Language Issues of Compiling Ada to Hardware

M. Ward*and N. C. Audsley
Real Time Systems Group
University of York
York, England

November 9, 2001

Abstract

Implementing real-time systems on FPGAs allows for much easier timing analysis. However, not using a von-Neumann architecture for implementation raises issues with the specification of the Ada language.

An alternative queuing and blocking policy is presented to allow priorities to be removed from the system. A new method of attaching variables to device registers using ports is presented. Changes to the interrupt handling mechanism made necessary by the freedom given by the change in implementation medium are discussed. These require a new library to provide the changes needed to target the implementation at FPGAs. Ways of indicating fine-grained parallelism at language level are examined to improve the parallelism used in the final circuit.

1 Introduction

The complexity of modern processors makes accurate Worst Case Execution Time (WCET) analysis difficult. It has been shown [5] that using Field Programmable Gate Arrays (FPGAs) [6] as the implementation medium can significantly reduce the pessimism in WCET calculations. Their structure also allow concurrent processes to be implemented in true parallel.

There are a number of references in the Ada language specification[1] to processor based implementations of the language. Some of these presume the attributes of a processor when defining the way the language should behave, and be implemented. This has no effect on the implementation of the sequential form of Ada, but does on the run-time used to control concurrency and the system integration aspects of the language. These are mainly representation clauses used to attach variables to device registers and the treatment of interrupts. Since the implementation medium can give truely parallel implementations, ways of introducing fine grained parallelism into the language are examined. Finally, the changes this makes to the analysis methods applied to high integrity systems is examined.

2 Run-time Issues

There are two main issues with the Ada run-time system:

1. Priorities
2. Queuing and blocking policies used for access into protected objects.

Priorities are used to order the execution of tasks on a processor, but in a truely parallel implementation, no ordering is required. The queuing and blocking policies used to control access to protected objects are designed for implementation on a von-Neumann architecture. By changing these, a more efficient implementation can be achieved.

*This work was supported by EPSRC and BAE SYSTEMS

2.1 Priorities

Priorities are mainly used to order the execution of tasks in a concurrent environment. Since the hardware implementation gives a truly parallel environment, priorities are no longer required for this purpose. This leaves the question of whether they are needed for other purposes. They can be used to regulate access to protected objects through the queuing protocols used to control the order of access, but, as noted below, this style of queuing is expensive in a hardware implementation. If some other form of queuing protocol is used, then the priorities are no longer needed in the system at all. This greatly simplifies the design process as there will be no need to assign priorities and prove that these priorities allow the task set to be scheduled.

2.2 Blocking and Queuing Policies

When accessing a protected object, a task will be queued or blocked if the it cannot gain access immediately. When trying to access an entry, the task will be queued. If it tries to access a protected procedure or function it will be blocked. This distinction means that they need to be treated differently.

Queueing

In both single- and multiprocessor systems, a queued task is suspended until it can gain access, and other tasks can be run in the meantime. The parallel implementation closely resembles the latter of these two types. The Ada reference manual defines two queuing policies - FIFO and ICP, and allows compilers to define additional policies. ICP is a priority based system, and given that priorities can be removed from the system it is preferable that this policy is not used. FIFO (First-In, First-Out) queuing allows the process that has been waiting longest to proceed when the resource becomes available. This policy has a bounded Worst Case Response Time(WCRT) (the sum of the WCRT of each process that accesses the protected object), but has a fairly high cost in terms of the logic required to implement it. This is due to it being a priority based method with the priorities being based on time (and therefore dynamic). This requires the construction of a priority encoder, able to cope with as many tasks as need to access.

A better solution for the queuing policy is required. This new policy must not have a worse response time than FIFO, but should be smaller to implement in hardware. One policy that fits these requirements is the round-robin schedule. In this, each process gets one access in turn, if a process is not queued when its turn comes round, then it is ignored, and the next process gets its chance. If a process joins the queue just after its turn is passed, then its response time (in the worst case) is the sum of the worst case response times of all the other processes. This makes it no worse than the FIFO queuing policy. The hardware implementation of a round robin schedule is much simpler than that for FIFO, and can be distributed around the calling processes. If no process is queued on, or executing in, the protected object when a call is made, then the calling process can be given access immediately. This is the most likely case of events, so should be as efficient as possible. Both the FIFO and round-robin approaches can be implemented to not delay the calling process, by overlapping the access contention logic with other parts of the calling process.

Blocking

The treatment of a blocked task differs between single and multiprocessor environments. In a single processor environment a scheduling policy is used to prevent the execution of a process that will block or could cause another process to block. In a multiprocessor

environment, a blocked process can spin-lock until the resource becomes available, preventing the execution of other tasks on the same processor. If blocking occurs for extended periods, tasks can start to miss their deadlines.

Use of a priority based scheduling policy requires that the processes are assigned a priority which can be used to schedule the access to the protected resources. By removing the need for scheduling, it had been intended to remove the need for priorities entirely, so a different solution would be preferred. If the implementation of tasks is treated as a multi-processor system with each task being placed on a single processor, then the use of spin-locks is less wasteful. This is because a spin-locking task can only delay itself, there being no other tasks trying to use the same computational resources.

As the access to a protected subprogram is no longer being controlled through priorities, there is a need to define an ordering of access for the blocked processes. For the multiprocessor environment, the order of access is undefined. It is possible, due to the properties of the hardware for a task to suffer starvation, if it is also beaten to gaining access by another task. This also raises problems when considering the WCET of a process, as it requires that all other tasks that access the Protected Object (PO) need to be considered in the calculation of the worst case access time to the PO. Using the round robin approach proposed above allows the access times to be bounded to one worst case access per task, and prevents the chance that a process can suffer starvation.

3 Representation Clauses

When trying to integrate a control program running on a processor into an application system, the program communicates with the hardware through interface logic. This logic generally has a series of registers that allow them to be controlled by the program. In order for the program to access these registers, the variables in the program need to be attached to them in some way. In Ada, this is done with representation clauses. These allow the way a variable is represented, and where it is stored, to be declared within the program.

Representation clauses that describe the way a variable is stored are easy to include in the hardware compiler. In fact, part of this is already implemented - variables are created only for the width they need to be, not to the word size of the target processor.

The problem with representation clauses is when they are used to attach variables to hardware registers. The Ada language specification requires that this be done using an address, which works well in a processor based implementation as the registers just have to appear in the memory map, but not in an FPGA implementation as these devices do not inherently have a memory map. There are a number of solutions to this problem:

1. A memory map can be included into the design of the circuit.
   This is a ratherly complicated solution, as memory access conflicts (multiple accesses can occur at the same time as the implementation is truely parallel, not concurrent) need to be resolved.
2. Allow the attachment of a variable to a set of the device pins.
   The variable can then be connected directly to the register. This removes the need for an address bus, at the cost of a much higher number of pins being used for access to registers (a problem with smaller devices that have limited I/O resources).
3. Many small memory maps can be included in the circuit.
   Most hardware devices have multiple registers arranged as a small block of memory (which makes them easy to fit into a memory map), which makes the previous solution hard to implement. Instead of linking a register directly to the device pins, it could be linked to a pair of variables, one acting as the data bus to the device, the

other the address bus, which would need to be given a fixed value. These variables would, in turn, have been attached directly to device pins as in the previous solution.

The preferred solution is (3). This encapsulates the second solution and extends it to allow for multi-register devices. This requires more code than the normal processor based solution, but this is mostly due to the need to name the pins that are connected to the device, whereas in a normal implementation these connections are predefined. Figure 1 shows an example of how a four register device can be connected into a program.

When connecting a variable to the system off-chip, a port has to be created first. This is a specific type, whose only purpose is to create a connection between the program and the outside world. A port will act much like an address, allowing variables to be connected to it. A new library `fpga` has been created to hold the FPGA specific packages needed for the changes suggested in this paper. The naming conventions of chip packages preclude the use of ranges in the specification of the pins to be used as physically adjacent pins can have wildly different logical names.

4 Interrupts

The Ada interrupt handling model attaches a parameterless protected procedure to a named interrupt. This procedure is then called whenever an interrupt arrives on the named line.

The names of the interrupts are implementation defined, as are the hardware interrupts the logical ones refer to. This works for a processor as the interrupt lines are predefined for the system. On an FPGA, the programmability of the device allows any of the device pins to be used as an interrupt line, and the signal that indicates an interrupt is also free to be programmed. This can result in large numbers of names being required to specify all the possible interrupts for the chip. For example, the largest Xilinx Virtex device has 804 user input/output pins, each of these can be configured to recognise pulses or level changes as the interrupt request signal, and the polarity of these changes can be configured, giving 3216 different interrupts that need to be named.

A solution to this name explosion is to change the way that the interrupts are defined. If an indirect definition is applied, as in the case of ports described above, then this can be greatly simplified. Instead of insisting that a name be defined by the implementation, if an interrupt type is provided, interrupts can be created by the source program, being attached to pins and given values defining the polarity and style of interrupt signal. These can be represented using a record, with pin, polarity and style fields, each with their own types. The pin type already needs to exist to provide for the creation of ports, the remaining two types are discrete with a very limited range of options. Figure 3 shows the definition needed in Ada.interrupts, and an example of its use. The package Ada.interrupts.names becomes unnecessary, leaving a null implementation.

The removal of priorities from the implementation has an effect on the behaviour of interrupts in the final circuit. In a processor implementation, each interrupt is assigned a priority. This means that an interrupt is only handled when it is the highest priority process in the system. The priority system ensures the following holds:

1. Only one interrupt can be handled at a time.
2. An interrupt handler cannot (by priority) run whilst any other process is accessing the protected objects.

With no priorities neither of these properties hold. In the first case, handling multiple interrupts at once should not cause problems as the circuitry is fully parallel.

The second case is more troublesome, removing priorities would break the mutual exclusive properties of the protected object. The solution to this discussed above is the round-robin locking policy. However the effects of this is slightly different to the standard

```
port_1 : fpga.port(8);
for port_1'pins use
   fpga.pins_to_port(p1, p3, p2, p5, p9, p8, p7, p10);
port_2 : fpga.port(2);
for port_2'pins use fpga.pins_to_port(p21, p25);
LCD_port : fpga.port(4);
for LCD_port'pins use fpga.pins_to_port(p45, p65, p47, p57);
debug : integer range 0..15;
for debug'port use LCD_port;

   -- packed record definitions of reg1 through reg4
   -- defined as for a processor based application

for reg1'data_port use port_1;
for reg1'addr_port use fpga.address_port'(port2, 0);
for reg2'data_port use port_1;
for reg2'addr_port use fpga.address_port'(port2, 1);
for reg3'data_port use port_1;
for reg3'addr_port use fpga.address_port'(port2, 2);
for reg4'data_port use port_1;
for reg4'addr_port use fpga.address_port'(port2, 3);
```

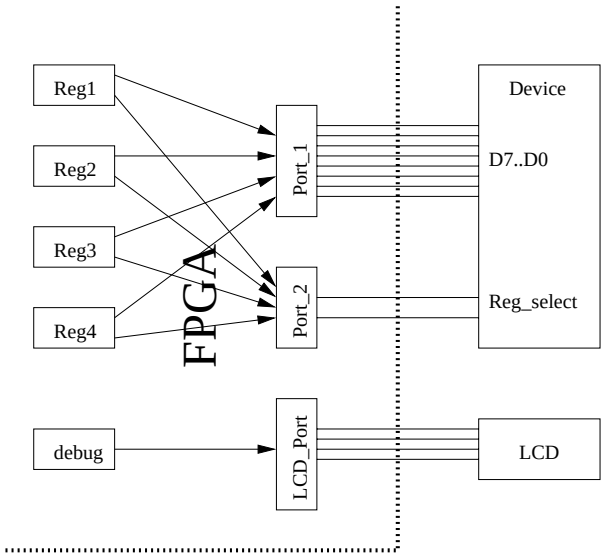

Figure 1: Connecting a device to a program using the preferred solution. The diagram shows the registers that are created from the code given above. The device has four eight bit registers, so needs an 8-bit data bus, and a 2-bit address bus, these connections are created as an eight-bit port (port_1) and a two bit port (port_2), which are connected to the specified pins. The 'reg' variables are then connected to these ports, the data being transmitted through port_1, and the value on port_2 being set as indicated during accesses. Since the LCD has only one register, no address port is needed, so the debug variable can be connected directly to the LCD_port port.

```
package fpga is

   type port is limited private;
   type pin_id is private;
   type address_port is
      record
         port : fpga.port;
         address : integer;
      end record;
   function pins_to_port(pins : pin_id) return implementation_defined;

private

   type port is implementation_defined;
   type pin_id is implementation_defined;

end fpga;
```

Figure 2: Contents of package fpga.

```
type interrupt_id is
   record
      pin : fpga.device.pin_id;
      polarity : fpga.device.int_polarity;
      style : fpga.device.int_style;
   end record;

pragma attach_handler(parameterless_procedure,
                      interrupt_id'(p47, positive, pulse) );

pragma attach_handler(parameterless_procedure,
                      interrupt_id'(fb21, negative, level) );
```

Figure 3: Definition and use of interrupt_id type in attach_handler pragmas

policy when an interrupt is received during the handling of that interrupt. When an interrupt is handled, a protected event starts, the handler will usually open a guard on an entry, and then exit. If there is a task waiting on the entry, then it will execute in the same protected event. The problem occurs if the interrupt is raised again at this point. In standard Ada, the handler would run again as soon as the protected event ends, as hardware priorities are, by definition, higher than software policies. With the round robin schedule, all other tasks wishing to access are given a fair chance to do so before the interrupt handler can.

The Ravenscar tasking profile restricts a protected object to one entry, and a single task queued on that entry, but this still leaves the problem of other protected procedures in the object. This can only be solved by changing the definition of the round-robin schedule, allowing the interrupt handler to run whenever the protected object isn't blocked. This reintroduces priorities, but in an implicit way that doesn't require the programmer to include priorities.

5 Device Dependencies

Conventional compilers are targetted at families of processors. These all have similar architectures, the main differences being the addition of extra instructions and widening of data paths. This means that the code generated will work on all members of the family. In a family of FPGAs however, the pins available for connections to the rest of the system vary greatly. The Virtex family have between 94 and 804 I/O pins, depending on the chip and package used, even different chips in the same packages will have different availabilities of pins. This great range of pin availability needs to be dealt with in some way.

Since the pin names are used in the definition of the connections for data ports and interrupts, these need to be defined in the program in some way. This needs to be done in such a way that only those pins available in the target device are defined. Including a series of libraries that define the pins, and other device specific attributes, for the programmer to with to give the correct pin definitions solves this.

There is another problem however. As the size of the devices change, so does the relative availability of logic resources. Without information pertaining to the size of the device, the compiler cannot optimise the generated circuits to fit the available resources. Also, the device families available from a vendor tend to use similar logic blocks, with only small differences between them. The same code generator can then be used for all families, if the compiler is made aware of the family it is generating for.

In a conventional compiler, the ability to change the family of a device is provided as a command line switch to the compiler. As there is already device specific information being included into the source, there can be problems of mismatches between the command-line switch and the device library included. This can result in circuits being generated that won't fit on any device due to pin name / logic use conflicts. It is proposed that a pragma be introduced into the device library that informs the compiler which device is being targeted.

6 Fine-grain Parallelism

One of the features of FPGAs is their ability to parallelise operations. Hence concurrent tasks can be implemented as parallel circuits. However, there is still a significant amount of fine-grain parallelism available within the sequential code of tasks. Potential savings can be partially realised by evaluating expressions in parallel, but there is often more parallelism available through executing statements in parallel. Detecting this parallelism through semantic analysis of the code is possible, but has the following problems[4]:

1. Compilers cannot spot all possible parallelism in the code, due to the possibility of contention between expressions. This means that certain gains possible from parallelism cannot be realised.
2. A compiler may try to parallelise a segment of code that must be done in sequence. For instance, two assignments to separate variables tied to control registers may have to be done in sequence in order to properly set up the device. Doing these assignments in parallel may cause the device to function incorrectly.
3. Parallelism can be defined using tasks, but the overheads involved in creating the tasks are prohibitively high for most fine grained parallelism.
4. Parallelism optimisations tend to happen at the intermediate or machine code levels. This makes some forms of parallelism hard to spot as the syntactic and semantic information has been (partially) lost.

There are a number of solutions to the problem of indicating parallelism in the program, with a range of impacts on the Ada language specification. These solutions are:

- Leave the specification as it is, and rely on compilers to detect all the parallelism in the program. This does not completely address problems 1, 2 and 4 described above.
- Introduce a pragma 'parallel' that indicates the statements that can be executed in parallel. This does not require a large change in the specification, and provides a means for the programmer to indicate which parts can be parallelised, and by default, those that shouldn't be. The statements immediately inside the parallel pragmas can be executed in parallel, as shown in figure 4. If the compiler cannot provide parallel execution, then the statements will be executed sequentially as normal. This addresses problem 2, but still leaves the problem of unused parallelism that the programmer hasn't indicated.
- Add two pragmas 'parallel' and 'sequence'. The parallel pragma would work as before, the sequence pragma would mark those parts of the code that should not be parallelised. This solves problems 1 and 2, helps with problem 4 and leaves the compiler free to parallelise much more of the program.
- Change the syntax of the Ada language to include a 'parallel' statement (as in Occam). This provides a syntactic basis for the detection and use of parallelism, allowing it to be controlled by the language specification. This does have some problems however. Due to the explicit nature of the parallelism, the specification for most of the sequential part of the language will need to be altered in the reference manual. This is to allow for the change in static semantics regarding the access of variables. Also this solution would require that existing compilers be rewritten to cope with the added syntax and change in semantics, whereas the previous solutions are pragmas that can be ignored by existing sequential compilers.

The preferred solution is the introduction of the two pragmas 'parallel' and 'sequence'. This allows the compiler to optimise (parallelise) most of the program only ignoring those areas specifically noted by the programmer. The statements immediately within the parallel pragmas can be executed so long as the variable accesses are safe (i.e. no two parallel branches should write to the same variable), and arrays need to be treated with care. Any code outside these pragmas can be parallelised according to current parallelising compiler techniques. These pragmas will also aid the parallelising compilers targeted at processors as they state explicitly where the parallelism is at the syntactic level, rather than trying to search for it at the machine code level.

This method does not require large changes to be made to the language specification, as the restrictions within the parallel pragma is such that the statements within the parallel block can be executed correctly either in parallel or sequence. Whilst the compiler can

```
-- Pre-code
pragma parallel(start);
  begin
    -- block 1
  end;
  begin
    -- block 2
  end;
  begin
    -- block 3
  end;
pragma parallel(end);
-- post-code
```

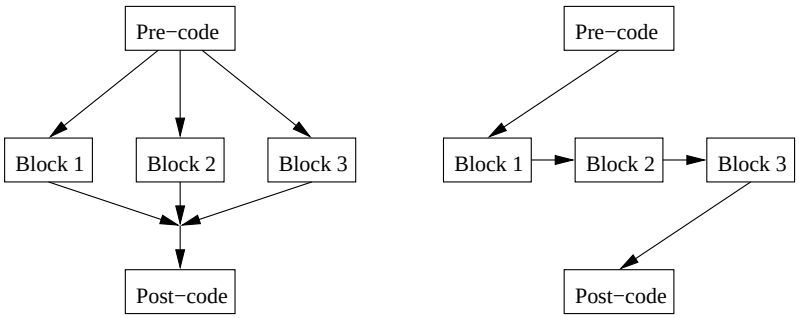

Figure 4: Parallel compilation example

do some checks on the correctness of the parallel sections (e.g. variable access), it is up to the programmer to write correct algorithms (as in most systems). Concurrent model checkers (e.g. the UPPAAL tool [3]) can be used to prove the correctness of the entire system, since the the parallel parts are merely fine-grained concurrency.

7 Ravenscar Analysis Tables

The ISO report on the use of Ada in high integrity systems [2] describes several analysis methods for the safety checking of code. Some of these relate to the program as written, others to the code produced once the program has been compiled. The former are not affected by the change of implementation technology, but the later are as there is no code as such in the output of a hardware compiler. These analysis methods are described below:

1. *Stack usage analysis* A normal processor based application uses a stack for storing local data of subprograms, the call history of the subprograms and for passing parameters between subprograms. The hardware implementation does not need a stack for these purposes, since the SPARK subset is non-recursive, instead creating a register for each variable. This removes the need for this style of analysis, but it is replaced with the need to place and route all the logic for the variables, which is also a static analysis.
2. *Timing analysis* The timing of the final hardware system can be derived in a similar way to that for a processor based system. The paths of the program can be found from its source. The basic blocks are timed by analysing the clock cycle time of the circuit, and counting the number of cycles required for each block. This process was described in more detail in [5].
3. *Other memory usage analysis* This analysis is concerned with ensuring there are no access conflicts to shared resources, e.g. memory and hardware devices. In the hardware implementation this is largely concerned with ensuring the pins used for connecting the program to the system are only used once, and not used by more than one process in the program.
4. *Object code analysis* Proving that the final circuit correctly implements the program will be easier than showing that compiled code implements the program. This is due to the use of template instantiation to create the circuit for each statement. The final circuit contains a template for each statement in the program, and the connections between these can be easily seen.

8 Conclusions

The use of FPGAs as an implementation medium for real-time systems allows much easier worst case execution time analysis, but raises problems with the Ada specification. These problems have been discussed, and preferred solutions for each presented.

- Making the implementation truly parallel removes the need for priorities. A round-robin schedule based on accesses was proposed to give fair access to protected objects.
- Having a non-von-Neumann architecture changes the way devices are accessed by the program. The address attribute has been replaced with port based attributes for connecting variables to device registers. This has also required the introduction of a port type.
- The change in architecture also changes the connection of interrupts to protected procedures, and the behaviour of the protected procedures.
- Ports and interrupts need individual library files to be defined for each device to provide port names and allow the compiler to optimise the circuit for the target device.
- Two pragmas have been suggested for deliminating the presence of fine-grained parallelism in programs.
- Alterations to the analysis of high-integrity systems were suggested.

References

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

[2] Guide for the use of the Ada programming language in high integrity Systems. Technical Report ISO/IEC 15942, IEC, 2000.

[3] H. Jensen, K. Larsen, and A. Skou. Scaling up UPPAAL - Automatic verification of real-time systems using compositionality and abstraction. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926, pages 19–30. LNCS, Springer-Verlag, 2000.

[4] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[5] M. Ward and N. C. Audsley. Hardware Compilation of Sequential Ada. In *Proceedings of CASES 2001 - to appear*, 2001.

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