Language Issues of Compiling Ada to Hardware

M. Ward** and N. C. Audsley

Real Time Systems Group
University of York
England

Abstract. A key issue in the implementation of real-time systems is the ability to statically analyse the timing behaviour of the system, to ensure that important timing requirements are met. The increasing complexity of modern processors make the determination of the worst-case execution time of software difficult via analytical means.
This paper discusses the compilation of Ada programs directly to hardware circuits, so removing the need for a processor. The solution is discussed in terms of Ada (specifically a Ravenscar / SPARK subset). The contributions of the paper are twofold. Firstly, the compilation of Ada to hardware (Field-Programmmable Gate Arrays) is described. Secondly, issues with the Ada language are addressed. These relate to processor concepts (eg. interrupts, representation clauses) that are embodied in the language that need to be modified for compilation to hardware.

1 Introduction

Real-Time Systems are those where the timing of an output is as important as the accuracy of its value. Often, these types of systems have to be proved correct in both the functional and time domains. Confidence in the timing of the software is achieved through static analysis[4]:

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.

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. Modern processors include speed-up features (e.g. pipelines, branch prediction units) that make

* This work was supported by EPSRC and BAE SYSTEMS

WCET analysis difficult. Despite much work on this (e.g. [8][12]) high levels of pessimism still plague the WCET analysis process.

This paper addresses the issue of pessimism reduction or removal by utilising general purpose hardware rather than CPU's for the target implementation. This change of implementation medium raises issues with the Ada language definition, which are discussed later.

1.1 FPGA Background

Field Programmable Gate Arrays (FPGAs) were first released by Xilinx in 1985[21]. An FPGA provides a grid of logic, memory blocks and connections with which a circuit can be built up (figure 1). Each logic cell can perform a number of logic functions, and also store some data in registers. The connections in the FPGA allow any logic cell to be connected to any other. These features allow complicated and large circuits to be programmed into the chip.

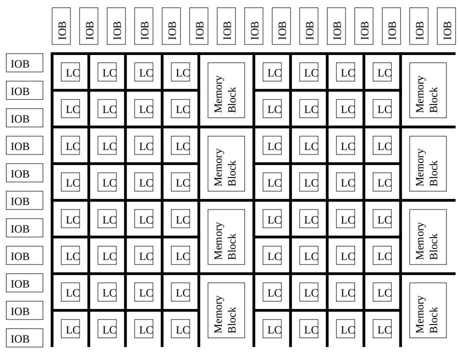

Fig. 1. FPGA structure. This shows the top left corner of an FPGA with memory blocks as well as logic cells. The programmable routing is shown as thick lines.

FPGAs can be reprogrammed in milliseconds, simply by changing their configuration bitstream. The bitstream is created by vendor tools from a high-level circuit description (e.g. VHDL).

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 [19], Pebble [13] and SL [9]) 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.

HDLs are unsuited to the implementation of software-based real-time systems. They have no communication constructs (though VHDL++ [6] proposes these), or timing facilities (e.g. delays and clocks) both of which are important to real-time programming languages [4]. HDLs provide coarse grained parallelism support through the circuit block structure.

There have been several implementations of HLLs for hardware compilation. Most of these are based on the C language (e.g. Handel-C [3], Transmogrifier C [7] and BRASS [5]), though Occam (Handel [16]) and Java (Xilinx Forge [21]) have been used. These languages are not suitable for real-time compilation as they do not provide the high-level communication and programming-in-the-large facilities required for real-time systems software.

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[4] including, syntactic support for the concurrency, communication and timing needs of a real time system. This paper considers the hardware compilation of Ada95, restricted by the Ravenscar (concurrent) and SPARK (sequential) subsets.

2 Ada95 Hardware Compiler

Figure 2 shows the structure of the hardware compiler. The four passes it takes are:

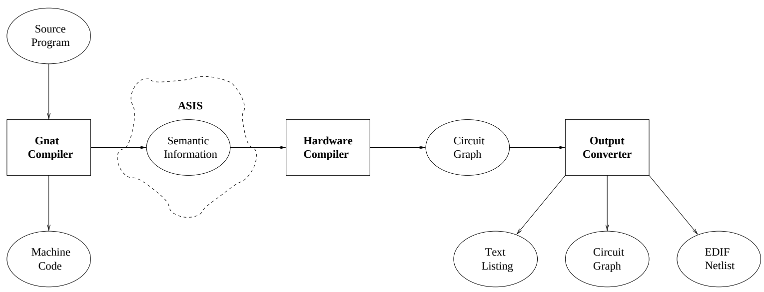

Fig. 2. Hardware compiler structure

1. *Parsing and Semantic Analysis* - The source program is compiled by the GNAT Ada compiler[1], which generates conventional code as well as the se-

[1] The GNAT Ada compiler is a fully validated Ada95 compiler, the public domain version is available from `ftp.cs.nyu.edu`

mantic information used by the Ada Semantic Interface Specification (ASIS) [2]. The latter allows access to the syntactic and semantic information produced by the existing compiler during the compilation process.
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 an EDIF (Electronic Design Interchange Format) netlist which can be converted to a configuration file for use on an FPGA.

A full description of the compilation process, can be found in [20].

3 Timing Analysis

This section compares the WCET analysis of the FPGA and conventional implementations. Consider the compilation of the Ada procedure given in figure 3. 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.

```
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;
```

Fig. 3. Example Ada procedure. This calculates the sum of all triangular numbers up to the given parameter.

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.
– *Expected* - This is the average expected execution time of the longest path. This figure is analysed with a realistic view of caching and pipeline effects.

The results of these analyses are shown in table 1. The clock cycle figures for the two implementations given in table 1 cannot be compared directly, as they run at vastly differing clock speeds. Instead, the timings need to be used.

Analysis	Clock Cycles		Time (ms)	
	FPGA	CPU	FPGA	CPU
Naive	1323	341,666	0.066	0.683
Model	1323	77,766	0.066	0.155
Path	753	44,602	0.038	0.089
Expected	753	13,122	0.038	0.026

Table 1. The WCET timings of FPGA and conventional implementations of the example procedure given in figure 3. The time figures use a 20MHz clock for the FPGA and a 500MHz clock for the processor.

3.1 FPGA WCET Analysis

In the FPGA 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), see table 1. The inner loop takes three clock cycles per iteration (as there are three states in the inner loop); the outer loop takes five cycles, plus the time taken for the inner loop to complete. The procedure itself will take 3 cycles, plus the time for the outer loop to complete its iterations. Given that `n` is never more than 20, the maximum number of cycles for the procedure to complete is 1323 cycles. The semantics of the loop can be, through path analysis [17][18], used to reduce this number to 753 cycles.

3.2 Conventional WCET Analysis

Analysing the conventional implementation can be done by adding up the times for each instruction, then modifying it 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 [11, 14] has shown that data cache accesses can be modelled. 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.

3.3 Discussion

It can be seen that the speed up of the FPGA over a CPU implementation (for the path analysis) is around 2.4 times. This improvement is due to the much lower pessimism in the analysis of the FPGA implementation. The fact that the measured timings are in favour of the processor implementation shows that the worst case analysis of the FPGA implementation is much easier to do accurately than a conventional implementation.

In table 1, the expected execution times suggest that the conventional implementation executes approximately a third faster than the FPGA. However, the FPGA allows true concurrency. Consider two concurrent copies of the procedure in figure 3, on a conventional processor, these would have to be executed serially, taking 0.052ms, whereas on the FPGA, two copies of the circuit can be constructed and executed in parallel, still taking only 0.037ms for both processes.

4 Ada Issues

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 when the implementation specific parts of the language are used.

4.1 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. The solution adopted is to include 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). A register is 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 are attached directly to device pins. Figure 4 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 extensions to the language. 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.

4.2 Run-time Issues

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 queueing protocols used to control the order of access, but, as noted below, this style of queueing is expensive in a hardware implementation. If some other form of queueing 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 set to be scheduled.

Blocking and Queueing Policies When accessing a protected object, a task will be queued or blocked if 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.

```
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.pis_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);
```

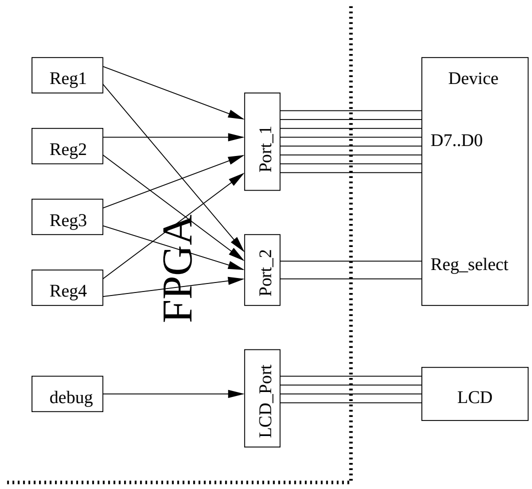

Fig. 4. Connecting a device to a program using the preferred solution. The diagram shows the registers that are created from the code given above. The device has four eight bit registers, so needs an 8-bit data bus, and a 2-bit address bus, these connections are created as an eight-bit port (port_1) 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.

The parallel implementation closely resembles the latter of these two types. The Ada reference manual defines two queueing 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) queueing 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 queueing 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 queueing 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 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.

4.3 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. In general, automatic detection of all potential parallelism in application code during compilation is not possible[15]. Application code must include indication of parallel operations.

For Ada, there are a number of solutions:

- 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 5. If the compiler cannot provide parallel execution, then the statements will be executed sequentially as normal.
- 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.
- 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

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

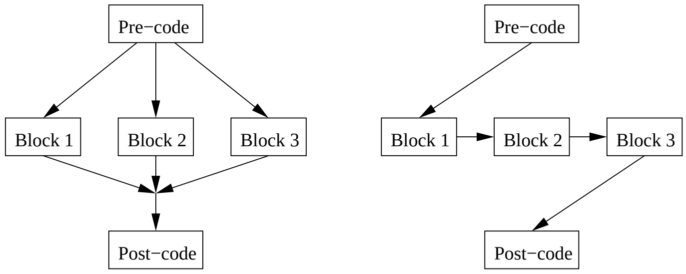

Fig. 5. Parallel compilation example

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 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 [10]) can be used to prove the correctness of the entire system, since the the parallel parts are merely fine-grained concurrency.

5 Conclusions

This paper has introduced the need for a hardware implementation of the Ada language on FPGAs in order to reduce the pessimism inherent in WCRT analysis. The benefit of this approach has been shown to be the reduction of the WCRT of a process to being less than a half of the processor based implementation.

Some of the issues that the hardware implementation raises with the Ada reference manual have been discussed, and solutions proposed to some of the problems. None of the solutions cause great changes to the Ada specification, but do provide substantial improvements in implementation efficiency for certain compilation targets.

References

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

2. Ada Semantic Interface Specification (ASIS). Technical Report ISO/IEC 15291, IEC, 1999.
3. M. Bowen. *Handel-C Language Reference Manual.* Embedded Solutions Limited, 2.1 edition, 1998.
4. A. Burns and A. Wellings. *Real Time Systems and Programming Languages.* Addison Wesley, 2nd edition, 1996.
5. T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *IEEE Computer*, 33(4):62–69, 2000.
6. S. Franklid. Manual for VHDL++. Technical report, Dept. Information Technology, DTU Lyngby, 1999.
7. D. Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs. In *FPGAs for Custom Computing Machines*, 1995.
8. 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.
9. 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.
10. 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.
11. Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software.* Klewer Academic, 1999.
12. 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.
13. W. Luk and S. McKeever. Pebble: A Language for Parametrised and Reconfigurable Hardware Design. In *Proceedings of FPL '98*, volume 1482, pages 9–18. LNCS, Springer-Verlag, 1998.
14. T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Proc. of the 6th Int. Con. on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 255–262, December 1999.
15. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
16. I. Page and W. Luk. Compiling Occam into FPGAs. In *FPGAs*, pages 271–283. Abingdon EE&CS Books, Abingdon, UK, 1991.
17. C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Paths. *Real-Time Systems*, 5(1):31–62, 1993.
18. P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, 1989.
19. VHDL Language Reference Manual. Technical Report 1076-1993, ANSI/IEEE, 1993.
20. M. Ward and N. C. Audsley. Hardware Compilation of Sequential Ada. In *Proceedings of CASES 2001*, pages 99–107, 2001.
21. Xilinx product information : http://www.xilinx.com/products, 2001.