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. Figure 2 depicts an equivalent Gantt chart.

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. Port Heptane from MIPS to Nios architecture.
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.

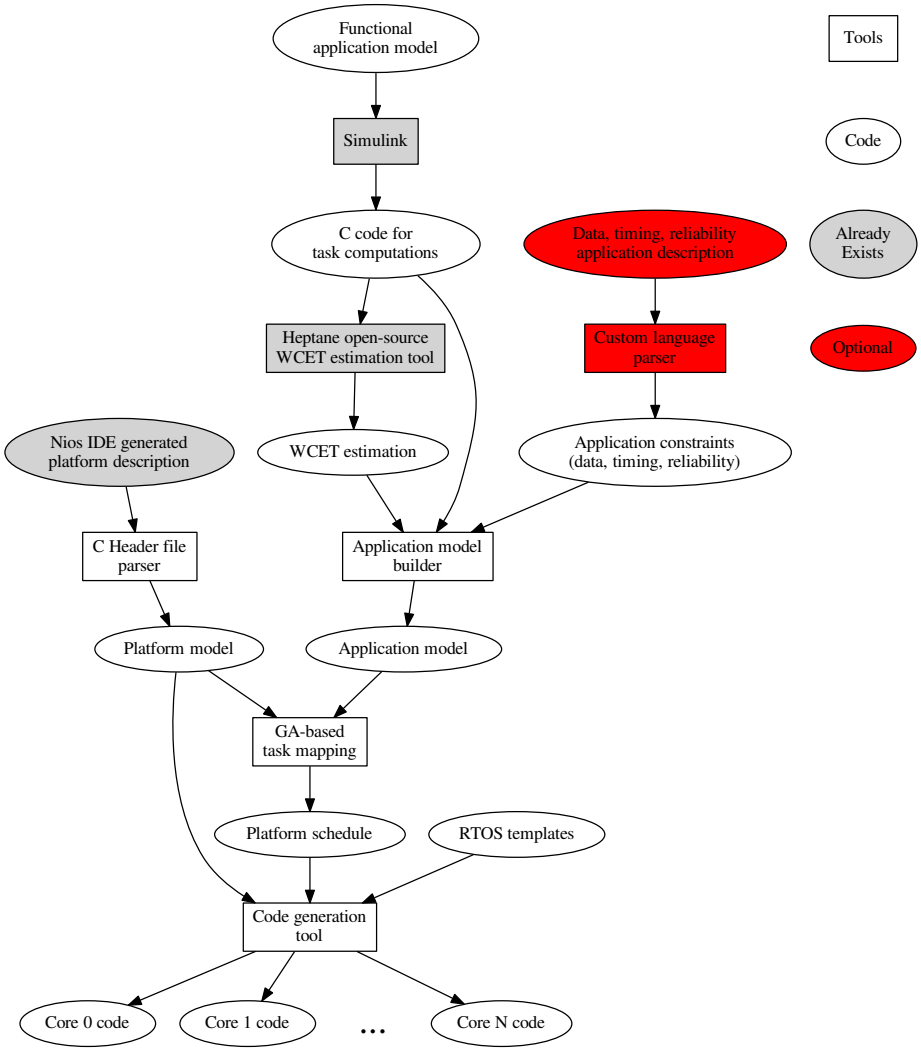

Figure 1: The proposed tool architecture.

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.

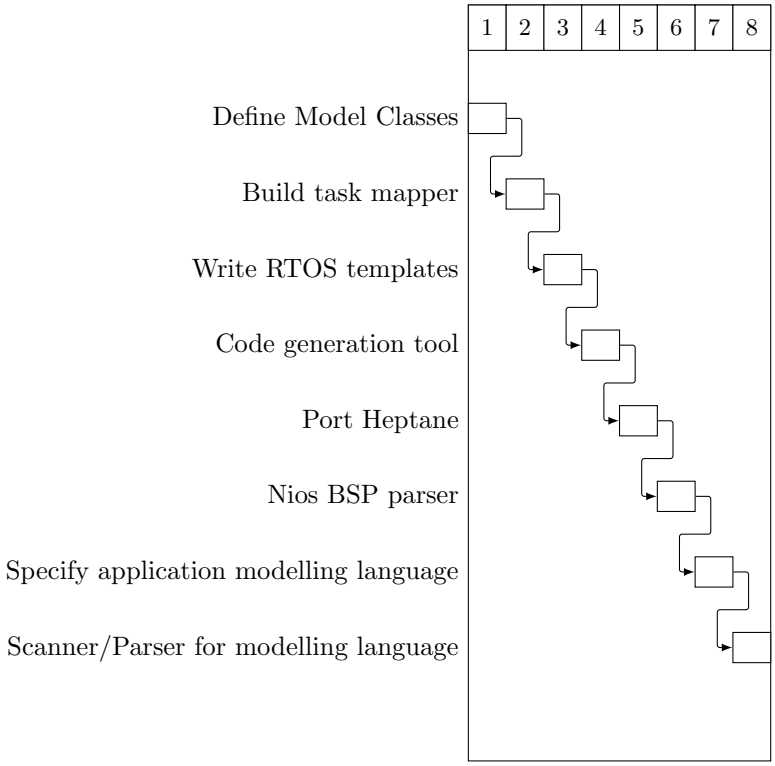

Figure 2: Timeline for development of main tool components

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 Synchronous Languages - Building off the past

4.2 Code Rules

Any data passed between tasks must be done through pointers. Functions will return void pointers to predetermined data structures. These pointers will be passed between tasks. If tasks cannot share the same memory space, then the size of the data structure must also be known and the data will be copied to the appropriate location.

5 Application Mapping

The initial scheduling algorithm will be taken from Bolchini and Miele [1]. 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 [2].

Figure 3 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.

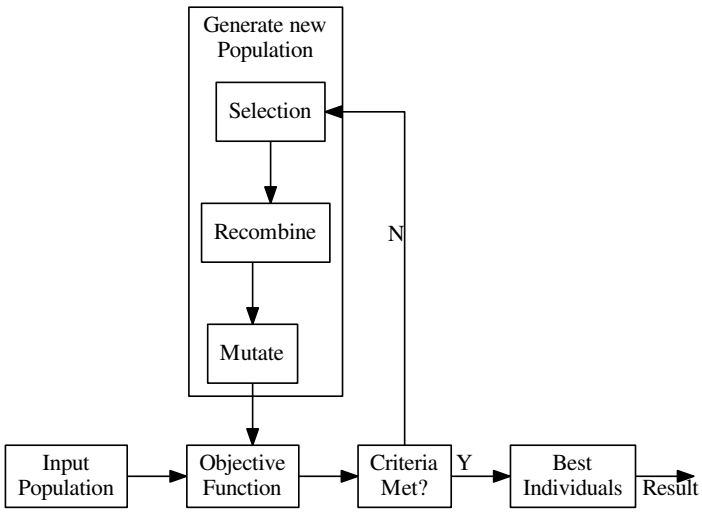

Figure 3: The basic structure of a genetic algorithm [2].

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* [3]. 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 [3] in Figure 4. 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.

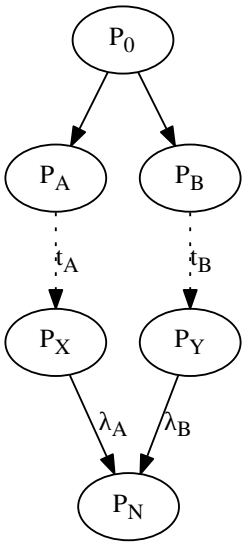

Figure 4: Example of modified partial critical path priority selection [3].

The implementation details of the genetic algorithm will be adopted from [1]. 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 [4]. 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 [5] is used to implement the genetic algorithm.

5.1 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)

6 Code Generation

Code generation will be decomposed into several phases.

References

[1] 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.

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

[3] 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.

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

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