Synthesis of Legacy Real-Time Ada Software to FPGA

N.C. Audsley and I.J. Bate

Real-Time Systems Group, Dept. of Computer Science, University of York, York, UK
Neil.Audsley/Iain.Bate@cs.york.ac.uk

Abstract. Automatic system synthesis techniques prevalent in the hardware / software codesign community have typically focussed upon relatively small self-contained systems that contain little or no legacy software. In many domains that can benefit from the concepts of codesign, the systems are much larger, with considerable legacy software that must be re-used in any future design.
This paper considers the use of codesign techniques in large real-time safety related embedded systems that contain large amounts of legacy Ada software, e.g. avionics and aerospace systems. Such systems are usually developed with an early (manual) hardware / software split, but need to continually evolve their functionality throughout the lifetime. This is usually achieved by expanding software functionality, with minimal architectural or hardware changes.
This paper argues that codesign principles should be utilised during system evolution to rapidly search for system architectures and software configurations that can be mapped to a flexible target hardware architecture, ensuring that all non-functional requirements (safety, real-time, etc.) for the system are met. Specifically, the paper considers the case of legacy Ada, showing how it can be mapped to a flexible FPGA based architecture, as part of a codesign process that allows easy re-mapping should the Ada code evolve.

1 Introduction

In many domains new systems must be built using legacy software – eg. upgrades of avionic or aerospace systems, where it would be impractical (for reasons of certification cost, development cost and timescales) to completely re-write all software. This paper considers the re-hosting of legacy software typified by large hard real-time Ada based systems [1]. Such software is relatively constrained in terms of its funcionality and interactions (both within the software and with the environment), but is designed and implemented to be predictable in terms of its run-time behaviour.

For hard real-time systems, timing predictability is key – failure of the system to meet timing requirements (eg. process deadlines) can result in catastrophic failure of the system. Thus, it is important to be able to show that all timing requirements of the system are met prior to run-time. One approach is to utilise

static offline timing analysis [2]. This proceeds by extracting a model of the key timing properties of the system then calculating the worst-case timing behaviour of the system. If the timing properties of the system are met in the worst-case, the system will meet its timing requirements at run-time (assuming that the implementation does not invalidate any assumptions made in the model). Note that extensive testing of an implementation does not necessarily cover the worst-case.

Clearly, the legacy software could be re-hosted on a conventional CPU. However, timing analysis for systems implemented on conventional CPUs is pessimistic, in that the analytical worst-case (in terms of timing behaviour) of an individual process (and of the entire system) is usually a great deal worse than the actual worst-case seen at run-time. The major causes of pessimism in timing analysis are process scheduling (large variations in the elapsed time between a process starting and completing for each of its invocations (whether periodic or otherwise) [2]); worst-case execution time (WCET) analysis (WCET is accurate for older processors (eg. 68000) which have no architectural speed up features whilst for modern processors with architectural speed-up features (eg. pipeline, cache), accurate WCET is difficult, with a large difference between the analytical WCET and the actual WCET[3][4]); and resource sharing (where processes need to synchronise access to some shared resource (eg. data or physical resource) blocking may occur if a process wishes to access a shared resource in a mutually exclusive manner).

This paper considers the re-hosting of legacy Ada software onto a Field-Programmable Gate Array (FPGA). The use of FPGAs as an implementation target for hard real-time systems can remove much of the pessimism in timing analysis. In [5, 6], an approach is described for directly compiling real-time programs written in Ada to a hardware circuit. Execution of the application is then achieved by executing the circuit downloaded to some reconfigurable hardware (the approach uses FPGAs). This approach avoids the problems of WCET calculation due to CPU architecture[1]. It also enables Ada tasks to be truely parallel, so removing problems of scheduling. However, this approach is not scalable, in that for large Ada programs, extremely large FPGAs would be required. One solution would be to target a number of FPGAs from one Ada program. Alternatively, this paper investigates the mapping of parts of an Ada program to hardware circuit, the remainder being compiled to CPU instructions to be executed on a FPGA hosted CPU cores. The paper combines offline timing analysis with conventional codesign principles to provide a process that is to find a mapping of Ada to FPGA such that timing requirements are met according to static timing analysis. It is noted that whilst the focus of this paper is upon Ada, the general approach described is applicable for most imperative languages, particularly those that are concurrent.

The remainder of this paper is arranged as follows. Section 2 provides general background on Ada and FPGAs. An overview of the method proposed is given

[1] Noting that control-flow path anlaysis problems remain.

in section 3, with evaluation following in section 4. Finally, previous work and conclusions are given in sections 5 and 6 respectively.

2 Background and Context

This section provides background on Ada and FPGAs. Note that Ada has been, and is still being, widely used in the implementation of safety-critical systems, eg. avionics / aerospace. There is considerable interest in these industries for the re-hosting of legacy Ada code onto FPGAs as part of a cost-effective upgrade path.

2.1 Ada for Hard Real-Time Systems

The Ada language [7] facilitates the programming of real-time systems. It contains facilities for programming-in-the-small (ie. sequential programming), facilities for programming-in-the-large (ie. data abstraction and packages), together with facilities for concurrent programming (ie. tasks and inter-task communication). In addition, standard subsets of Ada have been developed such that programs conforming to the subsets are guaranteed to be statically analysable for timing, resource and functional properties.

The SPARK subset of Ada [8] restricts the sequential part of the language. Conformant programs can be proved (partially) correct. SPARK does not contain any dynamic constructs, including concurrency (and synchronisation), access (pointer) types and variant records (hence no object-oriented capabilities). Subprograms 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. The SPARK Ada subset is consistent with the requirements for real-time system timing analysis in that all conforming programs are statically analysable for their worst-case properties.

The Ravenscar tasking profile[9] is a statically analysable tasking subset. Unlike full Ada, 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. The definition of Ravenscar will become part of the ISO Ada standard at the 2005 standard revision. It will become part of Annex H (Safety and Security) which comments on applicability of Ada language features for use in safety-related systems.

A SPARK / Ravenscar conformant Ada program consists of a number of concurrent tasks, that interact via protected objects. These objects enforce mutual exclusion over some procedures and associated data within the object. Interaction with other devices is achieved by representation clauses, which associate a specific memory location with a program variable, so achieving a memory mapped programming model. Also, conformant programs are analysable for timing (and other statically determinable) properties.

2.2 Compiling Ada

Conventional compilation of Ada to CPU instructions follows the normal compilation path [10]. Note that the concurrent features of Ada require a run-time support system (or kernel) to be present when executing the code. One function of the run-time is to provide scheduling between the different application tasks. Given the restricted concurrency model of Ravenscar conformant programs, the run-time required for such programs is simple – indeed, a simplistic run-time was one of the prime motivations for the Ravenscar subset in order to reduce the cerfication cost of tasking Ada programs in safety-critical systems.

SPARK / Ravenscar conformant Ada programs are ideal for direct compilation to hardware circuit. In [5, 6, 11] an Ada compilation process is described for such programs, targetted at FPGAs. Essentially, concurrency within Ada can be represented on hardware as truely parallel tasks. In terms of the Ravenscar tasking subset, the main implication is that task scheduling is no longer required – indeed, no run-time is required at all. The sequential language used within a task is relatively straightforward to compile to hardware, as the restrictions of the SPARK subset ensure that no dynamic statements are present in a task.

Protected objects enforce mutual exclusion over some procedures and associated data. Hardware compilation does not remove the need for mutual exclusion, so protected objects remain. When contention exists over access to a protected object, the default locking policy of Ada is used, that is ceiling protocol [12], where ceiling priorities are defined in terms of the priorities of the tasks that access the object.

The compilation process is illustrated in Figure 1. The syntax and semantic analysis phases are performed by a public domain GNAT Ada compiler. This compiler implements the Ada Semantic Interface Standard (ASIS) which enables third-party tools access to the decorated abstract syntax tree generated during compilation. The hardware compiler developed at York (and detailed in [5, 6, 11]) consists of two main phases:

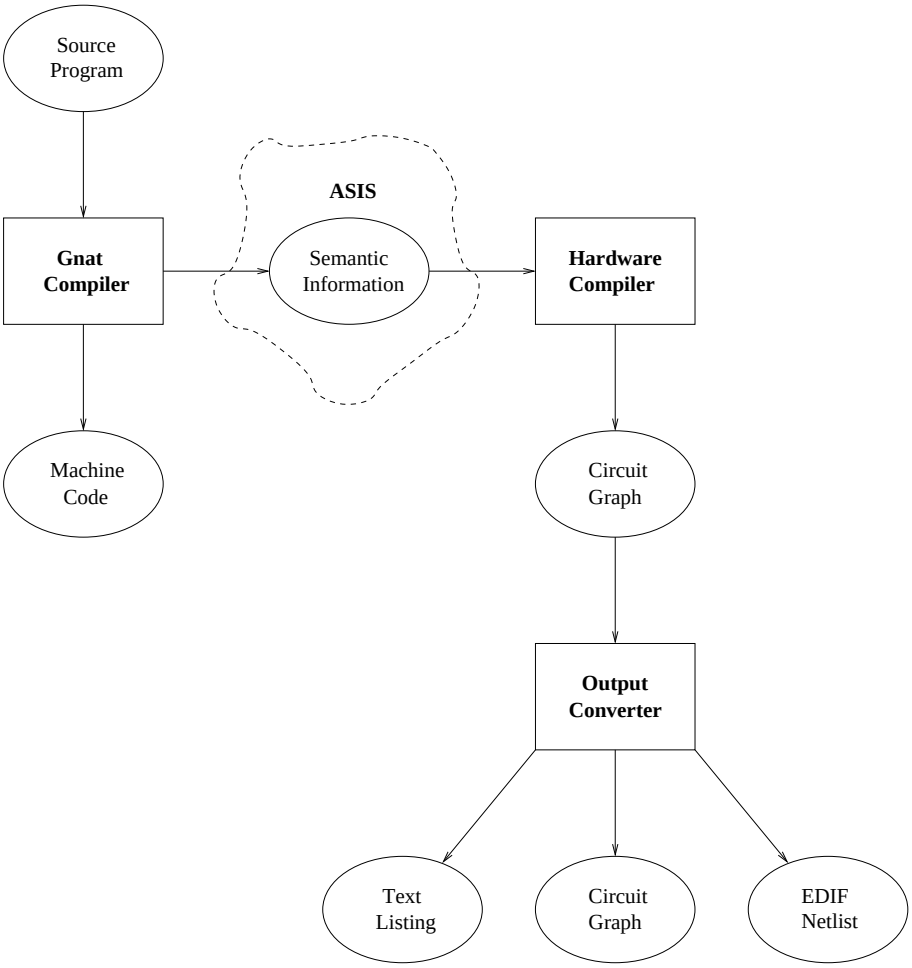
1. the hardware compiler walks the decorated abstract syntax tree (via the ASIS interface) applying hardware templates for each of the language constructs encountered, forming a circuit graph representing the entire Ada program;
2. the circuit graph is converted into either device dependent EDIF for later translation to an FPGA bitstream; to text for debugging; or to a pretty-printed graph.

The conversion from EDIT to the FPGA bitstream is achieved via commercial toolsets, specific to a particular FPGA.

2.3 Timing Analysis

Static timing analysis of Ada systems is based upon response time analysis for fixed priority pre-emptive systems [13] (noting that the Ada language prescribes the fixed priority pre-emptive scheduling policy). This requires each task period, deadline, WCET and maximum blocking time to be known. Maximum blocking

Fig. 1. Hardware Ada Compiler



times can be calculated, given a priority ordering of the tasks. Response time analysis can cope with any priority ordering required, although best results occur with deadline monotonic priority ordering [2]. For the purposes of this paper, the priority ordering used by the analysis is that defined within the Ada source program.

The analysis derives the response time of a task, under the worst-case system loading. If the response time of all tasks is no greater than their respective deadlines, the system will meet its timing requirements at run-time – the system is termed *feasible*.

2.4 FPGAs

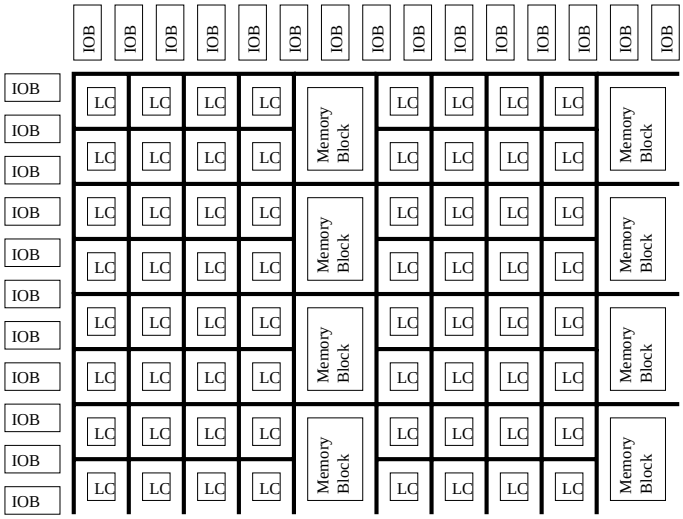
Unlike the program once, logic-term PAL / GAL devices, Field Programmable Gate Arrays (FPGAs) provide reprogrammability and logic functions to the system designer [14, 15]. This allows them to provide more complex functionality than an equivalently sized programmable logic device. A typical sea-of-gates FPGA consists of an array of functional blocks, connected by programmable interconnects (see Figure 2). Functional blocks contain the logic elements of the design. Views on granularity of block functionality differ among FPGA families but overall device functionality is the same. Programmable interconnects route signals between logic blocks, and come in varying lengths to suit the range of connection distances required. By connecting logic blocks, larger logic expressions are realised. Typical actual devices may involve over 100,000 functional blocks and also provide considerable memory blocks every few columns of functional blocks (upto 4 MBytes on large devices).

I/O blocks, around the edge of the FPGA, provide the means to interface to/from external signals. Typically FPGAs are I/O pin rich (upto 500 I/O pins per device, depending upon the exact packaging). Most FPGAs support a wide range of bus and interface standards so can be placed directly on a bus without any glue logic. Data storage forms a large part of most systems, and so to keep data access speed high, FPGAs generally contain some form of memory. Most large devices include fairly large (2-8 kbit) RAM blocks around the array, configurable in the word width of data. In some devices smaller memory blocks can be constructed out of logic blocks, for local storage of small data items.

The program for an FPGA can be specified in languages such as Ada [5, 11, 6], Handel-C [16], VHDL. In any case, the program is synthesised to a *net list*: an abstract circuit description. This is fitted to the device it is to be implemented on. Finally the fitted design is converted to the configuration bit stream used for programming the FPGA. FPGAs are reprogrammable; configuration takes some milliseconds.

Due to the parallelism of the FPGA, different parts of the original program actually execute in parallel at run-time, on the same physical device (cf. mul-

Fig. 2. FPGA Structure



tiprocessors where many operations can occur in parallel, but not on the same device [2]).

In a simple system, each process (ie. Ada task) in the source program can be compiled to a separate circuit and allocated a dedicated area of the FPGA on which to execute – thus all processes execute in parallel. In conventional CPU scheduling theory terms, this is similar to the scenario where each process executes upon a separate processor, although inter-processor communication times and communications contention must be taken into account. On an FPGA, the inter-process communication times are extremely short (as they are all on the same physical device). An additional factor that must be considered is the allocation of the space on the FPGA – the device has finite size.

Given that an FPGA has finite size, there is a limit on the amount of area that can be allocated to processes for execution. At this stage, there is the possibility of introducing softcore processors onto the FPGA which can then execute a number of software tasks. There is a net saving of FPGA space if the tasks now implemented in software take more room (in total) than the softcore processor, should they be implemented directly in hardware. Note that many softcores can exist on the same FPGA, perhaps if simpler (potentially slower) softcores were used.

[2] This includes devices which contain multiple processors, as the degree of parallelism is limited to the number of separate processing elements. An FPGA can have millions of separate small parallel elements. [14].

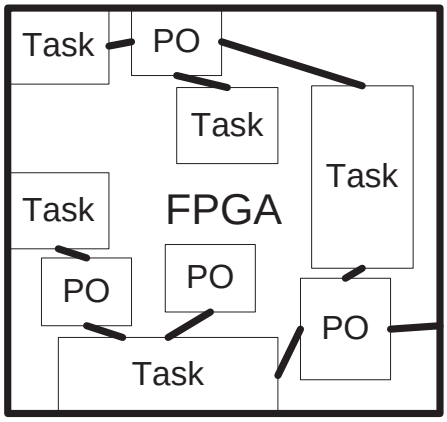
2.5 Target Architecture

The physical target architecture assumed is that of a single Field Programmable Gate Array (FPGA) [14], coupled to a number of RAM banks. Clearly, limiting the target architecture to a single FPGA restricts the solution space. However, the physical size of FPGAs can be large (upto a billion transistors), ensuring that substantial functionality can be achieved on a single device. Also, the presence of the RAM banks ensures that (parts of) the FPGA can be used for softcore CPUs, further extending the size of the functionality that can be implemented upon the target.

The softcore CPU used within this paper implements a 68020 architecture in a reasonably small area (around 1000 LUTs on a Xilinx Spartan device). The penalty for such a minimal implementation is that the maximum clock speed is relatively slow (10MHz, with each instruction taking around 5 cycles). Note that no architectural speed-up features are implemented, enabling accurate WCET of any software component compiled to the processor.

In terms of the compilation of Ada to the target architecture (as introduced in section 2.2), there are a number of approaches. Firstly, Ada tasks and protected objects compiled straight to circuit, with a dedicated area of FPGA assigned. This is illustrated in Figure 3, noting that I/O outside of the FPGA occurs via the protected objects (not the tasks).

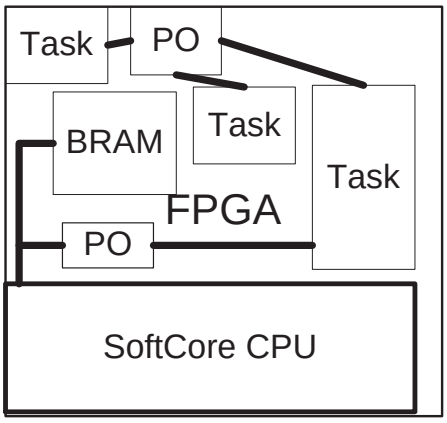
Fig. 3. Basic Mapping of Ada to FPGA



Given the finite area of the FPGA, as the source program size increases there is a need to use softcores to execute some of the code (assuming the implementation target remains as a single FPGA). This is illustrated in Figure 4, where the softcore CPU utilises on-chip RAM (Block Ram or BRAM) for instruction / data

storage. Communication between directly compiled elements and the tasks and protected objects executing on the softcore processor takes place via a protected object accessible from the softcore bus.

Fig. 4. Mapping of Ada to FPGA using Softcore



As the amount of BRAM is limited (around 100Kb is not uncommon on devices), external RAM is required for task and protected object instruction / data storage. This is illustrated in Figure 5. This is illustrated in Figure 5 where a driver for the external RAM is placed onto the FPGA, providing the appropriate data, address and control signals over the external I/O lines to the RAM.

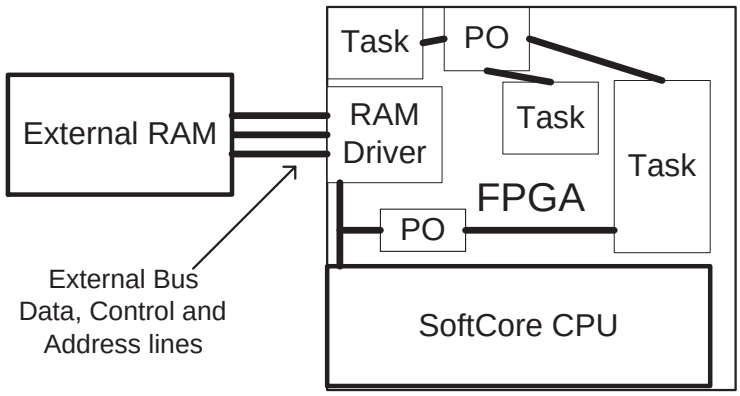
Clearly, a combined approach, using both BRAM and external RAM can be used. In this architecture, the BRAM can be used as instruction / data cache (eg. see the OpenRISC softcore [17]).

Note that the last mapping described forms the basis for the mapping used in the approach for mapping legacy Ada to FPGA described in the remainder of this paper. Initially, each task is assigned its own softcore processor, with tasks moved out into hardware to balance the use of the FPGA. Usually, the method concludes with an architecture similar to that in Figure 5, with one (or more) softcores running tasks and protected objects, some placed into the hardware directly.

3 Overview of Method

The method follows a timing analysis driven approach, as motivated earlier in this paper. The timing characteristics of an Ada program are modelled sufficiently for analytical timing analysis to occur. The actual implementation of the

Fig. 5. Mapping of Ada to FPGA using Softcore and External RAM



system is then checked against the assumptions of the model. If the assumptions still hold (eg. that the WCET of a software task is no more than some value), then the full implementation will meet its timing requirements.

The method is illustrated in Figure 6. It consists of an iterative process with two main parts:

1. Modelling and simulating (at a high level) the timing and interaction properties of the software.
2. Compilation to hardware circuit and CPU instructions of a given allocation of the software.

These stages provide feedback in terms of timing characteristics of the actual software (eg. WCET of software tasks, or circuit speed and size of an FPGA task); analytical timing analysis; and simulation of the system. This is sufficient for the system configuration, in terms of the allocation of tasks to hardware or software, to be evaluated. As a consequence, a new allocation can be determined to further improve the system.

The psuedo-code of an algorithm is given in Figure 7 that describes the interaction of constituent parts of Figure 6. The algorithm starts with a NULL allocation, which represents the scenario where all of the Ada program is allocated to hardware. Whilst this allocation meets the timing requirements (ie. the system is feasible), it assumes a sufficiently large target FPGA. This restriction is removed by the iteration of the algorithm, which refines this allocation to one where the system fits into the resources specified, whilst still meeting its timing requirements. If no such allocation can be found, the algorithm will terminate suggesting an allocation that requires too much resource, although meets the timing requirements.

The method is detailed further in the following sections.

Fig. 6. Overview of Process.

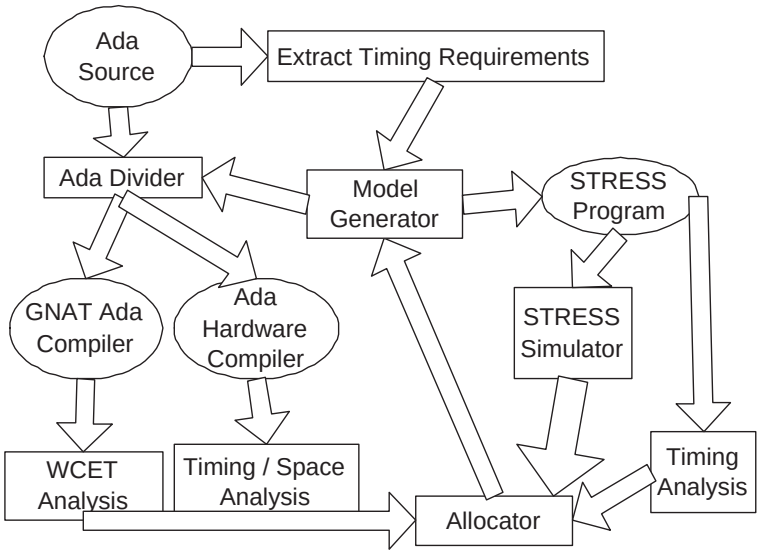


Fig. 7. Algorithm for Allocation.

```
Allocation = NULL
Extract timing requirements from Ada
do
{   Divide Ada according to current allocation
    Compile Ada software
    SW_Results = Analyse software for WCETs
    Compile Ada hardware
    HW_Results = Analyse hardware for time & space use
    Generate STRESS program from allocation and timings
    STRESS_results = Simulate STRESS program
    Timing_results = Timing analysis of STRESS program
    New_allocation = Allocate (SW_results, HW_results,
                        STRESS_results, Timing_results)
} while (Allocation != New_allocation)
```

3.1 Extract Timing Requirements

Initially, the Ada software is analysed to extract the key timing and interaction properties. These form the timing requirements of the system. This is possible as the software is written to conform to the SPARK and Ravenscar subsets. The timing property of a task can be established (by inspection, or automatic analysis) due to the programming conventions employed in the software. For example, a periodic (or time-driven) task which is executed every "period" time units is always of the form:

```
task Periodic_Task body is
  -- declarations
begin
  -- task specific initialisation code
  begin loop
    -- code for task
    X := X + period ;
    delay until X ;
  end loop ;
end task ;
```

Likewise, a sporadic (or event-driven) task can be detected as it waits for its initiating event within a protected object, which is essentially a shared resource providing mutually exclusive access to the functions and data within it. For example, a task which waits for an event before executing is of the form:

```
task Sporadic_Task body is
  -- declarations
begin
  -- task specific initialisation code
  begin loop
    Protected.Wait_for_event;
    X := current_time ;
    -- code for task
    X := X + minimum_time ;
    delay until X ;
  end loop ;
end task ;
```

In the above, the call to the protected operation to wait for the event will only return when the event has occurred. In addition, note that a minimum time between executions of the sporadic task can be specified by delaying at the end of the loop in a similar manner to that of the periodic task above. Further details of standard forms for tasks and protected objects for Ada based hard real-time systems are available in [1].

Estimation of execution time of an Ada task is rudimentary, based on a the number of source code lines. Accuracy at this stage is not required, as subsequent stages of the method may change the WCET estimation, whilst ensuring that

the system timing requirements are still met. Also, the choice of softcore processor can be varied to improve WCET estimation (eg. complex pipelined processor replaced by non-pipelined); or WCET times themselves can be reduced by increasing the clock speed.

3.2 Model Generator

Having extracted the timing and interaction properties from the Ada source software, these are now modelled as a STRESS program. The STRESS language [18] was developed for modelling and simulating real-time systems at varying levels of abstraction. An Ada task is represented as a STRESS process and a protected object using STRESS semaphores. An example is given in Figure 8. In the figure, the timing properties of the STRESS process are specified, the values used being those extracted from the timing requirements of the Ada program – this includes the same priority values as used in the Ada program. Thus, the period specified for the STRESS process has an identical value to that used in the corresponding Ada task. The worst-case execution time of the STRESS process is broken into the timing statements (in square brackets).

Fig. 8. STRESS Program.

```
system
  node node_1
    processor proc_1
semaphore S0
      periodic J0
  period 15 deadline 15 offset 4
        priority 1
    [1,1]  p(S0) [1,1] v(S0) [1,1]
endper
periodic J1
  period 20 deadline 20 offset 0
        priority 2
    [2,2]  p(S0) [4,4] v(S0) [1,1]
  endper
    endpro
  endnod
endsys
```

It is possible to add far more detail to the STRESS program, as the language allows modelling of the kernel (including scheduling and resource management policies) using a rudimentary system of variables, types, expressions and control-flow statements. These facilities are currently used only to model (at a coarse

level) the behaviour of the run-time system underpinning a SPARK / Ravenscar compliant Ada program. Further examples of the power of STRESS are given in [18].

A key feature of STRESS is that the architecture of a system can be represented. This allows the allocation of Ada tasks and protected objects to hardware and software to be modelled. This is shown in Figure 8 as "node" and "processor" blocks – the allocation in the figure is such that both processes are on the same processor. Note that other system specification languages, such as SDL [19], do not enable easy modelling and simlulation of software entities such as kernels (including potentially changing the scheduling and resource management policies).

3.3 STRESS Simulator

The discrete event STRESS simulator [18] simulates the system as represented in the STRESS program. The goal of this simulation is to examine the patterns of timing behaviour seen at run-time. Initially, this is not particularly interesting, as the initial model has maximum parallelism (ie. one STRESS process per processor), reflecting an implementation where each Ada task is truely concurrent – so representing the allocation of all Ada tasks to hardware. As the STRESS model changes, perhaps with multiple STRESS processes on a processor, the simulation will encompass the scheduling and resource management of the STRESS representation of proteected objects.

The output from a STRESS simulation is a log file of events that occurred in the simulation. This details which process was executing on each processor at any time. Also, missed deadlines are indicated, together with details over resource usage (ie. STRESS equivalent of Ada protected objects) and any blocking. This information provides an indication of the average case behaviour of the system, in so far as how close processes get to missing deadlines, together with an indication of the cause of any bottlenecks in the system. Essentially, a degree of sensitivity analysis is achieved.

Simulations can also be examined graphically, although this is of limited use within an overall automated system sythesis process.

3.4 Ada Divider

The Ada divider takes the original Ada source code and modifies it such that part of it is compiled by the software Ada compiler and part by the hardware Ada compiler. This is achieved by examining the current model (ie. as defined by the Model Generator) and marking the source Ada with appropriate "pragma" clauses. In the Ada language, "pragma" clauses are compiler directives. Two new pragmas are defined in order to achieve the compilation required. The "pragma hardware_compile" directs the hardware compiler to compile, and the software compiler not to compile; the "pragma software_compile" directs the software compiler to compile, and the hardware compiler not to compile.

Currently, the pragmas are applied at a task / protected object / shared procedure granularity. Stubs are inserted within the "software_compile" parts of the Ada, wherever calls or interaction with "hardware_compile" components occurs. This eases subsequent linking of software object components in particular.

3.5 Compilation

The compilation of Ada to binary (ie. the software route) utilises the GNAT Ada compiler [10], noting that the target is the simple processor architecture suggested in section 2. The compilation to hardware is achieved using the hardware Ada compiler described in section 2.2.

Splitting the compilation of a single Ada program into two parts can lead to linking problems of the software, and of system integration issues between the software components and hardware components on the actual target. In section 3.4 it was noted that stubs are used in the "software_compile" units to ease this problem. These map software calls onto memory mapped locations (utilising Ada representation clauses). The integration process of hardware and software must ensure that the hardware referred to by the software call is present at the correct memory location.

3.6 Timing and Resource Analysis

The timing and resource analysis occurs in three places:

1. *Worst-Case Execution Time Analysis of compiled software components.*
   This analysis is accurate, introducing no pessimism, since the simple processor used can be modelled precisely.
2. *Timing and Space Analysis of compiled hardware components.*
   The low-level part of the hardware Ada compilation process utilises Xilinx place and route tools [14]. These are able to report the maximum clock speed for a particular circuit (ie. a particular compiled Ada task), together with the CLBs required for implementation.
3. *Timing Analysis of STRESS program.*
   This analysis performs a check on the feasibility of the entire system, as modelled by the STRESS program.

The relationship between the three analyses is important. As stated previously, the offline timing analysis (3) tells whether the system is feasible given certain assumptions. These assumptions, such as the WCET of a software task, must be met by the implementation (ie. (1) above).

The space analysis of individual hardware components is augmented by the space required by any softcore processor elements required by software components, to give an overall estimation of the space requirements of the system. This cannot be totally accurate until all hardware components (including components compiled from the source Ada, softcore processors required for software execution, buses etc.) are placed and routed together.

3.7 Allocator

The Allocator aims to minimise space usage by the system whilst ensuring that worst-case timing behaviour is sufficient to meet timing requirements. The Allocator evaluates the information gathered from analysis and simulation of the current allocation and attempts to improve the allocation. This occurs by application of one or more heuristics, including:

- Move a task to execute on a processor if its space requirement when compiled as a hardware component is greater than the size of a processor.
- Combine the tasks executing on different processors if their timing characteristics are complementary (which will improve the utilisation of the processor).
- Move tasks that share a protected object onto the same processor (may reduce blocking).
- Change the WCET of a task in the STRESS model (if the WCET software implementation of the task is too high).
- Move tasks with relatively high computation times and short deadlines to circuit.
- Move tasks with short periods / deadlines to circuit.
- Move protected objects to hardware (as a speed up to reduce blocking).

One or more of the heuristics can be applied at each iteration of the method, depending upon the information gathered from analysis and simulation. If the resulting analysis and simulation indicate that the changes have benefited the system, they are kept. Note that if necessary changes can be disregarded (ie. a backtrack occurs) in order to explore a different part of the solution space.

The Allocator is structured as a simulated annealing process. Given information from the analysis and simulation of the current allocation, one of more of the heuristics are declared valid for use. The simulated annealing process is then permitted to vary the allocation only by applying those heuristics. After the process has "cooled", the allocation found is returned. If the current allocation cannot be improved by the heuristics, a completely different (randomly generated) allocation can be generated and returned. At all stages, the current "best" solution is recorded.

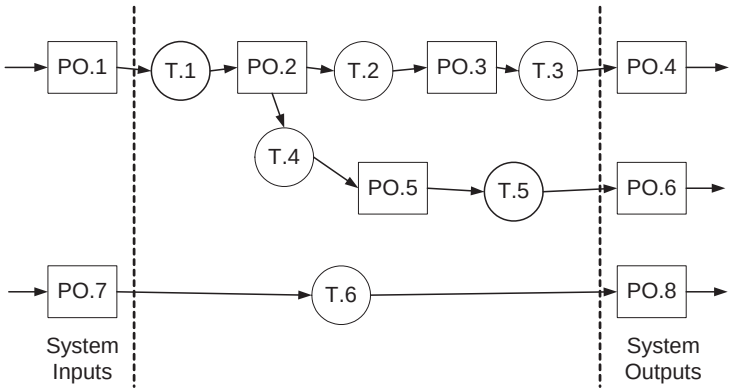
The Allocator has a number assumptions set prior to the start of the exploration. These include the size of the target device and the size of a softcore processor when implemented on the device. Currently, the allocator assumes that there is but a single available softcore (68020) and targets a fixed FPGA device (Xilinx Spartan IIe 200 FPGA).

Finally, a number of thresholds are set to determine how the heuristics perform. For example, the final heuristic above has a threshold to state when it is beneficial to move a protected object to hardware, depending upon the blocking experienced by the tasks that use that object.

4 Evaluation

The effectiveness of the approach is shown by an illustrative example. A schematic of some legacy code is illustrated in Figure 9. Here, tasks (in circles) communicate with each other via protected objects (squares), with the direction of data-flow indicated via arrows. Thus, protected object PO.1 provides routines for task T.1 inputs from the environment. T.1 then writes its output into PO.2, which is read in turn by T.2. The system is to be re-hosted on a single Xilinx Spartan IIe 200 FPGA. The available softcore is a 68020 core (as discussed in section 2.5), which occupies 18% of the FPGA.

Fig. 9. Example: Control-Flow of Task Interactions



The desired timing behaviour for the system is given in Table 1. This represents the required behaviour whether implemented in software or hardware. Timings for are given in ms. Software WCET and blocking times are for a software implementation of the task, that is running on the softcore processor. If all tasks shared a single processor, then by deadline monotonic timing analysis [13] all tasks will meet their deadlines (assuming that priorities are assigned as given in Figure 1 where 1 is the highest priority and 6 the lowest).

The initial part of the method is to assume each software task is allocated its own processor. The softcore processor assumed takes 18% of the FPGA. Thus, for 6 tasks, each allocated its own processor, the FPGA would be over utilised, at 108%. It is clear that some tasks need to be moved to share a processor. The space freed may then be utilised by moving some tasks to a hardware implementation.

After initial application of the method, 3 processors are utilised. One for tasks T.1, T.2 and T.3; one for tasks T.4 and T.5; one for task T.6. The main reason for this configuration is that the timing characteristics of the groups of tasks are complementary. As a result, the FPGA utilisation is reduced to 54%.

Table 1. Example Task Set

Task	Deadline	Period	Software WCET	Software Block	Software Priority	Hardware Size(%)
T.1	10	80	10	3	1	18
T.2	20	100	20	2	2	21
T.3	30	120	30	2	3	32
T.4	50	300	50	5	4	25
T.5	50	400	50	6	5	22
T.6	100	1000	200	0	6	34

However, this leaves a substantial amount of the FPGA that can be utilised by hardware implementations of critical tasks.

The size of each task, if implemented purely in hardware is given in Figure 1, expressed as a percentage of the whole FPGA (note that it is impossible to have all tasks in hardware as their total size is 152% of the FPGA). Movement of tasks to hardware is biased towards short deadline tasks, and tasks with high utilisations (ie. software WCET divided by period). Thus, the method identifies task T.3, as it has the highest utilisation. Weighting the allocator differently may lead to a different tasks in hardware.

It is noted that this approach enables the mapping of task sets that would not be schedulable on a single processor, eg. containing tasks with a high individual utilisation. For normal CPU-based systems, this would require additional processors to be present and relatively complex task allocation schemes to be utilised to split the task set over the CPUs. With the approach outlined in this paper, it is entirely probable that such high utilisation tasks will be moved to execute in hardware directly, or in their own softcore, without the expense of an additional CPU. Hence, the approach outlined in this paper is far more flexible.

5 Previous Work

The field of hardware / software codesign has provided a wide range of approaches and techniques [20]. Most techniques, (eg. COSYMA [21], SpecSyn [22], POLIS [23]) start from a high level specification of the system which is then tranformed into hardware and software. Other approaches start from a lower-level, ie. a software like language, which is then transformed into hardware and software. However, these approaches map to a fixed target architecture (eg. processor plus some hardware functions in ASIC), rather than the flexible targets afforded by FPGAs. None of the approaches consider concurrent languages (eg. Ada) as the source for the codesign process.

6 Conclusions

This paper has described a codesign based method for the re-hosting of legacy Ada systems onto FPGA. Given that the Ada systems under consideration are

implementations of hard real-time systems, it was imperative that the system timing requirements were shown to be met offline. This motivated a static timing analysis driven approach to finding a mapping between the source Ada software and an FPGA based platform. The method outlined ensures that if a solution is found, then the timing requirements will be met. Also, the method attempts to improve the efficiency of the solution (in terms of resources required (ie. size of FPGA)), whilst meeting timing requirements.

Outstanding issues that warrant further research include the granularity of software and hardware unit; extension of the target platform to multi-FPGA; inclusion of wider system parameters, such as power consumption of the system.

References

1. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages. 3rd edn. Addison Wesley (2001)
2. Audsley, N., Burns, A., Richardson, M., Wellings, A.: Hard Real-Time Scheduling: The Deadline Monotonic Approach. In: IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA, USA (1991) 133–137
3. Healy, C.A., Whalley, D.B., Harmon, M.G.: Integrating the Timing Analysis of Pipelining and Instruction Caching. In: Proceedings of the 16th Symposium on Real-Time Systems, IEEE (1995) 288–297
4. Lim, S.S., Han, J.H., Kim, J., Min, S.L.: A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In: Proceedings of the 19th IEEE Symposium on Real-Time Systems. (1998) 334–345
5. Ward, M., Audsley, N.C.: Hardware Compilation of Sequential Ada. In: Proceedings of CASES 2001. (2001) 99–107
6. Ward, M., Audsley, N.C.: Language Issues of Compiling Ada to Hardware. In: Proceedings of Ada Europe 2002. (2002)
7. Taft, S., Duff, R., eds.: Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1996(E). Volume Lecture Notes in Computer Science 1246. Springer-Verlag (1997)
8. Barnes, J.: High Integrity Ada: The SPARK Approach. Addison-Wesley (1997)
9. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In: Reliable Software Technologies, Proceedings of the Ada Europe Conference, Uppsala. Volume 1411., LNCS, Springer-Verlag (1998) 263–275
10. Ada Core Technologies: GNAT Ada Compiler : http://www.gnat.com. (2001)
11. Ward, M., Audsley, N.C.: Hardware Implementation of the Ravenscar Ada Tasking Profile. In: Proceeedings of CASES 2002. (2002) 59–68
12. Sha, L., Rajkumar, R., Lehoczky, J.: Priority Inheritance Protocols: An Approach to Real-Time Synchronisation. IEEE Transactions on Computers **39** (1986) 1175–1185
13. Audsley, N.C., Burns, A., Richardson, M.F., Tindell, K., Wellings, A.J.: Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. Software Engineering Journal **8** (September 1993) 284–292
14. Xilinx Corporation: Xilinx Product Information : http://www.xilinx.com/products. (2004)
15. Altera Inc.: Altera Product Information : http://www.altera.com. (2004)

16. Celoxica Ltd: Celoxica Product Information : http://www.celoxica.com. (2003)
17. Open Cores: Open Source IP Cores : http://www.opencores.org. (2004)
18. Audsley, N.C., Burns, A., Richardson, M.F., Wellings, A.J.: STRESS: A Simulator For Hard Real-Time Systems. Software, Practice and Experience **24** (June 1994) 543–564
19. Ellsberger, J., Hogrefe, D., Sarma, A.: SDL: Formal Object-Oriented Language for Communicating Systems. Prentice-Hall (1997)
20. DeMicheli, G.: Readings in Hardware / Software Codesign. Morgan-Kaufman (2001)
21. Henkel, J., Ernst, R.: A Hardware/Software Paritioner Using a Dynamically Determined Granularity. In: Proc. Design Automation Conference. (1997) 691–696
22. D. Gajski, F. Vahid, S.N., Chong, J.: System-Level Exploration with SpecSyn. In: Proc. Design Automation Conference. (1998) 812–817
23. Suzuki, K., Sangiovanni-Vincentelli, A.: Efficeint Software Performance Esimation Methods for Hardware / Software Codesign. In: Proc. Design Automation Conference. (1996)