism in the WCET analysis compared to conventional processor implementations. It was also suggested that the concurrent implementations on FPGAs will give greater power than a conventional implementation.

7. ACKNOWLEDGEMENTS

This work was funded in part by BAE SYSTEMS through the Dependable Computer Systems Centre at the University of York and EPSRC. The authors would also like to thank Xilinx for their help with the provision of hardware used to test the output of the compiler.

8. REFERENCES

[1] *Ada Programming Language Reference Manual*. ANSI, 1983.

[2] *Ada 95 Reference Manual*. Intermetrics, January 1995.

[3] Ada Semantic Interface Specification (ASIS). Technical Report ISO/IEC 15291, IEC, 1999.

[4] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[5] G. Bollella, J. Gosling, B. M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-time Specification for Java*. Addison Wesley, 2000.

[6] M. Bowen. *Handel-C Language Reference Manual*. Embedded Solutions Limited, 2.1 edition, 1998.

[7] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Uppsala*, volume 1411, pages 263–275. LNCS, Springer-Verlag, 1998.

[8] A. Burns and A. Wellings. *Real Time Systems and Programming Languages*. Addison Wesley, 2nd edition, 1996.

[9] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *IEEE Computer*, 33(4):62–69, 2000.

[10] S. Franklid. Manual for VHDL++. Technical report, Dept. Information Technology, DTU Lyngby, 1999.

[11] D. Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs. In *FPGAs for Custom Computing Machines*, 1995.

[12] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the 16th Symposium on Real-Time Systems*, pages 288–297. IEEE, 1995.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[14] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[15] S. Holmstrom. SL - A Structural Hardware Design Language. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, volume 1673, pages 371–376. LNCS, Springer-Verlag, 1999.

[16] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Klewer Academic, 1999.

[17] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proceedings of the 19th IEEE Symposium on Real-Time Systems*, pages 334–345, 1998.

[18] W. Luk and S. McKeever. Pebble: A Language for Parametrised and Reconfigurable Hardware Design. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, volume 1482, pages 9–18. LNCS, Springer-Verlag, 1998.

[19] T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 255–262, December 1999.

[20] I. Page and W. Luk. Compiling Occam into FPGAs. In *FPGAs*, pages 271–283. Abingdon EE&CS Books, Abingdon, UK, 1991.

[21] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Paths. *Real-Time Systems*, 5(1):31–62, 1993.

[22] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.

[23] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, 1989.

[24] VHDL Language Reference Manual. Technical Report 1076-1993, ANSI/IEEE, 1993.

[25] Xilinx product information : http://www.xilinx.com/products, 2001.