Hardware Implementation of RMS Scheduler

Jake Hafele, Thomas Gaul
jmhafele@iastate.edu, tvgaul@iastate.edu
CPRE 558 Section 1
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50014

Abstract

This project addresses the problem of software implementations of task schedulers, which can require large overhead in resources such as memory, energy, and context switching for embedded systems applications. Alternatively, by utilizing a hardware implementation, we could utilize either an FPGA or ASIC to implement a task scheduler, which could lead to potential benefits for resource usage for memory, energy, or context switching. For our design, we have modeled a real-time RMS task scheduler on an FPGA development board with the goal of comparing the synthesis results and speed of context switching for high speed and low energy applications with embedded systems. To evaluate our design, we will analyze functional wave forms and physical results on a reconfigured FPGA. We will also compare the performance of the hardware implementation with a generalized software scheduler model with an RMS implementation to determine where further work could be done.

I. Introduction

Our idea of hardware-based scheduling is novel within the scope of course content in CPRE 558, but it has been explored in many different applications by researchers. The idea was initially suggested in 1995 in the paper "Hardware implementation of a real-time operating system" [1]. From there, it was researched in several ways for the last 28 years. We explore a handful of these previous works as inspirations for our implementation of hardware-based scheduling.

The first paper in 1995 proposed implementing major parts of a real-time operating system in hardware and comparing it to its hardware counterparts. They implemented their primary hardware kernel on an FPGA and found that it performed overwhelmingly well. It performed many times faster than its software counterpart, and the kernel took up less than half the space. The only cost of this concept is the hardware development costs. The next paper we looked at explored the pros and cons of three setups: software, software with a coprocessor, and hardware-software [2]. They found that the hardware outperforms either of the software-exclusive versions and takes up less area and power than a coprocessor solution. Once again, the only issue with this solution is implementation complexity.

Some of the more modern papers look into some other aspects of application outside of overall performance. In the paper "A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems," they suggest the possibility of creating a scheduling accelerator that could run on an FPGA in addition to a processor-implemented on an FPGA, which could outperform a hardcore processor due to the advantages of a hardware scheduler [3]. In addition, the paper explores the option of more complicated algorithms that don't take bandwidth due to the hardware accelerator. The next paper we looked at, "A High-Performance Real-Time Hardware Scheduler" looked into algorithms for scheduling a multicore system, which is typically processor-heavy for optimization [4]. Like the early papers, the authors found that these hardware options outperformed software options in every category except implementation complexity.

For our project, we will explore implementing our own hardware implementation of an RMS scheduler to compare the performance and trade offs versus a traditional software scheduler for a low resource embedded systems application.

II. Project Objectives and Scope

One major downside to software scheduled pipelines is the large overhead required with implementing

schedulers. For many embedded systems application, this overhead can include both the added context switching resources for switching tasks, but also the resources lost in implementing an Operating System versus using a bare metal application for low cost devices. This could lead to a desire for a hardware scheduler implementation, which could be used on SoC applications in parallel with a bare metal micro controller. Our goal is to implement a basic hardware scheduler with an RMS implementation to show the feasibility and benefit versus a traditional software scheduler with RMS.

A. System Model

Our project and design are centered around embedded systems applications which have limited amounts of resources, memory, or power that can be utilized for the target system. To implement a traditional software scheduler, there may be additional overhead for including an operating system to use a RMS scheduler, which on its own could lead to more resource usage in memory, energy, or context switching.

B. Problem Statement

The major problem we our seeking to solve is the barrier to implementing task schedulers, and how utilizing hardware based implementations on either an FPGA or ASIC could help reduce the energy utilized or time spent context switching with traditional software schedulers.

C. Objectives and Scope

Overall, the goal of this project will be to apply what we have learned about RMS real-time system schedulers and analyze the potential benefits of implementing a real-time hardware scheduler on an FPGA platform by utilizing a hardware approach. With this, we can compare the performance tradeoffs and benefits versus a traditional software scheduler in the context of an embedded system application.

Objectives

- Model periodic RMS task sets as learned in class
- Model RMS schedule on FPGA hardware
- Multiple queues of tasks based on priority
- Compile/Simulate RTL related to real-time system topics
- Synthesize RTL for FPGA configuration for real-time operation
- Preempt task sets based on periodic priority per RMS
- Save register states in between tasks when preempted
- Implement counter based on clock to monitor RMS ready times

III. Solution Methodology / Approach

To compare the hardware and software schedulers, we will implement a sample RMS schedule on an FPGA with a hardware architecture to mimic context switching between multiple periodic tasks. This will help address our problem of high overhead in software applications by showcasing the feasibility of a RMS hardware scheduler, and demonstrate improvements in energy consumption and context switching relative to the software counterpart.

A. Algorithms / Protocols / Architectures

An RMS Schedule consists only of periodic tasks, which makes the schedule and task set static. It is assumed for our RMS schedule that the relative deadline of the schedule is the same as the period (Pi) of each task, meaning that each task is due to complete by the time the task refreshes at the end of its respective period. Each task can vary in its computation time (Ci), which will require N time units to complete each task. The priority of the tasks will be determined statically, based on the smallest period of the active tasks. A task will switch from active to complete after it has been active for as many time units as required per the computation time. Once a periodic task refreshes, it can preempt a lower priority task (with larger period).

To ensure that our hardware application of an RMS schedule would meet all of the above criteria, we designed the task set as below in Table I. This task set includes five different tasks, with two different period and computation times. We designed our task set to ensure that there would be full utilization to test the limits of our hardware scheduler, and also to ensure that preemption would occur with tasks T0 and T1 after time unit 4. The sample schedule for the given task set parameters can be seen in Figure 1.

B. Illustrative Example

The referenced task set parameters will be inserted into our Hardware design, which will be synthesized "at compile time" for the static RMS schedule, to

Task	Period (Pi)	Computation Time (Ci)
T0	4	1
T1	4	1
T2	8	1
T3	8	1
T4	8	2

Table I: RMS Schedule Parameters

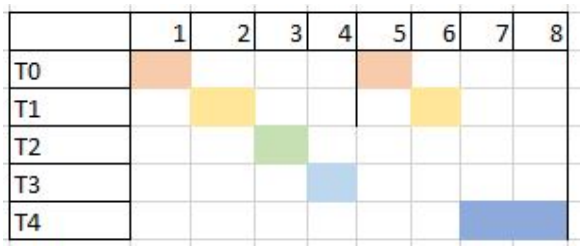

Figure 1: Sample RMS Schedule with Task Preemption

repeat indefinitely while running. We will be expecting the same RMS schedule including the correct periods, computation times, and task preemption for both the simulation waveforms and the synthesized FPGA result on an OLED display.

IV. Implementation / Simulation Architecture

For this project, we will be coding all of our designs using Verilog, which will be compiled and simulated using ModelSim on the Coover Linux VDI servers. We used two lab computers for this, and synthesized our FPGA design using Vivado, with a Xilinx FPGA Arty A7 on the Basys 3 Development Board, which was owned by one of our teammates personally. If other FPGAs were needed in the future, a Zedboard could be checked out from the ETG and could also utilize Vivado, with other pin out constraints updated.

The overall design of our hardware RMS scheduler would be to hook up into a standard processor design connecting to the program counter and the register files of a standard processor. The design adds on the hardware of more register files and program counters (one for each task) and the control hardware at the benefit of decreasing the context switching costs. Our design consists of a time counter, task control, a 3 to 8 on the hot decoder, a register file for each task, a task state register, and a multiplexer for both the program counter and the register data.

Each component plays an important role in completing the RMS task scheduling of finding which tasks are not complete and telling the priority incomplete task to complete.

A. Time Counter

The time counter is a basic counter that keeps track of elapsed time to ensure tasks are reset at the correct deadline and to ensure we do not run into overflow issues. It resets on the Least Common Multiple of all the tasks.

B. Control

The control module actually selects which task to be active with RMS scheduling based on the period, which is set at compile time and resets tasks as they hit their deadline. In the event of overload, when not all tasks can be completed, it resets them regardless of their completion status. The control module completes two major tasks. It tasks the current time compares it to the periods of the tasks, and resets them as necessary in addition to their LCM. It then updates the task's next deadline based on its period. Its other task is comparing uncompleted tasks to select which task, if any, to run. If all tasks are complete, not tasks are running, and it just sits waiting for a task to reset due to the deadline.

C. 3 to 8 One Hot Decoder

The 3 to 8 one hot decoder takes input from the control module to select which task is running top ensure that only the correct register file and the program counter are updated for the active task.

D. Task Register File (32bit DFF)

Each task has its own register file to allow for fast context switching to avoid dealing with pushing data to memory which is time costly. It reads and writes the data being manipulated by the program in the ALU and memory, which we have modeled in our test bench as a simple adder. The inputs to the ALU are multiplexed between all the tasks and the write enable ensures only the correct task writes it data back to a task's registers.

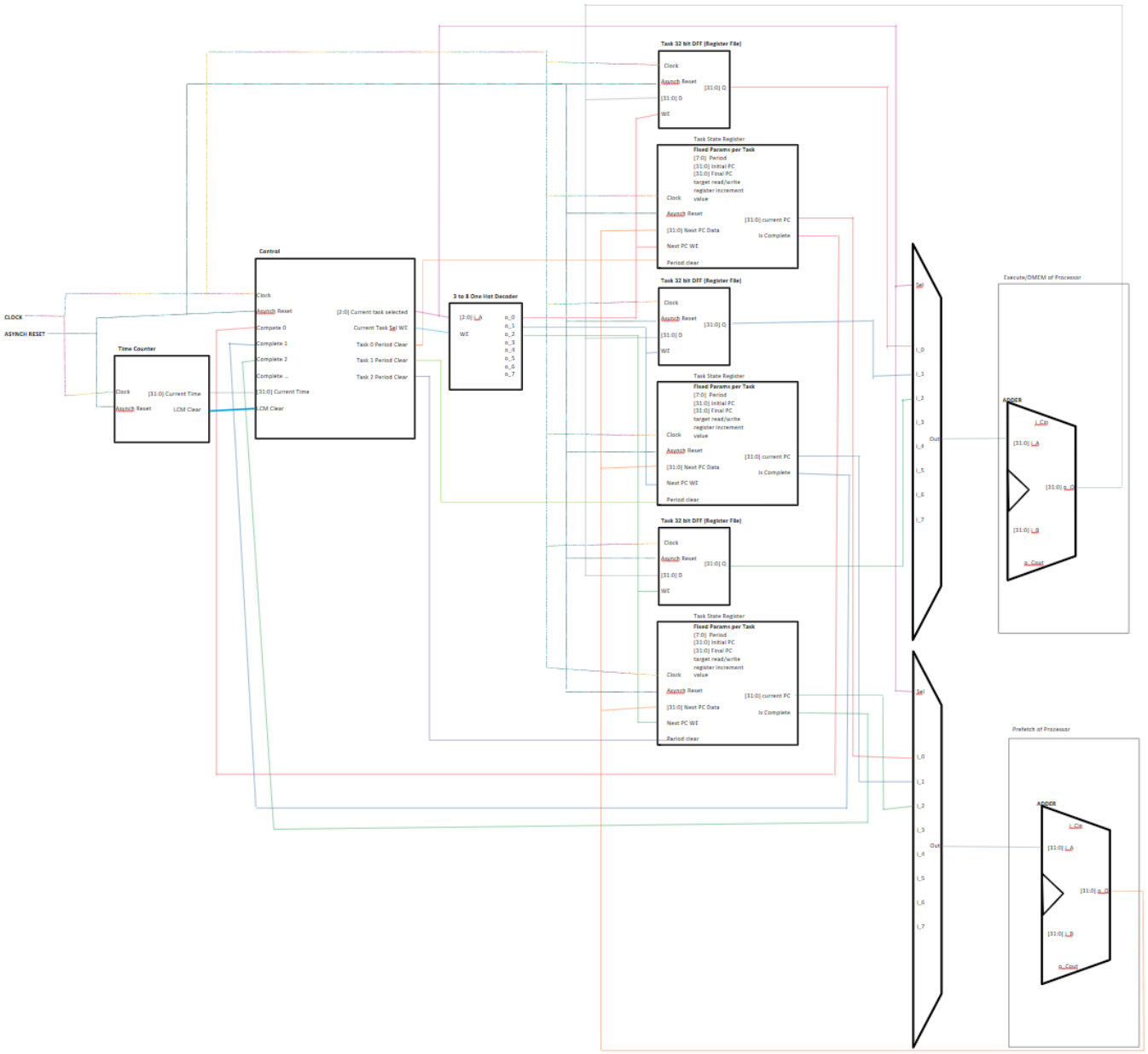

Figure 2: Top Level Schematic

E. Task State Register

The task state register tracks the state of each task in terms of completion and its program counter. Each task has its section of program memory and starting location in memory. Additionally, each task has its program counter to track where it is throughout its program, and finally, it has a final program counter to track where in memory the task has been completed execution and can then set its completed signal telling the control module it is completed. Upon reset, it is then set to is starting program counter. By basing the completion of a program counter, it ensures that if a

task is completed faster than its worst-case completion time, it passes slack off to completing other tasks. The current PC is then multiplexed between all the tasks and handled by the processor's logic based on the instruction to handle branches a jumps, and similarly to the Task registers, the write enable ensures only the active task is updated.

V. Evaluation

A. Testing Procedure

To test our implementation we used a variety of tools each module listed in the implementation section we created a separate test bed in VHDL to ensure that each module was functioning as intended. Those test beds were run in Questasim with a do file that is in each sub-module folder. Once all sub-modules were put together we then created a top-level test in Questasim with the same task set as in Figure 1. We ran all these tests on the Linux Virtual Machines the university provides.

To gain synthesis data and to run on an FPGA we synthesized and generated bitstream in Vivado then flashed it to an FPGA with an OLED display shown in figure that outputs important data about the test and stepped through the program generating the clock input with a denounced push button also on the FPGA.

We then compared the data in the waveforms and from stepping through the tasks in the FPGA and compared them to that of the hand-completed RMS schedule and ensured they lined up perfectly.

B. Analysis

To compare to a software RMS scheduler, we will consider two theoretical processors of a 32-bit single-cycle design: one that uses a software-scheduled RMS with an RTOS and another that uses our architecture to schedule tasks with RMS. The benefit of our design in hardware is context switching and other overheads imposed by the OS. Where every time there needs to be a context switch the current task needs to have its registers saved to memory, and the new task needs to have its data loaded to memory. In every instance where preemption needs to occur, we see an overhead of 64 clock cycles, one for each memory reference, which is optimistic given the latency for larger memories. This can get very costly with very many context switches. This is ignoring the other overhead of an operating system to have tasks preempted and calculating deadlines. Our hardware handles all those overhead costs at the same time as the other computations. This comes at the cost described in Vivado results.

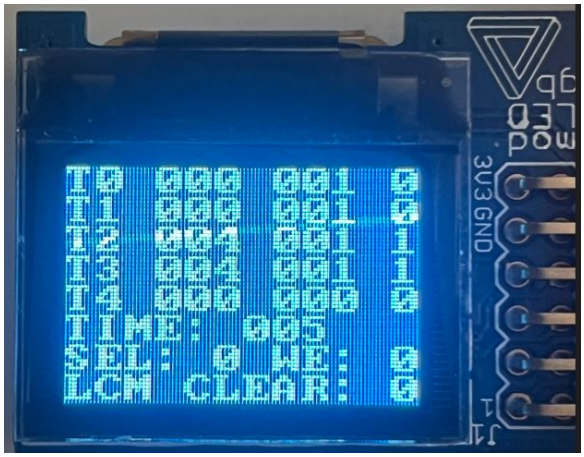

Figure 3: FPGA Implementation with OLED Screen

Vivado Implementation Results:

- Worst Negative Slack: 3.937ns
- Total Power Consumption: 0.089W
- Look Up Tables Used: 1120

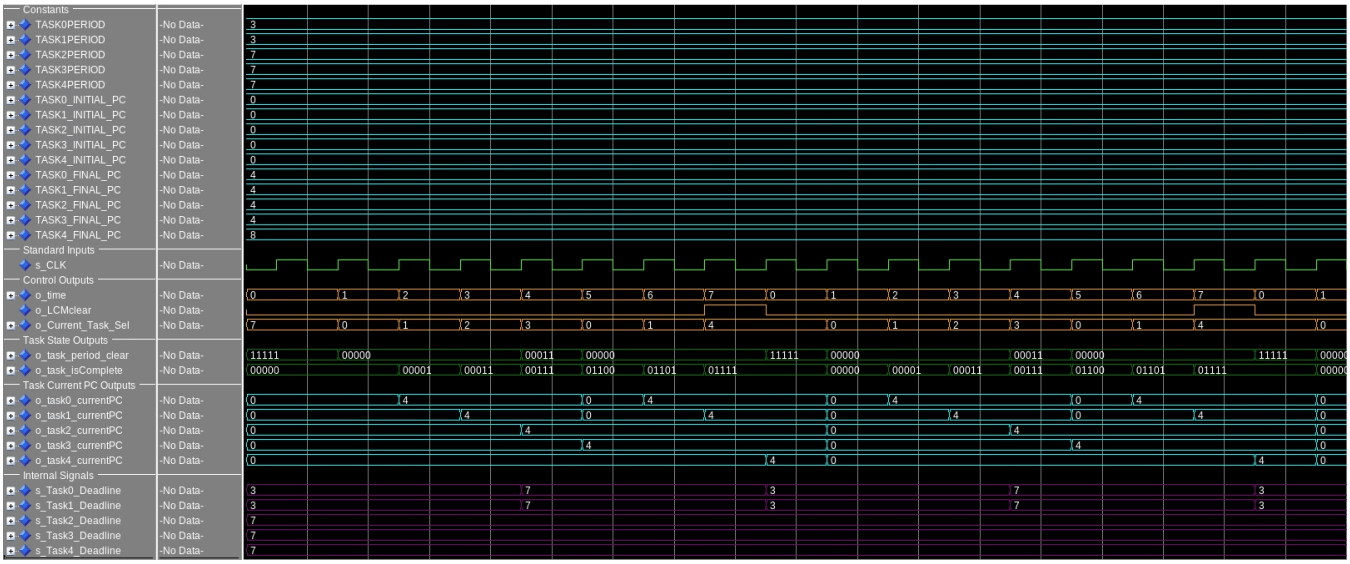

Figure 4: Sample RMS Schedule Ran on Hardware Simulation

VI. Conclusions

From this experiment, we learned about the benefits and trade offs of applying existing scheduler techniques from software to hardware applications such as an FPGA. We were able to design a hardware architecture system based on previous classes such as CPRE 281 and CPRE 381 and relate them to topics explored in CPRE 558, such as real-time systems and task schedulers. We were able to successfully model an RMS schedule in theory, in simulation, and in synthesis on an FPGA, all with the same results, guaranteeing our design functioned properly. This would enable future work to be done with hardware schedulers with this design as a baseline, where the same OLED display or FPGA top level design could be reused to save on work.

Learning objectives are listed in Table II

A. Potential Future Exploration

Changes could be made easily to this architecture to complete other scheduling algorithms or to handle other scenarios, such as aperiodic events.

1) EDF Implementation

EDF could be implemented with this architecture by tracking a down counter with the worse case computation time in the task state register that is decremented every clock cycle the processor allows to that task to run, and then the control units compare remaining time in addition to completion status to select the active tasks.

2) Aperiodic Events

This scheduler could easily be added to handle aperiodic events. The only change would be to have the program counter for aperiodic events be reset upon a set of a bit in memory from another task or from I/O trigger it to be scheduled with whatever priority the aperiodic task is given.

3) Integration to Full Processor

We would be able to better analyze the energy tradeoffs of our RMS scheduler if it were hooked up to a full pipelined processor, such as our 5-stage pipelined MIPS processor designed in another course, CPRE381. Since all of the RTL is already generated in that lab, it would be useful to reuse and compare our hardware scheduler with an existing processor to compare the addition of power consumption.

Project learning objectives	**Status**	**Document Pointers**
Description of project goal, scope, and relevant requirements	Fully Completed	Section II, Page 1
Description of solutions	Fully Completed	Section III, Page 2
Description of implementation details	Fully Completed	Section IV, Page 3
Testing and Evaluation	Mostly Completed	Section V, Page 5
Overall Project Success Assessment	Fully Successful	

Table II: Self-Assessment of Project Completion

VII. Team Member Contribution

Team Member contribution can be seen in Table III

Tasks/Member	**Jake H**	**Thomas G**
Literature survey	None	Previous Works
Design	RMS Schedule, Brainstorming	Top Level Schematic, Brainstorming
Implementation	Top Level Module, Register File, Counter	Task State Register, Control Module
Testing/Evaluation	Unit Testing, FPGA Synthesis	Unit Testing, Waveform Validation
Preparation of the Report and Presentation	Introduction, Objectives, Methodology, Slides	Implementation, Evaluation, Slides
Total percentage of contribution to the project	50%	50%

Table III: Project Tasks and Member Contributions

References

[1] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware implementation of a real-time operating system," in *Proceedings of the 12th TRON Project International Symposium*, pp. 34–42, 1995.

[2] M. Vetromille, L. Ost, C. Marcon, C. Reif, and F. Hessel, "Rtos scheduler implementation in hardware and software for real time applications," in *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*, pp. 163–168, 2006.

[3] Y. Tang and N. W. Bergmann, "A hardware scheduler based on task queues for fpga-based embedded real-time systems," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1254–1267, 2015.

[4] D. Derafshi, A. Norollah, M. Khosroanjam, and H. Beitollahi, "Hrhs: A high-performance real-time hardware scheduler," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 897–908, 2020.