Nios Code Generation Specification

Jonah Caplan

1   Introduction

The aim of this project is to develop an infrastructure for the automatic generation of C code for multicore Nios systems. We assume that an external model based design approach is used to generate control algorithms and that C code for these computations have already been generated in separate files (e.g. Simulink). The purpose of this tool is to efficiently map the control system onto an arbitrary platform while taking into account non-functional requirements such as deadlines, data flow, and criticality. The user must only specify the requirements for the system at a high level of abstraction and all intermdeiate code will be automatically generated.

This tool will not initially be geared towards solving codesign problems. We will assume a static hardware platform and ocnsider changing software requirements only. In order to facilitate later expansions, the tool will require the specification of the hardware in terms of generic model parameters. The task-mapping procedure will be platform independent and solve the problem generically even if we do not currently take advantage of this feature.

This document will provide the specification for the currently supported hardware models, the platform built from these components currently under study, the application models for analysis of software requirements, the mapping procedure combining both hardware and software models to generate a schedule, and the abstract template requirements for code generation.

2   Tool Structure

This key to success for this project in the given time frame is to leverage open source, take advantage of the wealth of prior work in the field from the 1990s in synchronous language design, and more current work on automated DSE and code generation. Furthermore, the wealth of generated code that describes the platform, as well as a very well structured RTOS uC/OS-II integrated into the BSP make the job of template development and hardware platform specification much easier. While the code can be a bit cumbersome to navigate for a human user because it is generated, this actually makes it much easier to parse for an automated process.

Figure 1 depicts the tool architecture. From this figure we can begin to derive a timeline for development by identifying and prioritizing milestones.

1. Define model classes for application and processors.

2. Integrate models with GA library to build task mapper.

3. Test mapper extensively with randomly generated task graphs, including timing data and fault-tolerance requirements.

4. Write RTOS templates.

    (a) Deconstruct existing code into discrete units with well-defined functionality.

    (b) Formalize inter-core communication protocols.

    (c) Write new templates for non-existing features (e.g. killing erroneous tasks, setting MPU parameters).

    (d) Rewrite all drivers for fingerprint unit, comparator and TLB.

5. Build code generation tool.

6. Write WCET estimation tool.

7. Define QSYS naming conventions and write system.h BSP parser.

8. Write language specification for high level model input.

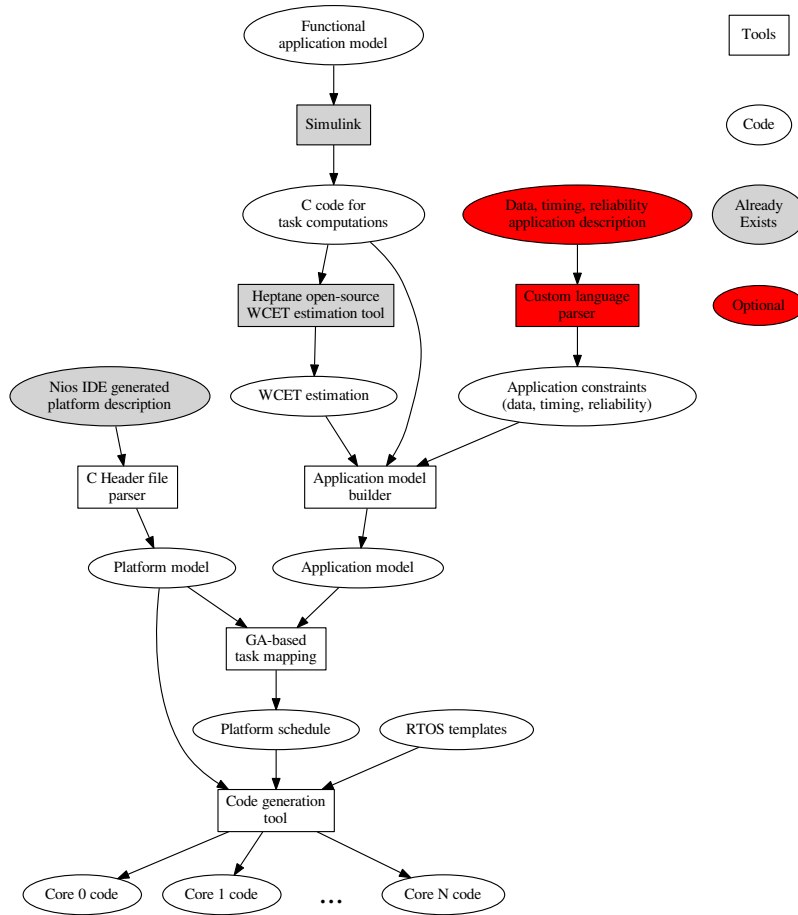9. Write scanner/parser using SableCC3 to build abstract Java application model from user input.

Figure 1: The proposed tool architecture.

3   Hardware Model

Hardware models are specified using an object-oriented semantics. The system is divided into hierarchical levels that are interpreted using static scoping rules to aid in the specification of larger

models.

The cost in tiucme of transmitting over communication channel models is omitted at this stage beyond specifying the connections between elements. Issues related to resource arbitration and interference are also not considered. The underlying infrastructure is assumed to be sufficiently quick and deterministic.

There are four categories of hardware models in the system: processor cores, peripheral cores, and memory and buses.

3.1   Processor cores

All processor cores are assumed to operate at the same clock frequency. Their parameters are determined during hardware design and are not altered during software mapping. They can be extracted from the *system.h* file generated from the .sopcinfo file by the Nios IDE during BSP generation.

Processor parameters are:

1. *Fault-tolerant*: While we do not have access to safety-critical Nios licenses, they do exist. We assume that a core can be designated as fault tolerant (FT) and that a cost is associated with fault tolerance (due to licensing, size, resource utilization, power consumption as appropriate for the scenario) and that it is therefore necessary to have fewer FT cores in the system.

2. *Scratchpad*: The processor must have a scratchpad in order to use fingerprinting as an error detection mechanism under our current implementation. The relevant parameters for the scratchpad will be listed separately.

3. *Timer*: The system timer will dictate the minimum period for events in the system. The timer period is statically assigned when specifying the hardware design in QSYS.

4. *Fingerprint Unit*: Fingerprint units will be necessary to allow the monitoring of processing cores (PC), i.e. those cores lacking FT capabilities, by a FT core.

5. *Data and instruction cache*: It may be necessary to disable data and/or instruction caches while executing critical tasks. It must be known if the processor is equipped with either.

6. *DMA*: A single channel DMA will be used to shuttle critical data in and out of the scratchpads.

7. *MPU*: An MPU will be required to ensure that each core is unable to maintain partitions between each core.

8. *Shared memory*: Shared memory space will be required to load instructions

9. *Interrupt signals*: The processor model must specify the actively connected interrupt signals.

This is a very general model of the processor. The mapping of tasks to cores will take place considering a very high level model of the system. The lower level parameters will only be used for code generation purposes. For the processor model, it is sufficient to consider whether or not each of these components are available. The details of each component will be hidden from the mapping problem at this higher level of abstraction.

3.2   Memory

Local scratchpads will be required for each core as well as shared main memory. A memory is defined simply as a start and end address. Memory latencies are not modelled. A memory model will also keep track of what other modules are connected to it.

There will be two partitioned sections of shared main memory. One to access common functions for redundant task executions and another for message passing between cores.

3.3   Peripherals

Certain details about the peripherals such as control registers will be entirely encapsulated in the template objects for the code generation phase. The higher level concerns that will dictate how these registers are sit will be included in the object representation of each peripheral. Mappings from the higher level concerns to code generation rules will be specified.

3.3.1   Timer

The frequency of the system level timer must be known in order to define the minimum resolution of time in the application model. Preexisting drivers exist and macros are generated by the Nios IDE to manage the timer. All relevant macros can be extracted from the generated code.

3.3.2   Fingerprint Unit

The fingerprint unit has the following features: maximum stack depth, statically set during hardware design, and block length size. The setting of

3.3.3   Memory Protection Unit

3.3.4   Comparator

3.3.5   DMA

3.3.6 μTLB

4   Application Model

4.1   Functional decomposition

An extension to MPSoCs is proposed to the application model and response-time analysis presented by Baruah, Burns and Davis in [1]. Tasks are defined in the uniprocessor case by the tuple $\tau_i = < T_i, D_i, \overrightarrow{C_i}, L_i >$. Tasks are assumed to be independent and periodic. It may be beneficial in a multicore system to consider core allocation at a finer granularity. Tasks may be composed of several functions with dataflow dependencies between them. It may be desirable to allocate several cores to a single task in order to allow the parallel execution of these functions. We will only consider a two core case as an example to simplify the discussion.

The problem of deciding whether or not to execute a task on a specific core, or on both cores, and how to assign functions within a task to a given core will probably need to be explored using standard DSE techniques such as a genetic algorithm as the size of the space will grow very quickly

Function	WCET
F0	20
F1	10
F2	10
F3	10
F4	20
F5	30
F6	10
Total	110

Table 1: WCET derivation for the task in Figure 2

(*how quickly?*). **The question is whether or not allowing the division of a task into subcomponents and to allow the acquisition of more than one resource at a time by a single task allows more systems with higher utilization to pass schedulability tests than standard techniques.**

Consider the complex task graph in Figure 2. Each function has some WCET associated with it. Suppose the WCET of the task is 110ms and is decomposed as in Table 1. It is possible to execute this task on two resources with the mapping shown in Figure 3. A difference of 30ms is achieved. Communication overhead could be modeled by including a bus as a processing element and to insert tasks into the graph to model the communication overhead. For now the inter-core communication overhead is assumed to be negligible.
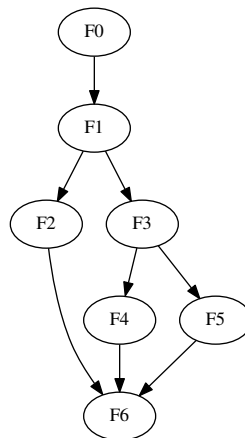

Figure 2: Example of task decomposition into functions with dataflow precedence relations.

Time	10	20	30	40	50	60	70	80	90	100	110

CPU1   [F0][F1][F2][F4]   [F6]
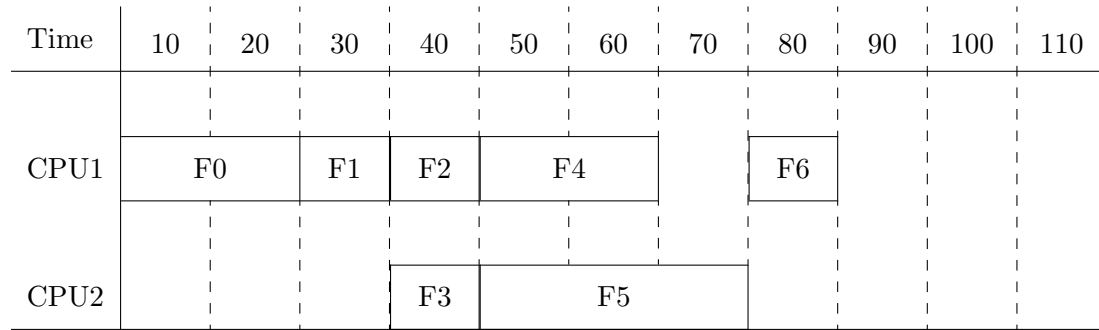
CPU2   [F3][F5   ]

Figure 3: Possible mapping for distributed task onto two cores.

Now consider a set of tasks $\tau_i \in T$ each of which can be optionally decomposed and distributed in such a fashion. The decision of which tasks to decompose will best be left to a DSE approach. The analysis that follows assumes that a strategy has already been chosen. The following discussion will apply SMC as a first step to demonstrate how the RTA can be accomplished using standard analysis. The mapping in Figure 3 is reexpressed as a transformed CFG in Figure 4. The communication overhead is now explicitly modeled (but can be given an execution time C=0). The program is divided into serial and parallel chunks. For example $S_0 = \{F_0, F_1\}$ and the first parallel branch $P_{0.0} = \{F_2, F_4\}$. *How to define and partition parallel and serial sections, especially when the number of cores increases?*
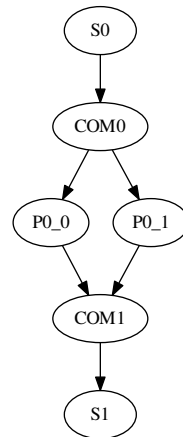

Figure 4: Transformed task graph based on mapping in Figure 3. The serial portions are denoted by S, the parellel portions by P, and communication overhead by COM.

Priority assignment will be done first on a task by task basis, and then on a function by function basis. The task priority assignment will first be done according to standard SMC practices by assigning the lowest priority to the task with the latest deadline. If a task is running on two cores then it must be the lowest priority on *both cores* before being selected. There may of course still be both a low and high criticality candidate. With priorities assigned for each task, then parallel tasks must be further decomposed into several subtasks. The priority of each subtask must be assigned

such that:

- Any sub-task belonging to the same task that executes on the same core and that has precedence must have a higher priority.

- The set of tasks which have higher priority and lower priority must be the same for all subtasks as for the parent task (excluding the subtasks themselves).

- Each subtask has the same L_i T_i and D_i as the parent task.

Let the set of all subtasks belonging to the same task τ_i be denoted $\bar{\tau}_i$. The response time R_i is decomposed into serial and parallel sections. The response time for a serial section is simply:

$$R_{i(s)} = C_{i(s)} + \sum_{\tau_j \in hp(i) \setminus \bar{\tau}_i} \left\lceil \frac{R_{i(s)}}{T_j} \right\rceil C_j(L_i) \tag{1}$$

Note that the set of higher priority tasks considered does not include sub-tasks belonging to the same priority. For parallel branches that begin at the same time, the maximum of all the branches must be taken:

$$R_{i(p)} = \max_k \left\{ C_{i(pk)} + \sum_{\tau_m \in hp(i) \setminus \bar{\tau}_i} \left\lceil \frac{R_{i(pk)}}{T_m} \right\rceil C_m(L_i) \right\} \tag{2}$$

The response time is the sum of all the parallel and serial sections of the task:

$$R_i = \sum (R_{i(s)} + R_{i(p)}) \tag{3}$$

With this definition, only the parts of a divided task that are on the same core will have an impact on the response time. The effect of having two serial functions that belong to a higher priority task execute on a core is the same as having two independent tasks with the same period T_i and combined execution time C_i.

With a schedulability analysis in place, it is possible to do DSE to find a legal mapping. **It remains to be determined whether this extra degree of freedom results in more schedulable systems at higher utilizations, and how this relationship is affected by the shape of individual CFGs, the amount of branching, and the communication overhead.**

Typical multicore scheduling algorithms are categorized as either partitoned or global. Global scheduling assigns tasks to cores at runtime from a global queue. Partitioned algorithms statically allocate tasks to cores at design time while still allowing for the sporadic generation of new jobs by each task. This application model and scheduling algorithm is closer to partitioned scheduling. The resources consumed by a task are statically assigned however a task is no longer restricted to consuming a single resource. Tasks will consequently take advantage of coarse-grained parallelism in its dataflow to provide different performance or energy tradeoffs. Further tradeoffs may also be discovered when integrating reliability concerns into the mixed-criticality discussion.

Table 2: Example task set

	C(LO)	C(HI)	T=D	L	P
τ_1	5	10	25	HI	3
τ_2	5	-	20	LO	4
τ_3	2	-	8	LO	1
τ_4	1	-	5	LO	2

4.2   Mixed-criticality fault-tolerant schedulability analysis (MCFTS)

The basic language for MC multicore schedulability can be used to extend mixed-criticality scheduling to include fault tolerance. First, we define the **task model**:

$$\tau_i = < T_i, L_i, C_i(L), D_i, pFH(L_i) > \tag{4}$$

where:

- T_i: frequency of task

- L_i: criticality of task

- C_i: mode-dependent execution time of task

- D_i: deadline of task

- pFH: criticality-dependent maximum allowable probability of failure/hour

Second, we define the **processor model**:

$$\pi_i = < E_i, v_i, f_i, d_i, c_i, r_i, hw_i > \tag{5}$$

where:

- E_i: energy profile

- v_i: operating frequency

- f_i: probability of transient fault

- d_i: availability of internal detection mechanism

- c_i: availability of internal correction mechanism

- r_i: recovery overhead

- hw_i: hardware profile (e.g. has FPU, has fingerprint unit, etc.)

Consider the example task set provided in Table 2. We wish to determine whether it is possible to execute the task set on a two-core platform. The safety of the system will depend on the required level for each criticality level as well as the predicted rate of transient errors per core. We assume that some dependable runtime mechanism exists for keeping track of the actual number of transient faults per hours of operation. It is also assumed that each node is capable of detecting errors. In this case two values are defined: $f(LO)$ is the predicted rate of soft errors and $f(HI)$ is a more conservative estimate to be used in the case that a higher error rate than $f(LO)$ is detected in the field. It is possible that the ratio $\frac{f(HI)}{f(LO)}$ is provided by a safety standard.

Each HI level task also has two execution times $C(LO)$ and $C(HI)$. Four criticality modes naturally emerge from the fact that both f and C have two possible values. The four modes are:

1. $LO_c LO_f$: Default mode. All tasks complete within maximum number of retries and within $C_i(LO)$.

2. $LO_c HI_f$: The predicted f_i on processor π_i is surpassed. More retries must be scheduled for HI tasks.

3. $HI_c LO_f$: The execution time of a task exceeds its $C_i(LO)$. In this case more time must be allotted for the execution of each task and its retries.

4. $HI_c HI_f$: Both the measure execution time and frequency of transient faults exceed their LO values.

Suppose that $f(LO) = 10^{-5}$ and $f(HI) = 10^{-3}$. Furthermore, suppose that $pFH(LO)$ is given to be 10^{-3} and $pFH(HI) = 10^{-8}$. It should be immediately clear that there is a problem since $pFH(HI) > f(LO)$ as well as $f(HI)$. The solution to this problem is to re-execute the the HI level task τ_1. If a task τ_i has utilization $u_i = \frac{C_i}{T_i}$ and is running on a processor with f_i then $pFH_i = u_i f_i$. Assuming that each run is independent, then executing a task n times results in an overall probability of failure:

$$pFH_i(n) = (u_i f_i)^n \tag{6}$$

In this case, the minimum number of re-executions can be determined:

$$n \geq \frac{log(pFH_{L_i})}{log(u_i f_i)} \tag{7}$$

It is now possible to determine the minimum number of executions of τ_1 for each mode:

$$(U_1(LO)f_1(LO))^n = (0.2 * 10^{-5})^n \leq 10^{-8} \rightarrow n \geq 1.40 \tag{8}$$

$$(U_1(HI)f_1(LO))^n = (0.4 * 10^{-5})^n \leq 10^{-8} \rightarrow n \geq 1.48 \tag{9}$$

$$(U_1(LO)f_1(HI))^n = (0.2 * 10^{-3})^n \leq 10^{-8} \rightarrow n \geq 2.1 \tag{10}$$

$$(U_1(HI)f_1(HI))^n = (0.4 * 10^{-3})^n \leq 10^{-8} \rightarrow n \geq 2.35 \tag{11}$$

The result is that only one replica is needed so long as the $f(LO)$ threshold is not exceeded. Otherwise, two replicas are needed. The task set is updated in Tables 3 and 4.

Once the task sets have been updated, standard schedulability analysis can be carried out with AMC-rtb and partitioned core allocation. The replicas are modelled as a parallel set stemming from a source and sink with zero cost.

Table 3: Updated task set for modes LO_cLO_f and HI_cLO_f

	C(LO)	C(HI)	T=D	L	P
τ_1	5	10	25	HI	3
τ_1'	5	10	25	HI	4
τ_2	5	-	20	LO	5
τ_3	2	-	8	LO	1
τ_4	1	-	5	LO	2

Table 4: Updated task set for modes LO_cHI_f and HI_cHI_f

	C(LO)	C(HI)	T=D	L	P
τ_1	5	10	25	HI	3
τ_1'	5	10	25	HI	4
τ_1''	5	10	25	HI	5
τ_2	5	-	20	LO	6
τ_3	2	-	8	LO	1
τ_4	1	-	5	LO	2
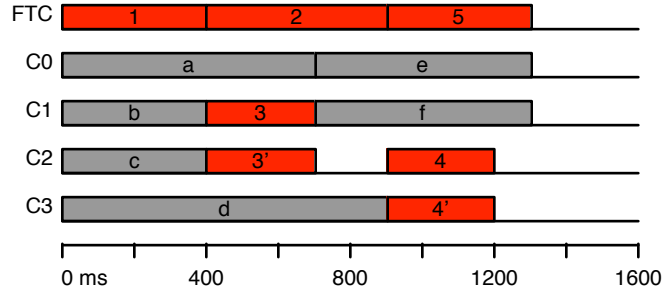
Figure 5: System scheduling using ODR and DCC. Non critical tasks are depicted in grey and critical tasks are depicted in red.

4.3 MCFTS and on-demand redundancy

The processor model allows for the existence of internal detection or correction mechanisms. These mechanisms may be expensive and difficult to scale and may not be available on every core in a system. On-demand redundancy (ODR) [?] suggests it is possible to run replicas on two unprotected cores in order to detect an error by comparing their outputs. The cores may otherwise execute independently when working on non-critical tasks. There are proposed mechanisms [?] to allow the dynamic pairing of cores over long distances as well as studies examining the benefits to overall performance in a non-embedded setting [?].

On demand redundancy is illustrated in Figure 5. The system contains one fault-tolerant core (FTC) which is assumed to have dependable internal detection or correction mechanisms. The other cores are assumed to be some combination of smaller, more powerful, and more energy efficient because they are unburdened with extra fault-tolerance logic. Cores C1 and C2 execute non-critical tasks independently. They are then coupled to execute Task 3 and its replica in parallel. This is equivalent to a single replica with error detection assumed to take place within the node in the previous section.

The probability of a task failing is slightly different now that it depends on the execution of two nodes, each with their own f_i. A task τ_i executing on two cores π_j and π_k will have a new failure rate:

$$f_{jk} = 1 - (1 - u_i f_j)(1 - u_i f_k) \tag{12}$$

That is 1 minus the probability that neither processor has a transient fault (assuming faults on each core are independent).

5 Code Profiling

5.1 Parsing

The profiling tool determines maximum stack size required by a function as well as its WCET to map the application to hardware resources and prove schedulability. The tool takes as input the disassembly file generated by `nios2-elf-objdump` as well as the root function to be analyzed. Listing 1 demonstrates how to correctly generate the disassembly and run the profiler. The user should eventually be able to run a series of profiles on various functions using a configuration file.
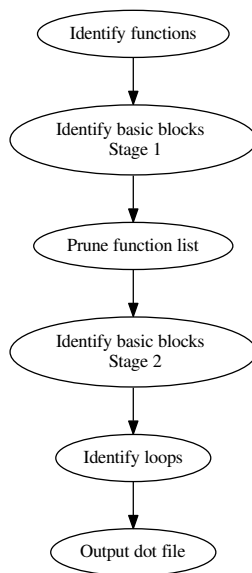
Figure 6: Parser workflow

Listing 1: Generating disassembly and running the profiler

```
$ nios2−elf−objdump −dt filename.elf > filename.objdump
$ java Profiler filename.objdump top_level_function
```

The parser works in several passes. First, function names are extracted from the `objdump` file. The next few passes build the control flow graph (CFG). The code associated for each function is broken down into basic blocks. Calls, returns, branches, and jumps are identified as the end of basic blocks. The parser can now identify which functions are called by other functions. All functions that are not called from the root are removed from the analysis at this point. Following this pruning of the CFG, the basic blocks are further divided because branch, jump and call *destinations* must be the start of basic blocks. Finally loops are identified by calculating the *maximum depth* of each basic block. A backwards edge occurs in the case that a successor has a lower depth. The parser finally ouputs a `dot` file that can be compiled into a `pdf`. Figure 6 depicts the parser workflow. Figure 7 contains an example of the `dot` file.

5.2   Stack Analysis

It is possible to start analysis once the parser has built the CFG. Stack analysis is quite straight-forward. Each basic block in a function is checked for instructions that increase the stack size. Note that stack instructions should not occur in a loop. If a basic block calls a function, then that function is also checked for stack instructions and then this result is added on to the original calculation. Recursive functions are not supported.
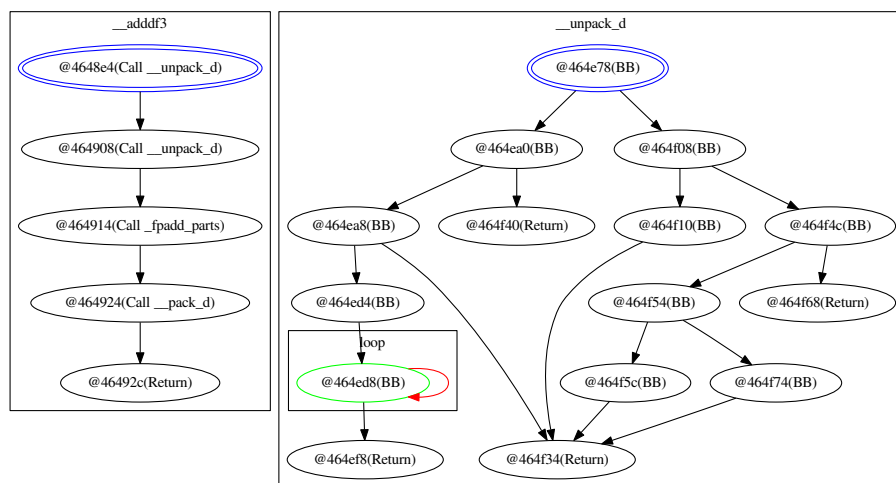
Figure 7: Example of visual CFG representation generated by the parser

5.3   Library functions

Library functions are important because they must be copied into the scratchpad along with the rest of the critical code. The library as well as the object that contains the function must be known in order to efficiently modify the linker script. All the library functions can be placed in a contiguous portion of memory for easy DMA transfer. The library directory for the default compiler used by the Nios II Software Build Tools (SBT) is located at:

    altera/13.1/nios2eds/bin/gnu/H-i686-pc-linux-gnu/nios2-elf/lib.

These libraries have been copied to code_gen/nioslib for convenience. The script extract.sh has been provided to extract object names and function names from each library. The main commands are explained in Listing 2. Static objects mapping functions to their containing object files and library are then created in Java, as demonstrated in Listing 4.

Listing 2: Extracting information from shared libraries

```
# list the object files in an archive
ar -t $archive

# extract the object files from the archive
ar -x library.a

#create an objdump for each file.o
nios2-elf-objdump -d file.o > file.objdump

#Extract the function names for each object file
grep "<.*>:" $f.objdump | sed -r 's/.*<(.*)>:/\1/' >> $OUTPUT
```

Listing 3: Extracting information from shared libraries

```
1
  static HashMap<String, String> libc = new HashMap<>();
3   static {
      libc.put("memset", "lib_a-memset.o");
5     libc.put("critical_factorization", "lib_a-memmem.o");
      etc...
7   }
```

The output from the library analysis is a code listing that can be added inside a region in the linker script to place those functions.

Listing 4: Placing library functions in `.critical` region

```
  /* Library functions are: __muldf3,__muldi3,__pack_d,__unpack_d,__mulsi3,__lshrdi3,
     __ashldi3 */
2 /* To place these functions in a section called critical in linker.x: */
     .critical :
4    {
         PROVIDE (_alt_partition_critical_start = ABSOLUTE(.));
6        *(.critical .critical.*)

8        /* INSERT THE FOLLOWING */

10       */libgcc:_mul_df.o
         */libgcc:_unpack_df.o
12       */libgcc:_pack_df.o
         */libgcc:_lshrdi3.o
14       */libgcc:_ashldi3.o
         */libgcc:_muldi3.o
16       */libgcc:lib2-mul.o

18       /* END OF INSERTED CODED */

20       . = ALIGN(4);

22       PROVIDE (_alt_partition_critical_end = ABSOLUTE(.));
     } > processor0_0_scratchpad
```

5.4   Worst case execution time estimation

The WCET for a function is calculated using implicit path enumeration technique (IPET) [2]. IPET is a method of pessimistically determining the WCET of a program without actually identifying the worst-case path. The first step is to convert the CFG into an integer linear program (ILP) and the second step is to determine the cost of each basic block using microarchitectural modelling and/or dataflow analysis.

The goal of the ILP is to maximize the objective function:

$$\sum_{i=1}^{N} c_i x_i \tag{13}$$

where:

- N: Number of basic blocks

- c_i: Execution time of block i

- x_i: frequency of block i

The CFG is transformed into a set of linear constraints by noting that the number of times a basic block is entered must equal the number of times it is exited. Each edge in the CFG is assigned a variable e_i. The entry edge into the root basic block has the constraint $e_0 = 1$. For all other edges, constraints are extracted based on the observation that for each basic block: $\sum e_{in} - \sum e_{out} = 0$. For example, in Figure 8: $e_0 + e_1 + e_2 - e_3 = 0$.
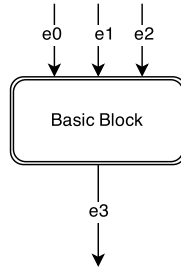
Figure 8: Sum of the edges into the basic block must equal the sum of the edges out: $\sum e_{in} - \sum e_{out} = 0$.

Loops require an additional constraint on the maximum number of iterations. Therefore for each loop $\sum e_{in} - \sum \text{maxIter} * e_{fl} \leq 0$, where maxIter is the maximum number of iterations for the loop and e_{fl} are the non-backwards edges into the loop (i.e. those that can only execute once per single round of loop iterations).
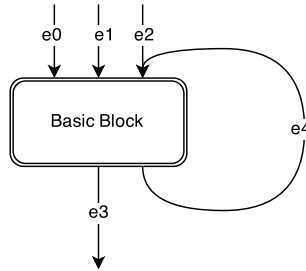

Figure 9: An additional constraint is required for loops: $\sum e_{in} \leq m$.

Function calls are handled quite simply. The entry-point to the top-level function simply equals 1. For all other functions, the entry-point equals the sum of all the edges leaving basic blocks that call that function. In Figure 10, the result is: $e_2 + e_3 - e_4 = 0$.

Library functions are difficult to analyze. Loops can be identified but their bounds cannot be easily determined analytically without detailed dataflow analysis beyond the current scope of the tool. An attempt was made to analyze the WCET of floating point operations by observing the number of times a loop executed in an OVP simulation environment. This information is then be
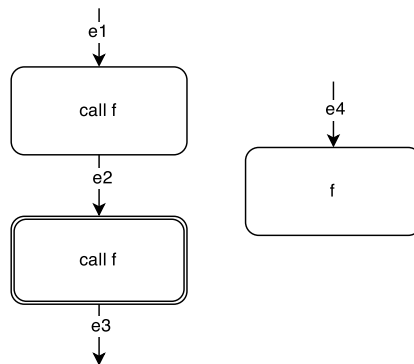
Figure 10: The sum edges leaving function call blocks is equal to the edge entering that function's root block.

used with associated microarchitectural modelling to come up with a cycle accurate estimation. There is no way to prove that the maximum observed loop iterations (over thousands of randomized trials) is in fact an upper boundary. Furthermore, it is a known problem that IPET yields pessimistic results if some paths through the control flow are impossible (but this is difficult to detect without detailed dataflow analysis).
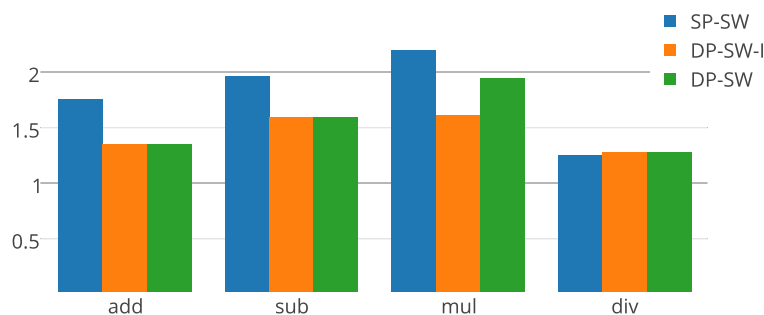

Figure 11: The results from IPET analysis compared to worst-case measured in OVP simulation.

Figure 11 shows the WCET calculated using IPET normalized to the measured execution time (maximum number of instructions observed in OVP) for all four operations. The OVP simulation provided boundaries for the library function loops. Single-precision is tested without integer multiplication hardware (SP-SW). Double precision is tested with integer multiplication hardware (DP-SW-I) and without (DP-SW). This chart demonstrates that software-based floating point operations are a source of imprecision that is difficult to overcome. For regular control flow, IPET will only be as effective as the hints provided by the programmer.

This result has motivated the inclusion of FPUs in the processing cores. The FPU provided by Altera executes single precision operations using the custom instruction interface to the Nios II. Each instruction has a known execution time in clock cycles which eliminates the pessimism in

calculating floating point operations. It is possible to force Simulink to generate code using only single precision variables and operations. As a result, there is a tradeoff between the accuracy of the WCET estimation, the size of the core (inclusion of an FPU), and limiting calculations to single-precision. The FPU will also remove thousands of instructions from the critical function and reduce the interference due to instruction loads from main memory as well as lower execution time considerably.

5.5 Microarchitectural modelling

6 Application Mapping

The initial scheduling algorithm will be taken from Bolchini and Miele [3]. A genetic algorithm (GA) is used to explore the design space. The general details that follow on genetic algorithms are based on the overview in [4].

Figure 12 shows an overview of how to implement a genetic algorithm. There are two stages of GA in the given process. First tasks must be organized into groups of tasks, and each group of tasks must be assigned a fault detection technique or fault tolerance technique. DMR Fingerprinting is an example of a detection mechanism. This work will remain agnostic about the method of implementation of fault tolerance. We assume that appropriate hardening measures are applied to a single core.
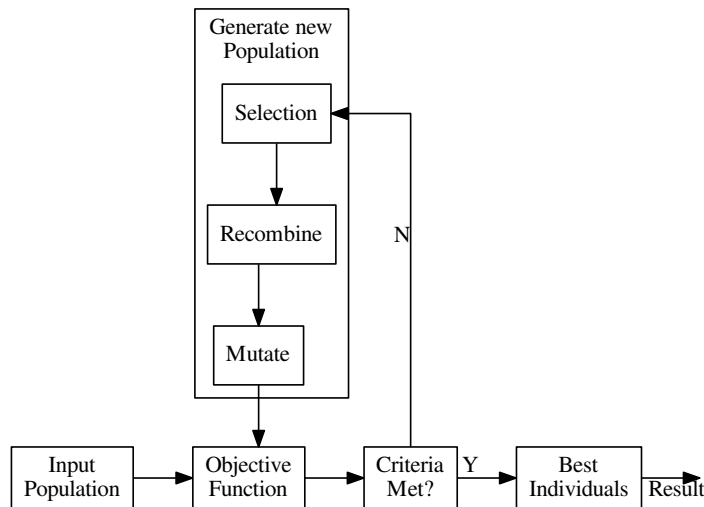
Figure 12: The basic structure of a genetic algorithm [4].

Once a population of various groupings and associated fault-handling mechanisms has been generated it is necessary to test their fitness. This must be done by generating a system schedule for each member of the population.

The generation of the schedule is itself done with a second round of GA. Each task is assigned

to an appropriate core based on the fault-handling requirements of its group. The group adopts the most stringent requirements of any of its members. Various mappings are generated as the GA population. Then it is necessary to order the tasks on each core. This is complicated by the fact that there may be inter-core data dependencies. For now, we will neglect the overhead associated with data transfer between cores.

The first step for scheduling on each core once groups have been assigned is to generate a list of tasks for which all the start conditions have been met (i.e. data dependencies on previously executed tasks). The highest priority task is chosen from this list for each core. This process cycles through all the cores and then proceeds to the next event in time (i.e. a task has ended) to repeat the process.

The priority of a task is determined using *modified partial critical path priority* [5]. The method consists of choosing tasks that minimize the critical path on *other cores*. It is possible that a successor to a task will be in the pool of tasks. Therefore, the critical path length is only considered from the first successor task that will execute on a different core.

Consider the example from [5] in Figure 13. From a start point P_0, either task P_A or P_B must be selected. The chain of tasks from P_A to P_X represents a series of successor tasks that will all be executed on the same core in time t_A. The time λ_A represents the time from the first successor on a different core to the terminal or sink task P_N. The scheduler must determine whether to assign P_A or P_B higher priority. This is done by choosing the option that minimizes the critical path *on other cores*. This means that t_A and t_B are neglected and the path with shorter λ is chosen. One interesting question is whether it would be simpler and more effective on a system with *many* cores to simply choose the task with the largest number of critical children nodes or non-critical children nodes (in that order) since this will most likely facilitate parallel execution on a large system.
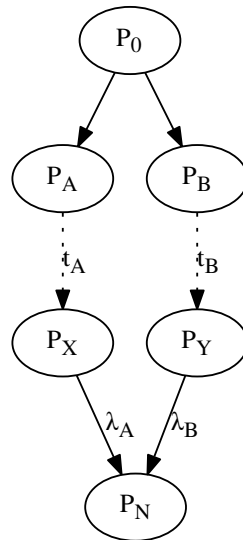

Figure 13: Example of modified partial critical path priority selection [5].

The implementation details of the genetic algorithm will be adopted from [3]. Tournament selection and single-point crossover operator recombination will be used. Mutations will be applied randomly. There are four mutations at the task level: change the group of a task, split a group in two, join two groups, or change the fault-management technique applied to a group. The mutation for the mapping phase consists of switching a task to another processor that can meet the assigned fault-management requirements.

Tournament selection means choosing a subset of the population and selecting the most fit member. This is repeated as many times as necessary. Single-point crossover cuts two parent chromosomes into two pieces and then exchanges the second substrings between the chromosomes.

The initial parameters for the GA will be the ones suggested in [6]. The crossover rate is 80%, the mutation rate is 10%. The program will run for 30 generations. The population size will be between one and two thousand.

The JGAP library [7] is used to implement the genetic algorithm.

6.1 Recovery

Fault model: can't fail twice. Need to leave room for reexecution. Fill up the retry slots with soft tasks. Just cancel them if the retry slot is needed.

Recovery only necessary in the case of correction. Detection implies no retry.

Why? Application where delivering the wrong answer can be catastrophic but dropping a frame is acceptable (i.e. not sensitive to jitter)

7 Code Generation

7.1 Case Study

Basic flow:

1. Functions are extracted from Simulink coder.

2. User manually inputs dataflow relationships for tasks.

3. User manually inputs timing and criticality requirements.

4. User chooses default platform.

5. User inputs task mappings.

6. Program generates RTOS code for each core, shared library code, and updates BSPs.

7. User may manually connect function outputs to IO.

   Extensions:

3a. User requests verification of timing requirements.

4a. User provides BSP, specifies names of relevant modules and requests that the tool determine the architecture.

4b. User provides a custom configuration file for the tool that specifies the architecture.

5a. User requests that the optimal mapping be determined automatically.

7a. User provides information about IO connections and requests automatic generation of full system.

7.2   Configuration File

The configuration file provided to the program must be able to express the basic flow as well as leave open the possibility of integrating extensions in the future. The syntax should be both easy to read for the user while also being easy to parse.

First, the functions must be specified. Each function must have a root name, a directory location as well as a constructor for global variables. The naming conventions for function names will follow the convention for Simulink generated code. The folder must contain the ert_main.c file that provides variable declarations and initialization code.

The configuration file will be specified through examples. First, the function declarations:

Listing 5: Function configuration syntax

```
1  <FUNCTION>
   #Comments start with #
3  # −c for critical task
   # −T for task period in ms
5  <LIST>
   −name AirbagModel            −dir /example_dir/AirbagModel_ert_rtw            −T  20 −c
7  −name CollisionAvoidance     −dir /example_dir/CollisionAvoidance_ert_rtw    −T 100
   −name CruiseControlSystem    −dir /example_dir/CruiseControlSystem_ert_rtw   −T  20 −c
9  −name Derivative             −dir /example_dir/Derivative_ert_rtw            −T  20 −c
   −name FuelSensor             −dir /example_dir/FuelSensor_ert_rtw            −T 100
11 −name TractionControl        −dir /example_dir/TractionControl_ert_rtw       −T  20 −c
   −name TransmissionControl    −dir /example_dir/TransmissionControl_ert_rtw   −T 100
13 </LIST>

15 #The dataflow is given for each pair of variables. One pair per line.
   #The names must correspond to the variables found in the function header file.
17 <DATAFLOW>
   #form is:
19 #modelname_Y.outputname modelname_U.inputname
   CruiseControlSystem_Y.Out1          Derivative_U.In1
21 CruiseControlSystem_Y.Out1          TractionControl_U.CurrentSpeedPoweredAxle
   TractionControl_Y.ControlFeedback   CruiseControlSystem_U.In2
23 Derivative_Y.Out1                   AirbagModel_U.Force
   </DATAFLOW>
25 </FUNCTION>


27
   <PLATFORM>
29 −usedefault
   </PLATFORM>
31
   #default platform has three cores cpu0, cpu1, and cpum
```

```
33  #critical tasks must be either mapped to cpum, or else mapped to BOTH cpu0 and cpu1
    #all critical tasks will be verified using fingerprinting
35  <MAPPING>
    AirbagModel cpu0 cpu1
37  CollisionAvoidance cpu0
    CruiseControlSystem cpum
39  Derivative cpu0 cpu1
    FuelSensor cpu1
41  TractionControl cpum
    TransmissionControl cpu0
43  </MAPPING>
```

Whitespace is ignored except for line breaks. Each top level function must have a unique name and only be instantiated once.

7.3   Code templates

This tool is designed to convert code generated by Matlab into a working and estimate safe properties for a mutlicore Nios system. The baseline architecture is depicted in Figure 14.
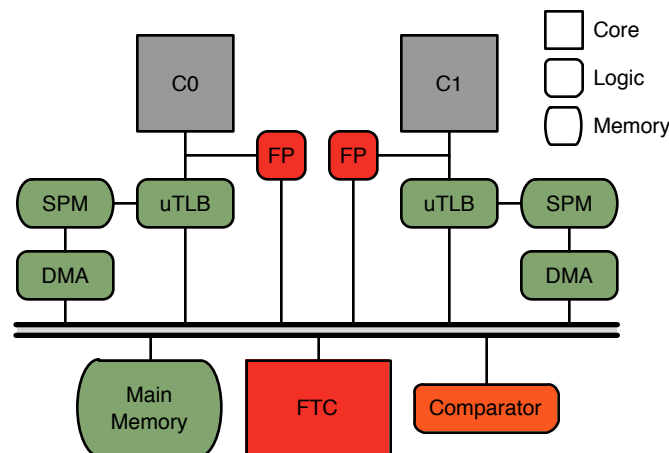

Figure 14: The baseline architecture for the generation tool

A fault-tolerant core (FTC) (assumed to have internal detection and/or correction mechanisms to manage soft errors) monitors execution on less reliable cores. Code must be generated for message passing, thread migration, bus management, memory management, runtime monitoring and reliability management. A breakdown of the generation templates will follow using the example Simulink generated control system, prepared as part of an undergraduate design project, in Figure 15.

The control sytem is broken up into seven tasks:

- Cruise control and adder
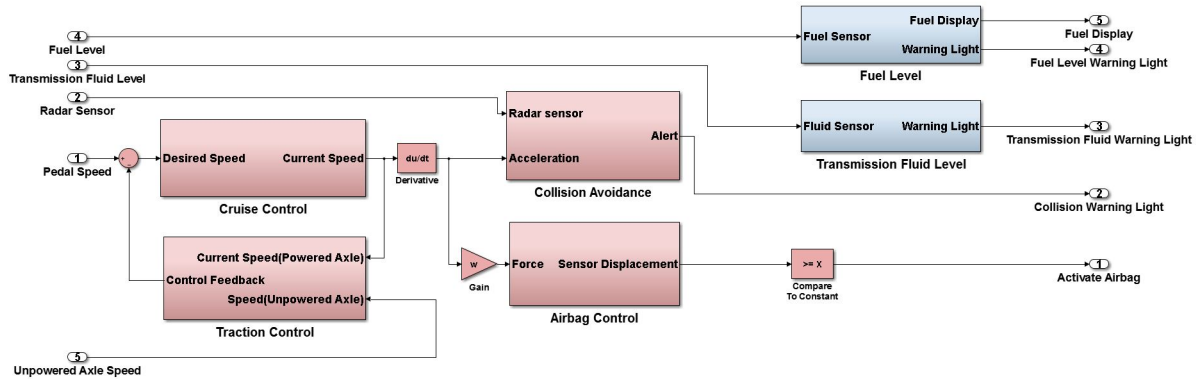
- Traction control

- Derivative

Figure 15: Example control system from Simulink

- Gain, airbag control and compare

- Collision avoidance display

- Transmission fluid level display

- Fuel level display

The three display related functions are considered non-critical. The other functions are critical as they relate to airbag activation. For this example all critical tasks must execute every 20ms, while non-critical tasks execute every 100ms. As a first step, the BSP, shared libraries, and code for these tasks must be generated given a predetermined core mapping (either by a provided tool or using external methods).

7.3.1 Monitor tasks

The monitor tasks must run on the FTC. They consist of:

- setting up task parameters and initializing all critical task models

- managing dataflow between tasks

- intercore task communication

- retrieving valid data when critical tasks execute correctly on other resources

- restarting cores in the case of a transient fault

- managing the global data space

- organizing coordinated virtual memory management and memory protection between cores to achieve correct fingerprinting

Figure 16 shows a high-level view of how the interaction between monitor tasks in order to send a message.
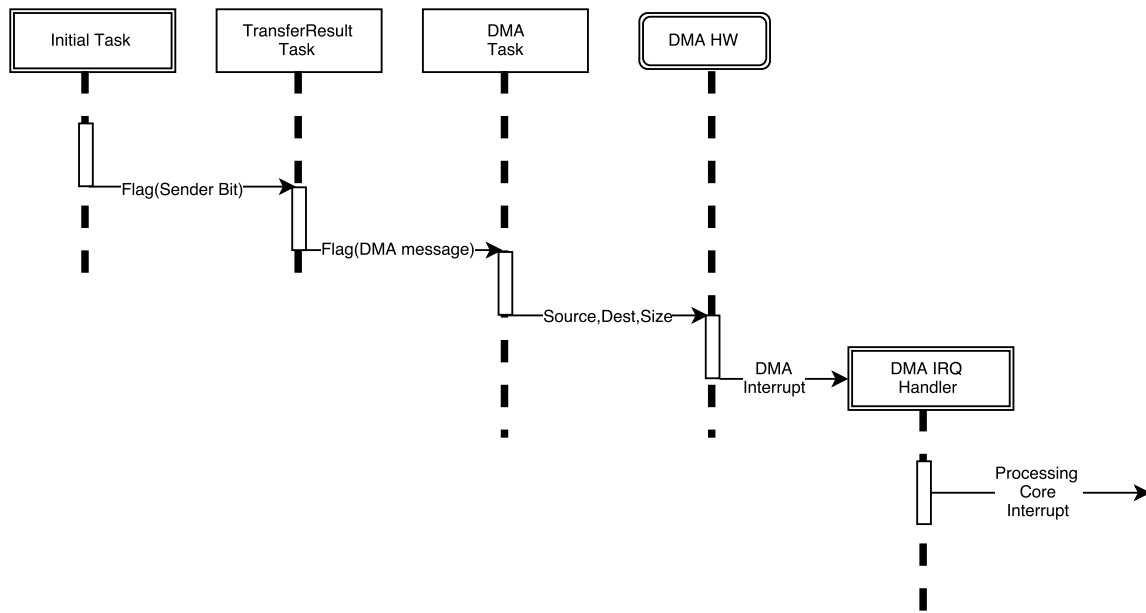
Figure 16: The interaction between monitor tasks when setting up a critical task to run on another core

MODULARITY VS. CONTEXT SWITCH OVERHEAD → FUNCTIONAL VS. TASK-BASED DIVISION OF LABOUR???

The monitor is responsible for ensuring that

7.3.2 Integration of Simulink generated code

The code generated from the Simulink model will dictate how to structure the communication between tasks. Functions must be generated using the reusable code option and using the input and output structures option. Discrete models must be used for Nios compatibility.

JC: show simulink options menu figure here

There are three functions generated for each group of blocks: an initalization, a termination, and a step function. Example function declarations are shown in Listing 6 along with the struct definitions for the input parameters. There are three different parameters for all the function: `RT_MODEL_AirbagModel_T` stores the state that is needed between time steps, `ExtU_AirbagModel_T` stores the external input parameters, and `ExtY_AirbagModel_T` stores the external outputs. The members of each input and output struct must either connect to inputs and outputs of other functions or communicate somehow with on-chip peripherals or through IO. The connection to IO is not currently implemented.

Listing 6: Airbag model function declarations

```
/* Model entry point functions */
extern void AirbagModel_initialize(RT_MODEL_AirbagModel_T *const AirbagModel_M,
  ExtU_AirbagModel_T *AirbagModel_U, ExtY_AirbagModel_T *AirbagModel_Y);
extern void AirbagModel_step(RT_MODEL_AirbagModel_T *const AirbagModel_M,
  ExtU_AirbagModel_T *AirbagModel_U, ExtY_AirbagModel_T *AirbagModel_Y);
```

```
extern void AirbagModel_terminate(RT_MODEL_AirbagModel_T *const AirbagModel_M);
```

In Figure 15 the cruise control outputs a single real_T value (Listing 7) and the derivative function has an identical input (Listing 8). The monitor will need to pass the signals between the data structures.

Listing 7: Cruise control output struct definition

```
/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real_T Out1;                            /* '<Root>/Out1' */
} ExtY_CruiseControlSystem_T;
```

Listing 8: Derivative input struct definition

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T In1;                             /* '<Root>/In1' */
} ExtU_Derivative_T;
```

There will be two more levels of packaging for the monitor that will be used later on for convenient DMA transfers. First, all three structs will be packaged in a single container. Then, for all critical tasks that will require DMA transfer and that have precedence relations between them (i.e. the completion of one triggers the beginning of the next), they will be placed in a single struct so that the DMA can transfer data with a single command. The structs, presented in Listing 9, are not all located in the same file. The model contains pointers that must be assigned to static variables manually. Examples are located in the ert_main.c file accompanying each generated project.

Listing 9: Structures of structures to facilitate DMA transfer.

```
/* LOCATED IN critical.h */
typedef struct {
  RT_MODEL_Derivative_T Derivative_M;
  ExtU_Derivative_T Derivative_U;
  ExtY_Derivative_T Derivative_Y;
} DerivativeStruct;

typedef struct {
  RT_MODEL_AirbagModel_T AirbagModel_M;
  ExtU_AirbagModel_T AirbagModel_U;
  ExtY_AirbagModel_T AirbagModel_Y;
} AirbagModelStruct;

/* LOCATED in monitor main.c */
typedef struct {
  AirbagModelStruct airbagModelStruct_0;
  DerivativeStruct derivativeStruct_0;
} DMAPackageStruct;
```

8 References

[1] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 34–43, IEEE, 2011.

[2] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, &Amp; Tools for Real-time Systems*, LCTES '95, (New York, NY, USA), pp. 88–98, ACM, 1995.

[3] C. Bolchini and A. Miele, "Reliability-driven system-level synthesis for mixed-critical embedded systems," *Computers, IEEE Transactions on*, vol. 62, no. 12, pp. 2489–2502, 2013.

[4] G. BX, "Evolutionary algorithms: Overview, methods and operators." http://www.geatbx.com/docu/docutoc.html#TopOfPage. Accessed: 2015-03-12.

[5] P. Eles, A. Doboli, P. Pop, and Z. Peng, "Scheduling with bus access optimization for distributed embedded systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 5, pp. 472–491, 2000.

[6] C. Bolchini *et al.*, "A multi-objective genetic algorithm framework for design space exploration of reliable fpga-based systems," 2010.

[7] JGAP, "JGAP: Java Genetic Algorithms Package." http://jgap.sourceforge.net/, 2012.