AFRL-OSR-VA-TR-2014-0306

An Adaptive Property-Aware HW/SW Framework for DDDAS

Phillip Jones
IOWA STATE UNIVERSITY

10/29/2014
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTC
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 21-10-2014	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* 09/30/2011 - 04/30/2014

4. TITLE AND SUBTITLE

An Adaptive Property-Aware HW/SW Framework for DDDAS

5a. CONTRACT NUMBER
FA9550-11-1-0343

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Jones, Phillip, H
Cytron, Ron
Gill, Christopher
Zambreno, Joseph,
Elia, Nicola

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Iowa State University, Dept. of Electrical and Computer Eng.,2215 Coover Hall,
Ames, IA 50011

Washington University, Dept. of Computer Science and Engineering., Bryan Hall, CB
10045, 1 Brookings Dr., St. Louis, MO 6130

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DR-04 USAF AFMC AFOSR/RTC
Air Force Office of Scientific Research (AFOSR)
875 N. Randolph St.
Arlington VA 22203 - USA

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution A - Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project focused on the design of an adaptable computing infrastructure to support Dynamic Data-Driven Application Systems (DDDAS) in the context of Unmanned Aerial Vehicles (UAVs). A unifying theme that drove our research was the concept of responding and adapting to surprise. In support of responding to surprise, research was pursued to allow elements of the computing platform to adapt to the system or tasks moving between different modalities. Key areas of focus that support adaptation to surprise and changing modalities were: 1) Data structures that span HW/SW boundaries, 2) Instruction Set Architecture (ISA) extensions and hardware support, and 3) establishing performance metrics.

15. SUBJECT TERMS

DDDAS, HW/SW co-design, Real-time, Mixed Criticality, Computer Architecture

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Phillip Jones
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	5	**19b. TELEPHONE NUMBER** *(include area code)* 515-294-9208

AFOSR Final Performance Report

(To: technicalreports@afosr.af.mil, frederica.darema@afosr.af.mil)

Project Title: An Adaptive Property-Aware HW/SW Framework for DDDAS

Award Number: FA9550-11-1-0343

Start Date: 09/30/2011

Reporting Period: 09/30/2011 – 04/30/2014

Program Manager
Dr. Frederica Darema
Air Force Office of Scientific Research
e-mail: frederica.darema@afosr.af.mil
phone: 703-588-1926

Principal Investigator:
Phillip Jones
Department of Electrical and Computer Engineering
Iowa State University
329 Durham Center
Ames, IA, 50011
e-mail: phjones@iastate.edu
phone: 515-294-9208

Co-Investigators:
Ron Cytron
Department of Computer Science and Engineering
Washington University
St. Louis, MO

Christopher Gill
Department of Computer Science and Engineering
Washington University
St. Louis, MO

Joseph Zambreno
Department of Electrical Computer and Engineering
Iowa State University
Ames, IA

Nicola Elia
Department of Electrical Computer and Engineering
Iowa State University
Ames, IA

An Adaptive Property-Aware HW/SW Framework for DDDAS

1. Summary and Objectives

This project focused on the design of an adaptable computing infrastructure to support DDDAS systems in the context of Unmanned Aerial Vehicles (UAVs). A unifying theme that drives our research is the concept of responding and adapting to surprise. Some examples of surprise are: 1) in-flight changes in mission objectives, 2) unexpected encounters with friends or foes, and 3) changes in the environment. Two examples of environmental changes could be a chipped blade (which changes the dynamics of the UAV), or unexpected turbulence.

In support of responding to surprise, research was pursued to allow elements of the computing platform to adapt to the system or tasks moving between different modalities (e.g. real-time, high-performance, small-footprint, energy-conserving).

Key areas of focus that support adaptation to surprise and changing modalities are:

- **Data structures that span HW/SW boundaries:** In this area, we explored SW data structures that can dynamically change their internal makeup to best suite a task's current modality (e.g. maximizing average throughput when in high-performance mode, or bounding worst-case access time when in a real-time mode), or migrating their storage and operators between SW/HW boundaries to improve performance and/or improve predictability. [1],[4]

- **Instruction Set Architecture (ISA) extensions and hardware support:** This area focuses on the design and evaluation of specialized hardware to support common tasks that can be found in UAV applications (e.g. sensor acquisition/processing, issuing actuator commands, signal processing, vehicle controller logic) for the purpose of increased computational efficiency (explicitly exploiting concurrency) and/or enforcing guarantees on task behavior (e.g. tightening bounds on execution variation, guarantees on resources availability) [1],[2],[4],[5]

- **Establishing performance metrics:** i) tolerance of vehicle stability to jitter in its control loop, ii) overheads associated with dynamically morphing data structures or migrating them across SW/HW boundaries. [3]

2. Accomplishments and Highlights

2.1 A Scalable hardware scheduler architecture for real time systems [1]. In Dynamic Data-Driven Application Systems, applications must dynamically adapt their behavior in response to objectives and conditions that change while deployed. One approach to achieve dynamic adaptation is to offer middleware that facilitates component migration between modalities in response to such dynamic changes. The triggering, planning, and cost evaluation of adaptation takes place within a scheduler. Scheduling overhead is a major limiting factor for implementing dynamic scheduling algorithms with high frequency timer-tick resolution in real time systems. In [1], we present a scalable hardware scheduler architecture for real time systems that reduces processing overhead and improves timing predictability of the scheduler. A new hardware priority queue design is presented, which supports insertions in constant time, and removals in O(log n) time. The hardware scheduler supports three (Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), and priority based) scheduling algorithms, which can be configured during run-time.

2.2 Hardware-software architecture for priority queue management in real-time and embedded systems [4]. The use of hardware-based data structures for accelerating real-time and embedded system applications is limited by the scarceness of hardware resources. Being limited by the silicon area available, hardware data structures cannot scale in size as easily as their software counterparts. In this work, we extend on our previous work in [1], and assert a hardware-software co-design approach is required to elegantly overcome these limitations. We present and evaluate a hybrid priority queue architecture that includes a hardware accelerated binary heap that can also be managed in software when the queue size exceeds hardware limits. A memory mapped interface provides software with access to priority-queue structured on-chip memory, which enables quick and low overhead transitions between hardware and software management. As an application of this hybrid architecture, we present a scalable task scheduler for real-time systems that reduces scheduler processing overhead and improves timing determinism of the scheduler.

2.3 Adaptable binary search tree [1]. In [1], we report on our efforts to improve software timing predictability by using multiple implementations of abstract data types, using only software (earlier in [1] presents a software/hardware co-design approach). We propose the idea of a hybrid binary search tree implementation. Sections of the tree would be implemented as AVL when it is predicted many search operations would be executed, and would be implemented as Red-Black when it is predicted a section of the application will execute many insertion/deletion operations. Specifically, in this work, we introduce a new technique for efficiently converting between AVL tree and Red-Black tree implementations.

2.4 Plant-on-Chip system design approach [3]. Digital control systems are traditionally designed independent of their implementation platform, assuming constant sensor sampling rates and processor response times. Applications are deployed to processors that are shared amongst control and non-control tasks, to maximize resource utilization. This potentially overlooks that computing mechanisms meant for improving average CPU usage, such as cache, interrupts, and task management through schedulers, contribute to nondeterministic interference between tasks. This response-time jitter can result in reduced system stability, motivating further study by both the controls and computing communities to maximize CPU utilization, while maintaining physical system stability needs. In [3], we describe a field-programmable gate array (FPGA)-based embedded software platform coupled with a hardware plant emulator (as opposed to purely software-based simulations or hardware-in-the-loop setups) that forms a basis for safe and accurate system analysis. We model and analyze an inverted pendulum to demonstrate that our setup can provide a significantly more accurate representation of a real system.

2.5 Cache design for mixed criticality real-time systems [2, 5]. In Dynamic Data-Driven Application Systems (DDDAS), applications must dynamically adapt their behavior in response to objectives and conditions that change while deployed. Often these applications may be safety critical or tightly resource constrained, with a need for graceful degradation when introduced to unexpected conditions. In [2], we motivate and provide a vision for a dynamically adaptable mixed critical computing platform to support DDDAS applications. We then specifically focus on the need for advancements in task models and scheduling algorithms to manage the resources of such a platform. We discuss the short comings of existing task models for capturing important attributes of our envisioned computing platform, and identify challenges that must be addressed when developing scheduling algorithms that act upon our proposed extended task model. These investigations lead to our work on criticality-aware cache architectures, highlight below.

Tasks sharing caches in mixed criticality systems are a source of interference for safety critical tasks. This sharing of memory not only leads to worst-case execution time (WCET) pessimism, but also affects the response time of safety critical tasks. In [5], we present a criticality aware cache design that implements a Least Critical (LC) cache replacement policy, where a least recently used non-critical cache line is replaced during a cache miss. The cache acts as a Least Recently Used (LRU) cache if there are no critical lines or if all cache lines are critical in a set. In our design, data within a certain address space is given higher preference in the cache. These critical address spaces are configured using critical address range (CAR) registers. The new cache design was implemented in a Leon3 processor core, a 32bit processor compliant with the SPARC V8 architecture. Experimental results are presented that illustrate the impact of the Least Critical cache replacement policy on the response time of critical tasks, and on overall application performance as compared to a conventional LRU cache policy.

3. Publications

[1] C. Kumar, S. Vyas, J. Shidal, R. Cytron, C. Gill, J. Zambreno and P. Jones, *Improving System Predictability and Performance via Hardware Accelerated Data Structures*, Proceedings of Dynamic Data Driven Application Systems (DDDAS), June, 2012

[2] C. Kumar, S. Vyas, R. Cytron, C. Gill, J. Zambreno and P. Jones, *Scheduling Challenges in Mixed Critical Real-time Heterogeneous Computing Platforms*, Proceedings of Dynamic Data Driven Application Systems (DDDAS), June, 2013.

[3] S. Vyas, C. Kumar, J. Zambreno, C. Gill, R. Cytron and P. Jones, *An FPGA-based Plant-on-Chip Platform for Cyber-Physical System Analysis*, IEEE Embedded Systems Letters (ESL), vol. 6, no. 1, pp. 4-7, 2014.

[4] C. Kumar, S. Vyas, R. Cytron, C. Gill, J. Zambreno and P. Jones, *Hardware-Software Architecture for Priority Queue Management in Real-time and Embedded Systems,* International Journal of Embedded Systems (IJES), vol. 6, no. 4, pp. 319-334, 2014.

[5] C. Kumar, S. Vyas, R. Cytron, C. Gill, J. Zambreno and P. Jones, *Cache Design for Mixed Critical Real-Time Systems*, Proceedings of the International Conference on Computer Design (ICCD), October, 2014.

4. Presentations

1. C. Kumar, S. Vyas, J. Shidal, M. Rich, R. Cytron, C. Gill, J. Zambreno N. Elia, and P. Jones, An Adaptive Property-Aware HW/SW Framework for DDDAS, Proceedings of Dynamic Data Driven Application Systems (DDDAS) Workshop, Omaha, NE, June, 2012.

2. C. Kumar, S. Vyas, M. Rich, R. Cytron, C. Gill, J. Zambreno N. Elia, and P. Jones, Scheduling Challenges in Mixed Critical Heterogeneous Real-Time Computing Platforms, Proceedings of Dynamic Data Driven Application Systems (DDDAS) Workshop, Barcelona, Spain, June, 2013.

3. C. Kumar, S. Vyas, J. Shidal, M. Rich, R. Cytron, C. Gill, J. Zambreno N. Elia, and P. Jones, An Adaptive Property-Aware HW/SW Framework for DDDAS, Dynamic Data Driven Application Systems (DDDAS) PI Meeting, D.C., September, 2013.

4. C. Kumar, S. Vyas, R. Cytron, C. Gill, J. Zambreno and P. Jones, *Cache Design for Mixed Critical Real-Time Systems,* Proceedings of the International Conference on Computer Design (ICCD), Seoul, Korea, October, 2014.

5. Trainees as part of this AFOSR project

Graduate Students – Iowa State University, Ames, IA

2011 – 2014	Chetan Kumar
2011 – 2014	Sudhanshu Vyas
2012 – 2013	Matt Rich

Graduate Students – Washington University, St. Louis MO

2011 – 2014 John Shidal

6. Conclusions

As a result of this work, we have better insight into computer architectures and tools for supporting and evaluating systems with dynamic needs. We are currently leveraging our work with Mixed-Criticality cache architectures and our Plant-on-Chip physical system evaluation approach to begin experimentation with closer to real-world workloads and vehicles. We estimate within the next year will we be deploying our ideas on actual battery-powered quadcopters for evaluation.

Appendix: Publications Attached

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2012) 1–9

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Improving System Predictability and Performance via Hardware Accelerated Data Structures

Chetan Kumar N G[a], Sudhanshu Vyas[a], Jonathan A. Shidal[b], Ron K. Cytron[b], Christopher D. Gill[b], Joseph Zambreno[a], Phillip H. Jones[a,1]

[a] *Iowa State University, Department of Electrical and Computer Engineering, Ames, IA*
[b] *Washington University, Computer Science and Engineering, St. Louis, MO*

Abstract

In Dynamic Data-Driven Application Systems, applications must dynamically adapt their behavior in response to objectives and conditions that change while deployed. One approach to achieve dynamic adaptation is to offer middleware that facilitates component migration between modalities in response to such dynamic changes. The triggering, planning, and cost evaluation of adaptation takes place within a scheduler. Scheduling overhead is a major limiting factor for implementing dynamic scheduling algorithms with high frequency timer-tick resolution in real time systems. In this paper, we present a scalable hardware scheduler architecture for real time systems that reduces processing overhead and improves timing predictability of the scheduler. A new hardware priority queue design is presented, which supports insertions in constant time, and removals in $O(\log n)$ time. The hardware scheduler supports three (Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), and priority based) scheduling algorithms, which can be configured during run-time. The interface to the scheduler is provided through a set of custom instructions as an extension to the processors instruction set architecture. We also report on our experience migrating between two implementations of an ordered-set implementation, with the goal of providing predictable performance for real-time applications.

Keywords: hardware scheduler, real-time system, priority queue, ordered set, hardware accelerated data structure

1. Introduction

In the context of Dynamic Data-Driven Applications Systems (DDDAS), we have been investigating data structure implementations that are suitable for avionics missions with multimodal dynamic requirements. These data structures serve DDDAS through their ability to adapt to evolving conditions and change their behavior to preserve an application's current mission or to facilitate migration to a new mission. In particular, we are interested in applications where elements of surprise may impose sudden and perhaps short-lived modality shifts. For example, a component of an application that has been operating under best-effort conditions may be required to respond in real-time based

Email addresses: ckng@iastate.edu (Chetan Kumar N G), spvyas@iastate.edu (Sudhanshu Vyas), shidalj@wustl.edu (Jonathan A. Shidal), cytron@cse.wustl.edu (Ron K. Cytron), cdgill@cse.wustl.edu (Christopher D. Gill), zambreno@iastate.edu (Joseph Zambreno), phjones@iastate.edu (Phillip H. Jones)

[1] Corresponding author

on emergence of a threat or environmental degradations. The modalities we currently consider are best-effort (high performance), real-time, and embedded (small footprint). The general idea is that an implementation while operating in one mode can respond to a request to change its mode. The response includes not only the data structure's initial movement toward the new mode, but also a schedule indicating its projected performance as it switches modality.

The data structures we develop in this manner are "elastic" in the sense that their functionality does not change, but their implementations adapt between the modes under consideration. An earlier and very specific example of our work is a hashtable implementation that is suitable for real-time [1].

While we are investigating algorithmic solutions to achieve elastic data structures, we focus in this paper on another technique for obtaining real-time implementations, namely the development of logic deployed in hardware to achieve predictability and improve performance. The subject of our study here is a *priority queue* and its use within a real-time operating system to facilitate scheduling.

A real-time operating system (RTOS) is designed to execute tasks within given timing constraints. An important characteristic of an RTOS is predictable response under all conditions. The core of the RTOS is the scheduler, which ensures tasks are completed by their deadline. The choice of a scheduling algorithm is crucial for a real-time application. Online scheduling algorithms incur overhead, as the task queues must be updated regularly. This action is typically paced using a timer that generates periodic interrupts. The scheduler overhead generally increases with the number of tasks. A high resolution timer is required to distribute CPU load accurately based on a scheduling discipline in real-time systems, but such fine-grain time management increases the operating system overhead [2], [3].

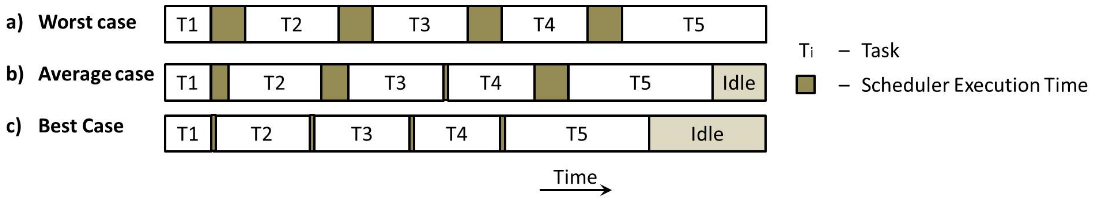


Figure 1: In order to allow analytical analysis of schedule feasibility, worst-case execution time (WCET) typically needs to be assumed. Thus, scheduler execution time variations that cause large differences between WCET and typical case execution time reduce utilization of system computing resources.

The extent to which a scheduler can ideally implement a given scheduling paradigm (e.g. EDF, RMS), and thus provide the guarantees associated with that paradigm, is in part dependent on its timing determinism. A metric for helping quantify the amount of non-determinism that is introduced to the system by the scheduler is the variation in execution time among individual scheduler invocations. This can be roughly summarized by noting its best-case and worst-case execution times. Variations in scheduler execution time can be caused by system factors such as changes in task set composition, cache misses, etc. Hence, reducing the scheduler's timing sensitivity to such factors can help increase deterministic behavior, which in turn allows the scheduler to better model a given scheduling paradigm.

Figure 1 illustrates how the variation in scheduler overhead affects processor utilization. To ensure that tasks meet their deadlines, the scheduler's worst-case execution times are often overestimated. This can cause a system to be underutilized and wastes CPU resources. In this paper, we examine how the scheduler overhead and its variation can be reduced by migrating scheduling functionality (along with time-tick processing) to hardware logic. The expected results of our efforts are increased CPU utilization and better system predictability. Another benefit is that the hardware clock provides accurate high-resolution timing.

The rest of the paper is organized as follows. Section 2 presents related work on hardware schedulers. Section 3 describes the scalable hardware scheduler architecture and implementation details. The evaluation methodology and results are discussed in Sections 4 and 5. Section 6 describes a software approach to an adaptive ordered-set data structure. Conclusions and future work are presented in Section 7.

2. Related Work

Many architectures [3], [4], [5], [6], [7], [8] have been proposed to improve the performance of schedulers using hardware accelerators. A real time kernel called FASTHARD has been implemented in hardware [3]. The scheduler of FASTHARD can handle 256 tasks and 8 priority levels. The Spring scheduling coprocessor [4] was built to accelerate scheduling algorithms used in the Spring kernel [9], which was used to perform feasibility analysis of the schedule. Mooney et al. [5] implemented a configurable hardware scheduler that provided support for three scheduling disciplines, configurable during runtime. A slack stealing scheduling algorithm was implemented in hardware [6] to support scheduling of tasks (periodic and aperiodic) and to reduce scheduling overhead. A hardware scheduler for multiprocessor system on chip is presented in [7], which implements the Pfair scheduling algorithm. A real time task manager (RTM) that implements scheduling, time management, and event management in hardware is presented in [8]. That RTM supports static priority-based scheduling and is implemented as an on-chip peripheral that communicates with the processor though memory mapped interface.

Most of the schedulers listed above implement some kind of priority based scheduling algorithm that requires a priority queue to sort the tasks based on their priority. Many hardware priority queue architectures have been implemented in the past, most of them in the realm of real-time networks for packet scheduling [10, 11, 12]. Moon et al. [10] compared four scalable priority queue architectures: fifo, binary tree, shift registers and systolic array based. The shift-register architecture suffers from bus loading, as new tasks must be broadcasted to all the queue cells. The systolic array architecture overcomes the problem of bus loading at the cost of doubling hardware storage requirements. The hardware complexity for both the shift register and systolic array architecture increases linearly with the number of elements, as each cell requires a separate comparator. This makes these architecture expensive to scale in terms of hardware resources. Bhagwan and Lin [11] proposed a new pipelined priority queue architecture based on p-heap (a new data structure similar to binary heap). A pipelined heap manager was proposed in [12] to pipeline conventional heap data structure operations. Both of these pipelined implementations of a priority queue are scalable and are designed to achieve high throughput, but at the expense of increased hardware complexity.

In this paper we present a scalable hardware priority queue architecture that implements a conventional binary heap in hardware. The priority queue is used as a part of the scheduler to improve system performance and predictability. The hardware priority queue supports constant time enqueue operations and dequeue operations in $O(\log n)$ time. The hardware utilization of the our priority queue increases logarithmically with the queue size and avoids complex pipelining logic.

3. Architecture Overview

The hardware scheduler architecture we propose is designed to reduce time-tick processing and scheduling overhead of the system. The design also uses concurrency in hardware to make the operations on a priority queue more predictable. The instruction set architecture of the processor is correspondingly extended to support a set of custom instructions to communicate with the scheduler. The hardware scheduler executes the scheduling algorithm and returns the control to the processor along with the next task to execute, and context switching is then done in software. A software timer periodically generates interrupts to check for the availability of a higher priority task. The check is accomplished using a single custom instruction that returns a preempt flag set by the hardware scheduler, based on which the processor can then choose to continue the execution of the current task or to run another. A high level block diagram of the hardware scheduler is shown in Figure 2.

The controller is the central processing unit of the scheduler. It is responsible for the execution of the scheduling algorithm. The controller processes instruction calls from the processor and monitors task queues. The timer unit keeps track of time elapsed since the start of the scheduler. This provides accurate high-resolution timing for the scheduler. The resolution of the timer-tick can be configured at runtime. The interface to the scheduler is provided through a set of custom instructions as an extension to the instruction set architecture of the processor. This removes bus dependencies for data transfer. Basic scheduler operations such as run, configure, add task, and preempt task are supported. The ready queue stores active tasks based on their priority. The sleep queue stores sleeping tasks until their activation time. The task with the earliest activation time is at the front of the sleep queue. At the core of the scheduler are the task queues, which are implemented as priority queues that keep the tasks in sorted order based on their priority (ready queue) or activation time (sleep queue).

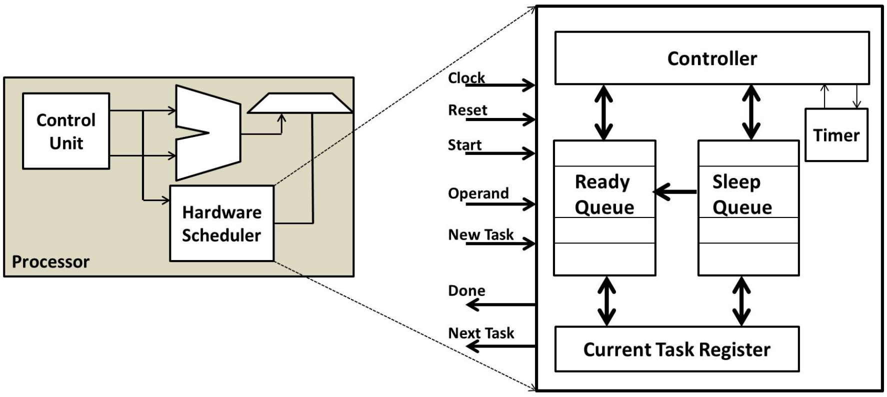


Figure 2: A high level architecture diagram of the hardware scheduler along with the custom instruction interface.

3.1. Priority Queue Architecture

One of the common software data structures for implementing a priority queue is the binary heap, which supports enqueue and dequeue operations in $O(\log n)$ time. The binary heap is stored as a linear array where the first element corresponds to the root. Given an index i of an element, $i/2$, $2i$ and $2i + 1$ are the indices of its parent, left and right child respectively. Here we present a hardware implementation of the conventional binary heap that supports enqueue and peek operations in $O(1)$ time and dequeue operations in $O(\log n)$ time. Although the dequeue operation takes $O(\log n)$ time to complete, the top-priority task can be returned immediately, so that a dequeue operation overlaps its work with that of the rest of the scheduler and the application. A high level architecture diagram for the priority queue is shown in Figure 3.

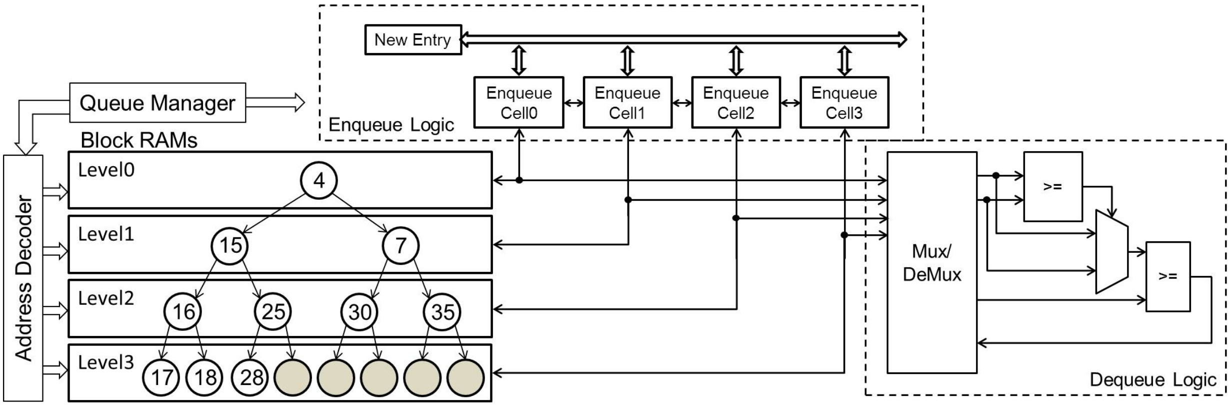


Figure 3: The priority queue architecture.

Central to the priority queue is the queue manager, which provides the necessary interface and executes operations on the queue. Elements in each level of the heap are stored in separate on-chip memories called Block Rams (BRAMs) to enable parallel access to heap elements, similar to [11, 12]. The address decoder generates addresses and control signals for the BRAM blocks. Queue operations are explained in detail below.

3.1.1. Enqueue

Enqueue operations in a binary heap are accomplished by inserting the new element at the bottom of the heap and performing compare-swap operation with successive parents until the priority of the new element is less than its parent. The worst-case behavior occurs when the priority of the new element is greater than the rest of the nodes present in the heap. In this case, the new element bubbles-up all the way to the root of the heap. We first calculate the path from a leaf node to the root. The leaf node is always one more than the current size of the queue. This path includes all ancestors from the leaf node to the heap's root. The heap property ensures that the elements in this path are in sorted order.

The shift register mechanism, shown in Figure 3, inserts a new element in constant time. This is similar to the shift-register priority queue described in [10]. Each level of the heap is mapped to an enqueue cell, which consists of a comparator, multiplexor and a register. The element to be inserted is broadcast to all the cells during an enqueue operation. The enqueue operation is then completed in the three steps shown in Figure 4. In the first step, all the elements in the path from the leaf node to root node are loaded into the corresponding enqueue cells. The address for each BRAM block is generated by the address decoder. In the second step, the comparator in each enqueue cell compares the priority of the new element with the element stored locally and decides whether to latch the current element, new element or the element above it. In the final step, the elements along with the new entry are stored back into the heap.

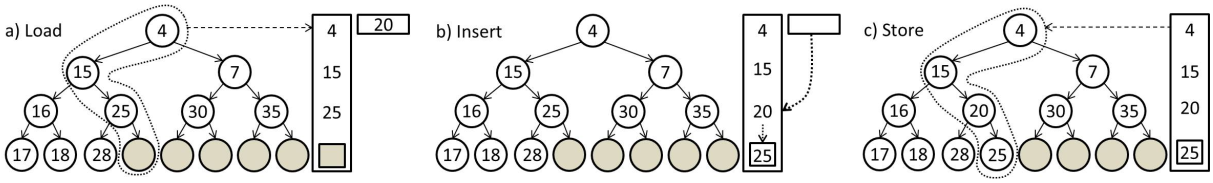


Figure 4: Steps of enqueue operation.

3.1.2. Dequeue

The dequeue operation can be divided into two parts: removing the root element from the queue (as the value to be returned by the dequeue call), and reconstruction of the heap. The root element is removed by replacing it with the last element of the queue to keep the heap balanced. The new root element is then compared with smallest of its children and swapped if the priority of new node is less than that of a child. This operation is repeated until the priority of the new root element is more than that of its children. An example of a dequeue operation is shown in Figure 5

Note that the highest priority value is obtained in constant time and as the priority queue is managed in hardware the processor is not required to wait for the operation to complete. The worst case execution time of a dequeue operation is $O(\log n)$, which would affect the rate at which consecutive operations can be performed on the queue. However, since requests for dequeue operations are paced by software, consecutive dequeue operations on the task queue are rare. Hence, this has little effect on the performance of the scheduler.

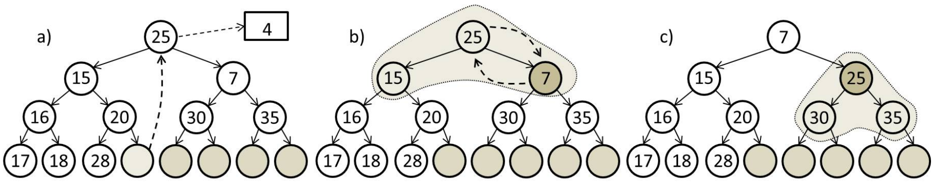


Figure 5: Steps of dequeue operation.

Comparing our approach with the related work reported in Section 2, our approach scales nicely without requiring hardware to manage pipelining and obtains suitably low latencies for the scheduler.

4. Evaluation Methodology

The hardware scheduler was deployed and evaluated on the Recongurable Autonomous Vehicle Infrastructure (RAVI) board, an FPGA development platform developed at Iowa State University. RAVI leverages Field Programmable Gate Array (FPGA) technology to allow custom hardware to be tightly integrated to a soft-core processor on a single computing device. It enables exploration of the software/hardware codesign space for designing system architectures that best fit an applications requirements. The portions of the RAVI board we used for our experiments included the Cyclone III FPGA, the on-board DDR DRAM and the UART port. The FPGA was used to implement the NIOS-II (Alteras soft-processor), the DDR stored software that was run on the NIOS-II, and the UART port supported data collection. A pictorial description of the setup is shown in Figure 6.

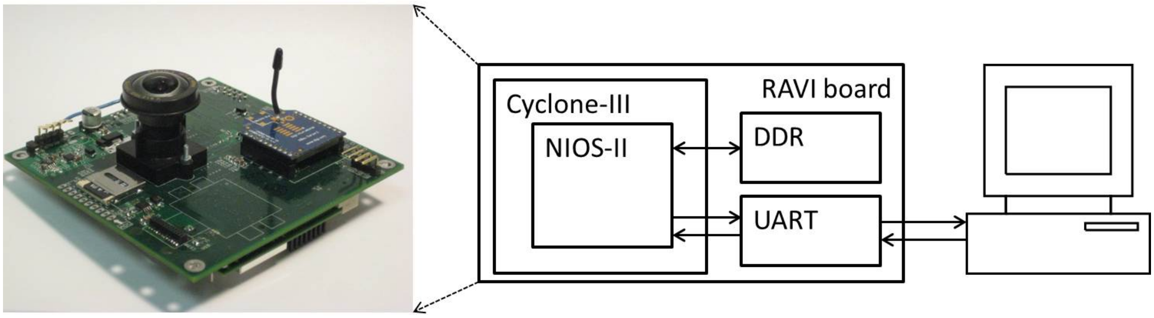


Figure 6: Evaluation platform.

The hardware scheduler is implemented as an extension to the instruction set architecture (custom instruction) of a Nios II embedded processor running at 50 MHz on an Altera Cyclone III FPGA. The scheduler supports up to 256 tasks and can be configured to use EDF or fixed priority based scheduling algorithm such as RMS. A software test bench was built to measure the overhead of the scheduler for different task sets and timer resolutions. An Earliest Deadline First (EDF) scheduler was deployed to measure the impact of running a dynamic scheduling algorithm on the processor. EDF is a dynamic priority-based scheduling algorithm in which higher priorities are assigned to the tasks with closer absolute deadlines. A software EDF scheduler implementation was used to characterize the runtime overhead involved in implementing a dynamic scheduling algorithm and to compare against our hardware implementation.

5. Results and Analysis

The overhead of the scheduler was measured for different sets of tasks and timer tick resolutions. Figure 7 shows the percentage overhead of the software scheduler. As evident in Figure 7, the scheduler overhead increases with the number of tasks and the timer-tick resolution. For a timer tick resolution of 0.1ms and with 256 tasks, the processor overhead reaches up to 18%. This would limit the amount of time available for the CPU and may cause tasks to miss deadlines. Most of this overhead results from time tick processing where the scheduler periodically processes interrupt requests to check for new tasks and managing the task queues. This has been a limiting factor for implementing dynamic priority based scheduling algorithms in embedded real time systems.

Figure 8 shows the scheduling overhead when the hardware scheduler is used. The results show that when the timer tick resolution is set to 0.1ms and with 256 tasks the scheduler overhead is less than 0.5%. This shows a 97% reduction in scheduler overhead as compared to the software model. Most of the scheduling overhead is eliminated by the hardware scheduler, as the time tick processing and a majority of the scheduling functionality is migrated to hardware. A call to the software scheduler is replaced by a custom instruction call to obtain the next task for execution or to preempt the current task. The predictability of the scheduler can be measured as the variation in the execution time of a single call to the scheduler. The best, average and worst case execution times of the scheduler are shown in Figure 9. The difference between the best case and worst case execution time is large in the software scheduler. Hence the scheduler can be a significant source of unpredictability in real time systems. The system then must be designed for the worst case behavior to ensure task deadlines are not missed, which would cause the CPU to be underutilized most of the time. On the other hand, the execution times of the hardware scheduler show more deterministic behavior with very little variation, which results in tighter worst-case execution time bounds.

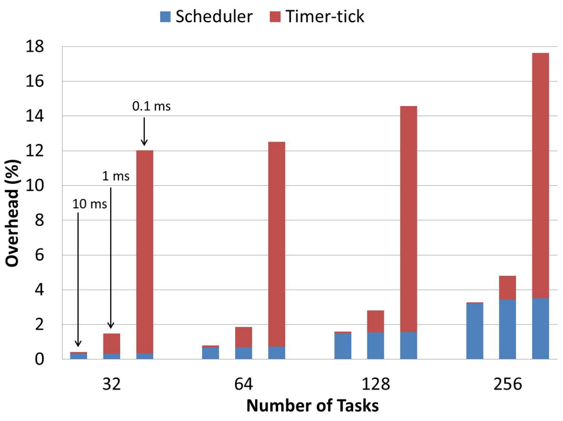


Figure 7: Software scheduler overhead.

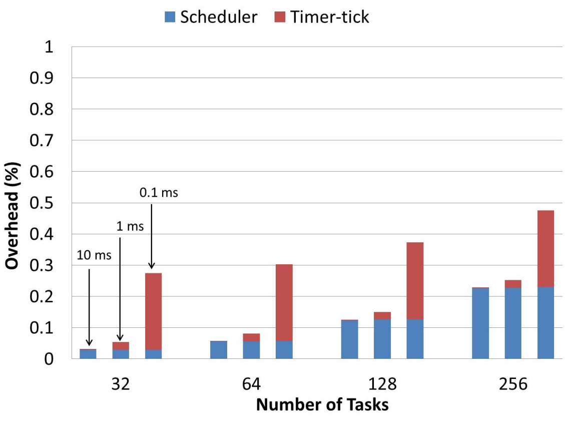
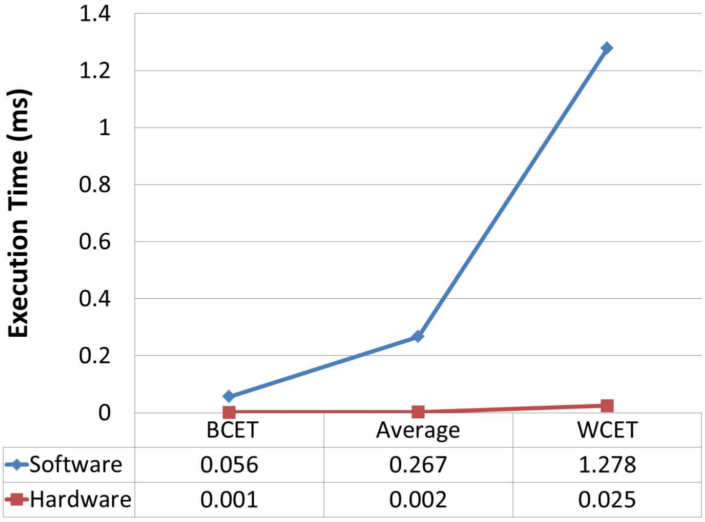


Figure 8: Hardware scheduler overhead.



	BCET	Average	WCET
Software	0.056	0.267	1.278
Hardware	0.001	0.002	0.025

Figure 9: Variation in scheduler execution time.

6. Adaptable Binary Search Trees

Although this paper is primarily concerned with deployment of data structure functionality in hardware to achieve real-time properties, we report here on our recent efforts to achieve predictability by using multiple implementations of an abstract data type in software alone. The problem we address is that of maintaining an *ordered set* using a *binary search tree* (BST). A BST organizes the elements of a set as follows. At each node n, all elements ordered less than n's value are stored in n's left subtree, and all elements ordered greater than n's value are stored in n's right subtree. Thus, an in-order traversal of the tree produces a listing of the set's elements in ascending order.

If a BST is balanced, then all single-node operations are bounded by $O(\log n)$ time, which is the height of a balanced tree of n nodes. The shape of a BST depends on the order in which elements are inserted and deleted from the ordered set. Without care, a BST can become *un*balanced: the most unbalanced tree behaves as a linked list, with all single-node operations taking $O(n)$ time. For example, such a tree results from inserting n elements in ascending order.

Self-balancing BSTs are therefore an important data structure for real-time systems, and we consider here two such implementations: AVL trees and Red-Black trees. The following table summarizes the worst-case behaviors of interest for these implementations:

Worst case	AVL	Red-Black
Rotations for insert	2	2
Rotations for delete	$\log n$	3
Height for n nodes	$< 1.44 \log(n+2) - 1$	$\leq 2 \log(n) + 1$

Because of the height bounds, both implementations achieve an $O \log(n)$ time bound to find a node in the tree. While the bounds are asymptotically the same, Red-Black trees are essentially as unbalanced as possible while maintaining their bounds, while AVL trees are as balanced as possible. The difference in subtree height at any Red-Black tree node n can be off by a factor of 2, while the subtree heights for any AVL tree node n differ by at most 1.

While asymptotically irrelevant, these differences can be considerable for real-time applications, and neither implementation is preferable in all situations to the other. As shown in the table above, AVL trees maintain their better lookup performance by performing at most $\log(n)$ rotations in response to changes in the BST. On the other hand, changes to a Red-Black tree preciptate at most 3 rotations, but lookup times could differ by a factor of 2.

We seek an implementation that can dynamically change its behavior between the two implementations in response to DDDAS considerations. Our work thus far has concerned the cost of converting one implementation to the other. We report here on a new technique for converting an AVL tree to a Red-Black tree. One approach is simply to traverse the tree and establish the color at each node. This has been considered by Glick [13], and while that algorithm requires no rotations to establish the Red-Black tree, a traversal of the entire tree is required.

For real-time systems, an operation that must traverse the entire tree is signicantly more expensive than all of the other operations on a BST. To avoid such expense, we observe the following property of establishing a Red-Black tree from an extant AVL tree. The color at a given node can be determined by the height of a node and the height of its parent in the BST. For AVL trees that include such height information, establishing Red-Black coloring can be accomplished incrementally as operations are performed on the BST. Information the algorithm uses to color a node is its height and the height of its parent. To color a node n, three cases must be considered. The first is if the parent of n has even height. In this case we simply color n black. The next two cases occur when the parent of n has odd height. If n has even height it must be colored red, if odd it is colored black. It follows from this construction that it is impossible for both a node and its parent to be colored red.

In a DDDAS system, we can observe the types of functions that are called on the BST and thus predict sections of the tree that may be more active than others. The tree can be converted from AVL to Red-Black according to these predictions. For instance, if the system anticipates a search-heavy section of operations, the system will convert the tree to AVL for faster searches. When the system anticipates new elements will be added and deleted from the tree, it can convert the tree to Red-Black.

Our approach to coloring nodes incrementally creates an intriguing idea of a *hybrid* tree that contains some AVL sections as well as Red-Black sections. This could allow sections of the tree that are being searched often to remain AVL for quicker searches, while other sections are Red-Black.

7. Conclusion

A scalable hardware scheduler has been implemented that supports 256 tasks and can be configured to run one of three (EDF, RMS, other fixed-priority) scheduling disciplines. A new hardware priority queue architecture is implemented that supports enqueue and peek operations in $O(1)$ time, returns the top-priority task in $O(1)$ time, and completes a dequeue operation in $O(\log n)$ time. The hardware scheduler reduced the scheduling and time tick processing overhead of the system. Our results show that the hardware scheduler has reasonably deterministic behavior with predictable execution times, which is necessary in high-performance real time systems.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under award CNS-1060337, and by the Air Force Office of Scientific Research (AFOSR) under award FA9550-11-1-0343.

References

[1] S. Friedman, N. Leidenfrost, B. C. Brodie, R. K. Cytron, Hashtables for embedded and real-time systems, in: Proceedings of the IEEE Workshop on Real-time Embedded Systems, 2001.

[2] T. R. Park, J. H. Park, W. H. Kwon, Reducing os overhead for real-time industrial controllers with adjustable timer resolution, in: Industrial Electronics. ISIE. IEEE International Symposium on, 2001, pp. 369–374 vol.1.

[3] J. Adomat, J. Furunas, L. Lindh, J. Starner, Real-time kernel in hardware rtu: a step towards deterministic and high-performance real-time systems, in: Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on, 1996, pp. 164–168.

[4] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. Stankovic, G. Wallace, C. Weems, The spring scheduling coprocessor: a scheduling accelerator, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on (1999) 38–47.

[5] P. Kuacharoen, M. A. Shalan, V. J. M. III, A configurable hardware scheduler for real-time systems, in: in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, CSREA Press, 2003, pp. 96–101.

[6] S. Saez, J. Vila, A. Crespo, A. Garcia, A hardware scheduler for complex real-time systems, in: Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on, 1999, pp. 43–48 vol.1.

[7] N. Gupta, S. Mandal, J. Malave, A. Mandal, R. Mahapatra, A hardware scheduler for real time multiprocessor system on chip, in: VLSI Design, 2010. VLSID '10. 23rd International Conference on, 2010, pp. 264–269.

[8] P. Kohout, B. Ganesh, B. Jacob, Hardware support for real-time operating systems, in: Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on, 2003, pp. 45–51.

[9] J. Stankovic, K. Ramamritham, The spring kernel: a new paradigm for real-time systems, Software, IEEE (1991) 62–72.

[10] S.-W. Moon, K. Shin, J. Rexford, Scalable hardware priority queue architectures for high-speed packet switches, in: Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE, 1997, pp. 203–212.

[11] R. Bhagwan, B. Lin, Fast and scalable priority queue architecture for high-speed network switches, in: INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, 2000, pp. 538–547 vol.2.

[12] A. Ioannou, M. Katevenis, Pipelined heap (priority queue) management for advanced scheduling in high-speed networks, Networking, IEEE/ACM Transactions on (2007) 450–461.

[13] J. Glick, How to make a red-black tree from an avl tree, `http://cseweb.ucsd.edu/classes/su05/cse100/cse100wa2.txt`.

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2013) 000–000

Procedia
Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2013

Scheduling Challenges in Mixed Critical Real-time Heterogeneous Computing Platforms

Chetan Kumar N G[a], Sudhanshu Vyas[a], Ron K. Cytron[b], Christopher D. Gill[b], Joseph Zambreno[a], Phillip H. Jones[a,1]

[a]*Iowa State University, Department of Electrical and Computer Engineering, Ames, IA*
[b]*Washington University, Computer Science and Engineering, St. Louis, MO*

Abstract

In Dynamic Data-Driven Application Systems (DDDAS), applications must dynamically adapt their behavior in response to objectives and conditions that change while deployed. Often these applications may be safety critical or tightly resource constrained, with a need for graceful degradation when introduced to unexpected conditions. This paper begins by motivating and providing a vision for a dynamically adaptable mixed critical computing platform to support DDDAS applications. We then specifically focus on the need for advancements in task models and scheduling algorithms to manage the resources of such a platform. We discuss the short comings of existing task models for capturing important attributes of our envisioned computing platform, and identify challenges that must be addressed when developing scheduling algorithms that act upon our proposed extended task model.

Keywords: mixed criticality, real-time system, hybrid computing, hardware accelerators

1. Introduction

An applications ability to respond dynamically and swiftly to new information is central to the Dynamic Data Driven Applications Systems (DDDAS) concept. To achieve this capability, run-time platforms are needed that can monitor and adapt to changing conditions, not only with respect to an applications evolving requirements, but also to the dynamics of the platform and the operating environment.

While conventional techniques have optimized such platforms for performance, a greater challenge today is to optimize the platforms ability to *anticipate strategic surprise* [1]. Success is then measured by the platforms ability to retask themselves in response to unexpected phenomena that spontaneously introduce requirements to monitor, avoid, or respond to such surprises. This challenge falls squarely in the domain of DDDAS and is especially important for assets that are in flight and therefore typically beyond the reach of physical modification.

Towards the development of such systems, our group's research aims to develop a prototype execution framework through which application and environment data are streamed at run-time, and which can adapt dynamically

Email addresses: `ckng@iastate.edu` (Chetan Kumar N G), `spvyas@iastate.edu` (Sudhanshu Vyas), `cytron@cse.wustl.edu` (Ron K. Cytron), `cdgill@cse.wustl.edu` (Christopher D. Gill), `zambreno@iastate.edu` (Joseph Zambreno), `phjones@iastate.edu` (Phillip H. Jones)

[1] Corresponding author

to maintain essential system properties, and to optimize the collection, analysis, and application of the data. In addition to adapting its own behavior to the data flowing through it, the execution framework will shape and optimize the flows of data themselves, to integrate multiple concerns of the DDDAS platform and its applications.

Illustrative example. Consider an autonomous helicopter operating within specified mission parameters, such as the requirement to acquire and maintain surveillance of particular ground vehicles. The data streams that flow through the system may include: 1) data from sensors that capture the state of the environment in which the helicopter is operating, 2) data from sensors that capture the physical state of the helicopter, 3) monitors that assess and track the health and performance of power and computational electronics, and 4) monitors that track and quantify how well the helicopter is performing tasks specific to the mission at hand.

However, since the very nature of surprise precludes its precise characterization *a priori*, a significantly more comprehensive and fundamental shift in how the resources of the mission platform may be retasked to address surprise is needed. In response to unexpected adverse conditions, tactical opportunities, or in-mission re-prioritization of objectives, the sensors, computational resources, and flight-control systems may require different combinations of coordination with respect both to their individual behavioral requirements and to cross-cutting constraints on overall system properties. For example, the filters that select and transform data may require reprogramming and reconfiguration so that an unexpected phenomenon of interest can be monitored, computational resources may require reallocation to ensure timely extraction of mission-relevant information from that data, and flight control parameters may require adaptation to keep the helicopter oriented to best observe the phenomenon while it persists.

To illustrate how data streaming through a DDDAS execution framework could dynamically optimize performance in the helicopter example, consider how an on-board camera could be used to track an object entering its field of vision. To ensure both overall image quality and the accuracy of object identification and tracking in particular, it is necessary to ensure that the helicopter's flight control algorithm keeps the helicopter reasonably steady. Such a control function would be designed to meet a particular set of specifications (*e.g.,* hold the helicopter within k meters of position (x, y, z), with an angle of deviation from level of no more than θ).

Although this kind of specification and applicable control theory are both well understood, in practice there are a number of dynamic factors that can impact both the effectiveness and precision of such control. For example, coaxial electric helicopters may suffer blade strikes that can cause a part of the main rotor blade to chip. Even a small chip on a rotor blade in turn can have a significant impact on flight dynamics for which the vehicle's flight controllers, platform resources, and applications may need to compensate.

Furthermore, once it has been determined that the helicopter has a damaged blade, to maintain both control of the vehicle and awareness of its condition, the execution framework may need to modify how sensor data is streamed through the system. For example, when the helicopter is flying smoothly with undamaged blades, using a light-weight filter for the accelerometer sensor data may be sufficient (*e.g.,* computing a weighted running average). However, in the presence of heightened vibrations a more computationally expensive filter that fuses data from accelerometer and gyroscope sensors maybe needed (*e.g.,* a Kalman filter).

Limitations of the current state of the art. At issue in the example above is the extent to which such diverse modifications to a mission can be characterized and the relevant hardware and software reconfigured to meet its constraints and accomplish its goals reliably, under a wide range of such possible adaptations. Both the original and modified missions would typically contain elements that classically would be called embedded (small footprint), real-time (must meet deadlines and have low latency), and high-performance (best possible throughput, data fidelity, and computational resolution). While it may have been possible pre-flight to analyze the original mission with respect to the platform's ability to operate stably and meet its requirements, relatively little time may be available to determine the suitability of the platform for its modified mission once a phenomenon of interest appears.

While the DDDAS paradigm appears well suited for addressing a number of aspects of the previously given example, traditional system design tools and methods treat disjointly the behaviors of individual hardware and software components, and the many system properties that cross-cut them, as Figure 1 illustrates. Traditional layers of abstraction also tend to isolate application-level concerns from hardware- and physical-level concerns.

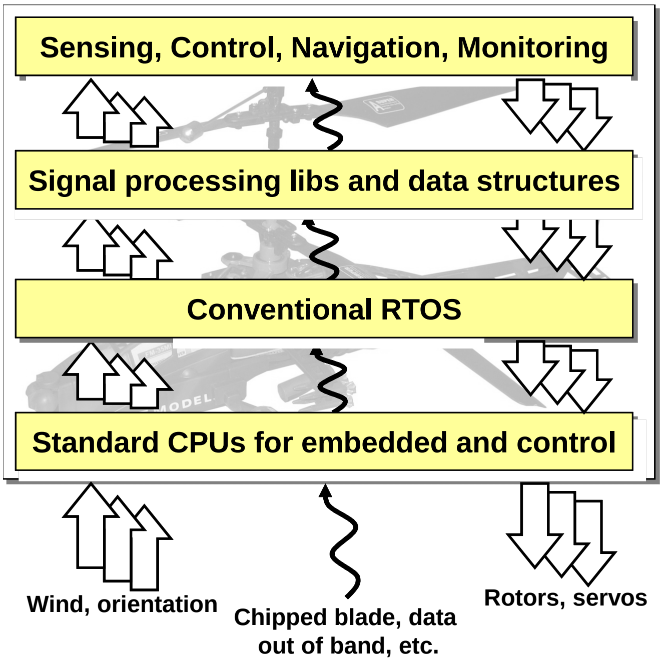

Sensing, Control, Navigation, Monitoring
Signal processing libs and data structures
Conventional RTOS
Standard CPUs for embedded and control
Wind, orientation
Chipped blade, data out of band, etc.
Rotors, servos

Fig. 1: Traditional layered architecture for helicopter example.

While this can help developers of traditional applications manage system complexity, such separation comes at a cost of less visibility and control over the interactions among the hardware and software components.

Grand Vision. The current state of the art in system software and hardware platforms poses crucial impediments to realizing the full potential of the DDDAS paradigm, and motivates an ambitious reconsideration of how hardware and software co-design can enable coordinated adaptive re-configuration while a mission is in progress.

The grand vision of our work is to move toward producing a computing platform that seamlessly conjoins system properties that are typically treated disjointedly (*e.g.,* real-time schedulability, and feedback from internal and external sensors). If successful, this holistic approach to cross-cutting system properties and the exploration of underlying enhancements to compilers, middleware, and computer architecture would reinforce and enhance DDDAS ability to optimize system performance through the interplay of streamed data and dynamic execution.

Figure 2 depicts the high-level vision we have for weaving system-level properties and concerns throughout the system stack. A significantly more dynamic treatment of system behavior and properties is enabled by our envisioned execution framework, which orchestrates the migration of functionality and sharing of information across computing layers, for the purpose of increased system efficiency and robustness. Middleware is tuned to application needs through dynamic data structure adaptation and a property-aware scheduler, while hardware resources are retasked under the direction of middleware to best serve system demands.

The system stack will support data structures whose properties are dynamically adapted in response to streamed data to make performance trade-offs at the system level (*e.g.,* timing jitter vs. memory footprint vs. throughput vs. thermal and power concerns). Continuing on this front, middleware and compiler mechanisms will support aspect-based weaving of system properties into these data structures. Additionally, low-level hardware-based system monitors and instruction set architectures will support low-latency dynamic adaptation to changes in streamed data from sensors and hardware monitors, thus integrating both hardware and software layers of an overall DDDAS architecture.

New task models and scheduling algorithms. This paper focuses on the need for advancements in task models and scheduling algorithms to support our envisioned platform's adaptation to *strategic surprise*. These models

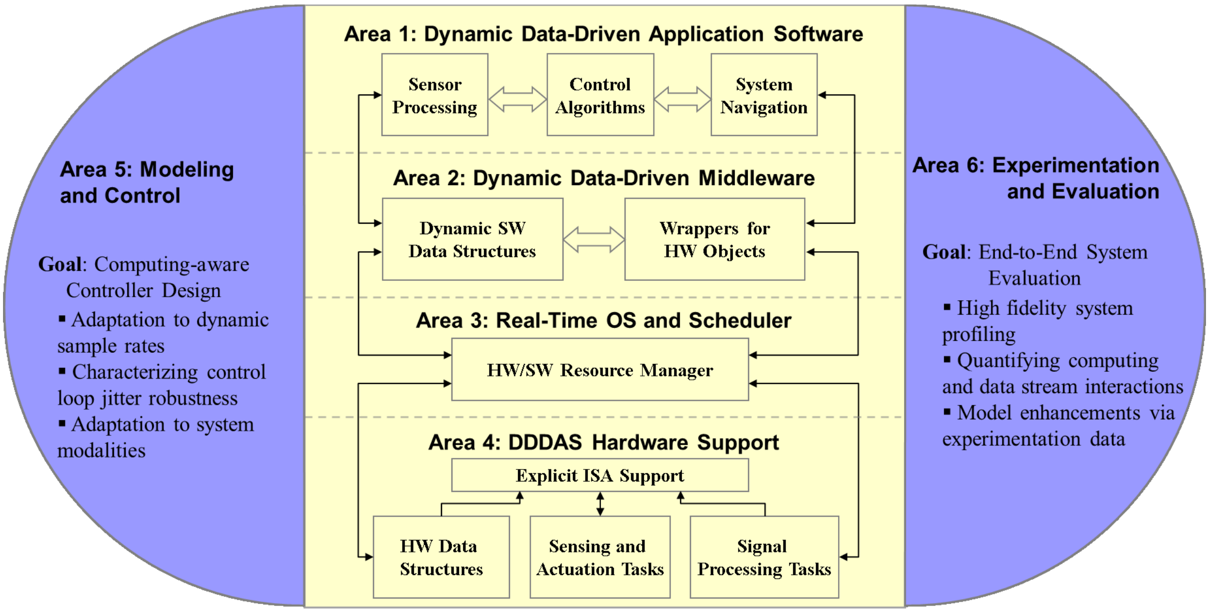

Area 1: Dynamic Data-Driven Application Software
Sensor Processing
Control Algorithms
System Navigation
Area 2: Dynamic Data-Driven Middleware
Dynamic SW Data Structures
Wrappers for HW Objects
Area 3: Real-Time OS and Scheduler
HW/SW Resource Manager
Area 4: DDDAS Hardware Support
Explicit ISA Support
HW Data Structures
Sensing and Actuation Tasks
Signal Processing Tasks
Area 5: Modeling and Control
Goal: Computing-aware Controller Design
▪ Adaptation to dynamic sample rates
▪ Characterizing control loop jitter robustness
▪ Adaptation to system modalities
Area 6: Experimentation and Evaluation
Goal: End-to-End System Evaluation
▪ High fidelity system profiling
▪ Quantifying computing and data stream interactions
▪ Model enhancements via experimentation data

Fig. 2: Integrated hardware/software architecture for adaptive reconfigurable execution.

and algorithms will form the heart of our platform's "Schedule" and "HW/SW Resource Manager". They must capture the dynamics of mission mode changes, the existence of heterogeneous computing resources that are shared among many tasks, and allow for graceful degradation under overload conditions. The remainder of this paper discusses: 1) short comings of existing models for developing schedulers for such mixed critical real-time heterogeneous computing platforms, and 2) challenges that must be addressed by algorithms that are applied to our proposed extended task model.

2. Related Work

To address the issue of scheduling in mixed criticality systems, two alternative task models have been proposed by Vestal et al. [2, 3] and de Niz et al. [4, 5]. In Vestal's multi-criticality task model, each task τ_i is assigned a criticality level L_i and may have alternative worst case execution times (WCET), $C_i(l)$, corresponding to different criticality levels. The higher the criticality level, the more conservative will be the WCET estimation. Vestal et al. suggested the use of Audesly's prioirity assignment scheme [6] and period transformation technique [7] to improve the schedulability and utilization of mixed criticality tasks. Many scheduling models and algorithms [8, 9, 10, 11, 12] have been proposed based on Vestal's model to improve the schedulability of certifiable mixed criticality tasks. Effectiveness of reservation-based and priority-based scheduling approaches to dual-criticality systems were studied in [8, 11] by using processor speed-up factor as a metric. PLRS, a scheduling algorithm for certifiable mixed criticality sporadic task systems is presented in [9] and an offline computation method was provided to check the schedulability of the task set. Criticality based earliest deadline first (CBEDF) algorithm was presented in [10] to schedule tasks on dual-criticality systems. Earliest Deadline First with Virtual Deadlines (EDF-VD) scheduling algorithm was proposed in [12] for scheduling of mixed-criticality implicit-deadline sporadic tasks on preemptive uniprocessors.

In de Niz's task model, each task τ_i can have two execution times: C_i - worst case execution time under normal conditions and C_i^o - overload execution budget. Each task is assigned a criticality level L_i. Based on this task model, a zero slack scheduling method was proposed by de Niz et al. [4], which works on top of any priority based scheduling algorithm. Each task can be executed in either normal or critical mode. The tasks in normal mode are scheduled based on their priority to maximize resource utilization. When executing in critical mode, all the lower criticality tasks are suspended to guarantee the execution of higher criticality tasks. A metric for overload-resilience called ductility was developed and Compress-On-Overload Packing (COP), an algorithm which works on top of zero slack rate monotonic scheduler to maximize ductility in distributed mixed-criticality systems was presented in [5].

The scheduling algorithms discussed above do not apply when tasks share mutually exclusive resources. Lakshmanan et al. [13] presented extensions to priority inheritance and ceiling protocols for zero slack scheduling [4] to solve the task synchronization problem in mixed criticality systems. A two tier dynamic resource management framework for mixed criticality embedded systems is presented in [14]. The framework provides static resource guarantees and enables fault isolation for distributed application subsystems with mixed criticality requirements and facilitates certification of safety-critical applications. A software based memory throttling mechanism is presented in [15], which controls the memory interference and guarantees the schedulability of critical tasks in mixed criticality real-time systems.

3. Mixed Critical Real-time Heterogeneous Computing Platforms: Proposed Task Model

Mixed critical real-time heterogeneous systems execute under varied operating conditions. Identifying different system configurations which is representative of all the scenarios and operating conditions would be challenging if not unrealistic. However, we believe it is possible to identify some of the key configurations or modes of operation during the design phase and transitions between these modes could be tested and certified. We call these as stable states of execution. As an example let us consider an autonomous helicopter with a mission to acquire and maintain surveillance of a ground vehicle. The stable states of execution and the occurrence of "surprise" during a representative autonomous helicopter mission is illustrated in Figure 3. The autonomous aircraft needs to minimize the consequences of these surprise situations. This could be achieved by dynamically changing the application characteristics to maintain the stability of the system. A task model should capture these dynamic changes in operating conditions to support the platform's ability to anticipate strategic surprise.

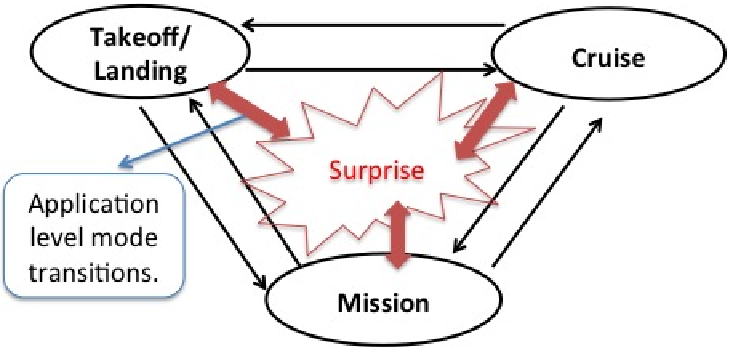


Fig. 3: Illustration of a "surprise" occurring during a representative autonomous helicopter mission.

The following are some of the main properties of mixed critical heterogeneous computing platforms which are not captured in the existing mixed criticality task models:

Dynamically changing task criticality. Existing mixed criticality models assume, the criticality level of a task to be constant. Some tasks may be critical only during certain operating conditions. Assuming the task to be critical all the time may lead to under utilization of resources as more conservative WCETs need to considered for schedulability analysis. A better utilization of resources could be achieved by varying the criticality of the tasks based on operating conditions.

Required and Optional Resources. Conventional resources, such as data structures and files, are considered as required resources. Some resources could be optional and its availability may increase application performance, predictability and/or improve quality of service. Reconfigurable hardware accelerators are an example of an optional resource. The availability of these optional resources enable the scheduler to optimize resource sharing for different system attributes such as stability, utilization and/or other utility functions.

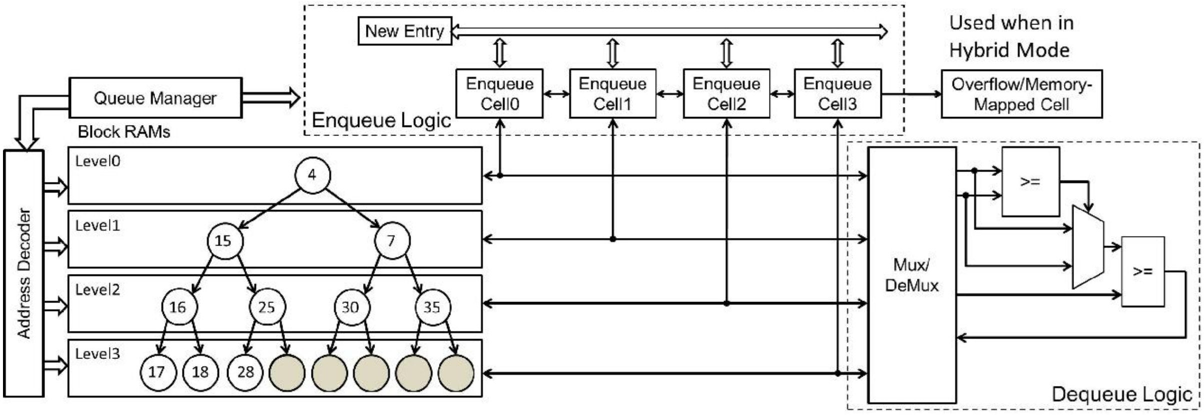


Fig. 4: Hardware priority queue architecture. As queue data is stored in hardware, allocating the hardware priority queue to a different task will incur considerable overhead. This should be accounted for during schedulability analysis.

Resource Preemption Overhead. Generally, overhead incurred due to preemption of resources is not accounted for (e.g. placing a semaphore on a data structure.). Some resources (e.g. hardware accelerators) may have large preemption overhead and this needs to be accounted for accurate schedulability analysis. For example, consider the hardware priority queue described in [16]. A high-level hardware architecture diagram of the priority queue is shown in Figure 4. For example, lets say that the hardware priority queue is being used by the scheduler. The queue data will be stored in hardware as shown in Figure 4. Now to allocate the queue to a another application (e.g. network bandwidth manager), the scheduler data stored in hardware needs to be copied to software memory and the hardware queue should be initialized with new data. The preemption overhead of the hardware priority queue varies depending on the size of the queue. This cannot be ignored during scheduling/resource allocation.

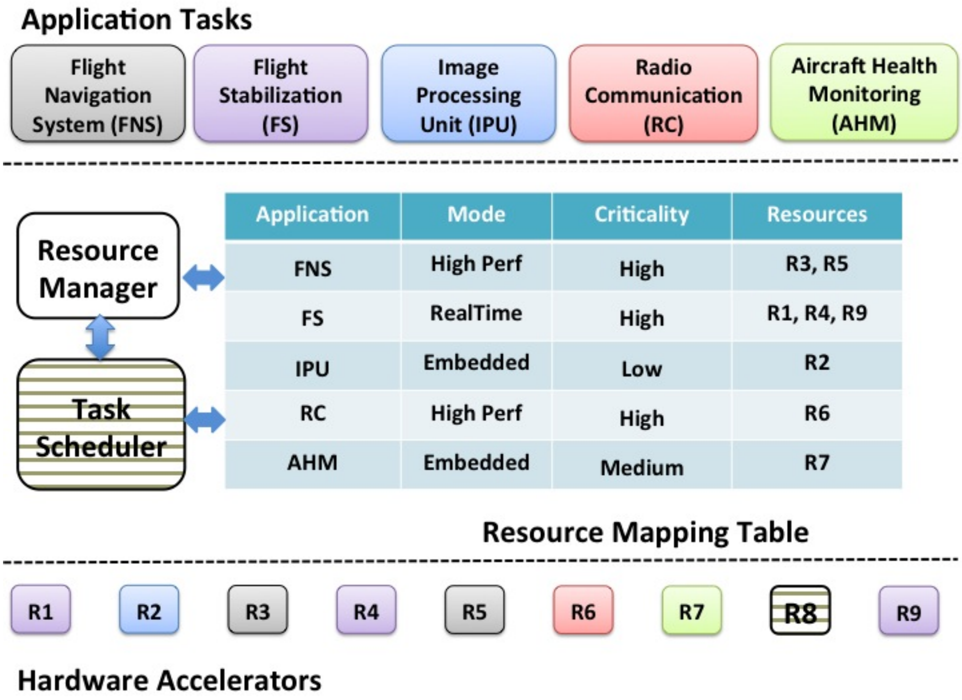
WCET dependency on resource allocated. The availability of reconfigurable logic will enable the use of hardware accelerators to improve application performance and predictability. The resources allocated and, in turn, a task's WCET can change during run time.

Proposed Task Model. Taking into consideration the properties discussed, a new task model is presented where each task, τ_i, is defined as:
$\tau_i = (T_i, A_i, D_i, P_i, V_i, RR_i[], L_i, M_i, C_i(RA_i[]))$ where,

- T_i is the task period,
- A_i is the arrival time,
- D_i is the relative deadline,
- P_i is the priority of the task,
- $U_i(t)$ is the task utility function, where t is the time elapsed since its arrival.
- $RR_i[]$ is resource requirement vector, which has the list of required and optional resources.
- L_i is the task criticality level,
- M_i is modality,
- $C_i(RA_i[])$ is the worst case execution time, which depends on resources currently allocated, $RA_i[]$, to the task.

Each resource R_x is defined as: $R_x = (n_x, t_x, sc_x, d_x[])$ where,

- n_x is the count of the available resources R_x,
- t_x is the resource type (Blocking/Non-blocking),
- sc_x is the worst case switching cost associated with the resource,



Application	Mode	Criticality	Resources
FNS	High Perf	High	R3, R5
FS	RealTime	High	R1, R4, R9
IPU	Embedded	Low	R2
RC	High Perf	High	R6
AHM	Embedded	Medium	R7

Fig. 5: Conceptual architecture of a resource manager and scheduler for the envisioned adaptive mixed criticality system.

- $d_x[]$ is the resource dependency vector.

One of our end goals is to develop a real-time resource manager and scheduler as part of the adaptive computational stack for heterogeneous mixed critical real-time systems, as conceptually illustrated in Figure 5. The hardware accelerators are dedicated hardware modules that are used to accelerate a certain functionality or operation. The applications make use of these hardware components to improve application throughput. Each application task is associated with a criticality level, which changes the resource requirements and parameters of the application. The criticality of applications may change during run-time, which is triggered by dynamically changing operating conditions (environment). The resource mapping table stores the task parameters, status and resource requirements of all applications. This is used by 1) the resource manager for the allocation and management of hardware resources, and 2) the task scheduler, which takes takes into account the criticality of the applications and generates schedules accordingly to ensure that the tasks are completed within their deadline. As the application parameters and resource requirements change during run-time, there is a need for a parametric-based on-line scheduling approach.

There are numerous challenges in scheduling and resource management for the envisioned adaptive mixed critical real-time systems. Some of the key challenges are:

- Identifying the stable states of execution and guaranteeing system stability during state transitions.
- Schedulability analysis: Scheduling space expands rapidly when there are n modes and n^2 transitions. The WCET of a task depends on the resources allocated and for n optional resources, there can be 2^n possible resource combinations for each task. Mitigating or overcoming this potentially overwhelming search space size is a key scheduling challenge.
- Defining the semantics of hardware accelerators.
- Calculating the speed-up factor for task using a hardware accelerator, if it is data dependent.

4. Conclusion

A vision for an adaptable computing platform to support DDDAS applications operating under unexpected conditions (i.e. *strategic surprise*) was introduced. We discussed the limitations of existing mixed criticality task models, which does not fully capture the dynamics of mixed-critical heterogeneous computing platforms and proposed an extended task model. We briefly discussed the challenges associated with developing scheduling and resource management algorithms for such a platform. These challenges are starting points for rich areas of continued research.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under award CNS-1060337, and by the Air Force Office of Scientific Research (AFOSR) under award FA9550-11-1-0343.

References

[1] P. Lee, Colloquium presented at the University of Washington, `http://www.youtube.com/watch?v=lLYobdHCCEo`.

[2] S. Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International, 2007, pp. 239 –243.

[3] S. Baruah, S. Vestal, Schedulability analysis of sporadic tasks with multiple criticality specifications, in: Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on, 2008.

[4] D. de Niz, K. Lakshmanan, R. Rajkumar, On the scheduling of mixed-criticality real-time task sets, in: Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, 2009.

[5] K. Lakshmanan, D. de Niz, R. Rajkumar, G. Moreno, Resource allocation in distributed mixed-criticality cyber-physical systems, in: Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on, 2010.

[6] N. Audsley, Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, Citeseer, 1991.

[7] L. Sha, J. Lehoczky, R. Rajkumar, Solutions for some practical problems in prioritized preemptive scheduling, in: IEEE Real-Time Systems Symposium, 1986, pp. 181–191.

[8] S. Baruah, H. Li, L. Stougie, Towards the design of certifiable mixed-criticality systems, in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE, 2010, pp. 13 –22.

[9] N. Guan, P. Ekberg, M. Stigge, W. Yi, Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems, in: Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd, 2011, pp. 13 –23.

[10] T. Park, S. Kim, Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems, in: Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on, 2011, pp. 253 –262.

[11] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, L. Stougie, Scheduling real-time mixed-criticality jobs, Computers, IEEE Transactions on 61 (8) (2012) 1140 –1152.

[12] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, L. Stougie, The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems, in: Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on, 2012, pp. 145 –154.

[13] K. Lakshmanan, D. de Niz, R. Rajkumar, Mixed-criticality task synchronization in zero-slack scheduling, in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE, 2011.

[14] B. Huber, C. El Salloum, R. Obermaisser, A resource management framework for mixed-criticality embedded systems, in: Industrial Electronics, 2008. IECON 2008. 34th Annual Conference of IEEE, 2008.

[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Memory access control in multiprocessor for real-time systems with mixed criticality, in: Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on, 2012.

[16] C. Kumar, S. Vyas, J. A. Shidal, R. Cytron, C. Gill, J. Zambreno, P. H. Jones, Improving system predictability and performance via hardware accelerated data structures, Procedia Computer Science 9 (0) (2012) 1197 – 1205, proceedings of the International Conference on Computational Science, ICCS 2012.
URL `http://www.sciencedirect.com/science/article/pii/S1877050912002505`

An FPGA-Based Plant-on-Chip Platform for Cyber-Physical System Analysis

Sudhanshu Vyas, *Student Member, IEEE*, Chetan Kumar N. G., Joseph Zambreno, *Senior Member, IEEE*, Chris Gill, *Senior Member, IEEE*, Ron Cytron, *Senior Member, IEEE*, and Phillip Jones, *Member, IEEE*

***Abstract*—Digital control systems are traditionally designed independent of their implementation platform, assuming constant sensor sampling rates and processor response times. Applications are deployed to processors that are shared amongst control and noncontrol tasks, to maximize resource utilization. This potentially overlooks that computing mechanisms meant for improving average CPU usage, such as cache, interrupts, and task management through schedulers, contribute to nondeterministic interference between tasks. This response time jitter can result in reduced system stability, motivating further study by both the controls and computing communities to maximize CPU utilization, while maintaining physical system stability needs. In this letter, we describe an field-programmable gate array (FPGA)-based embedded software platform coupled with a hardware plant emulator (as opposed to purely software-based simulations or hardware-in-the-loop setups) that forms a basis for safe and accurate analysis of cyber-physical systems. We model and analyze an inverted pendulum to demonstrate that our setup can provide a significantly more accurate representation of a real system.**

***Index Terms*—Cyber-physical systems, embedded systems, field-programmable gate aray (FPGA), hardware emulation, plant-on-chip.**

I. Introduction

EMBEDDED systems and digital control theory have independently developed into mature fields, despite the clear connection between controllers and embedded platforms. Initially, each digital control loop was implemented on a dedicated processor, thus maintaining a *separation of concerns*. The demand for tighter system integration and the use of economical commercial-off-the-shelf products has blurred this separation [1]. In modern systems, the tasks running on the processor unknowingly compete for processor resources. These resources, meant to improve average resource usage for nonreal-time systems, are becoming sources of nondeterministic computation time or *computation jitter*. Example causes include interrupts [1], branch misprediction [8], cache misses [11], and task management through operating systems [12]. These features limit the degree to which time invariance can be guaranteed, and cause systems to break control engineers' key assumption of constant sample rates and processor response time [2]. Ultimately, control loop robustness is greatly affected by this transition from a dedicated processor system to an environment of tasks competing for resources [5]. Thus, a more holistic view is now needed to develop and deploy controllers that take into account cyber-architecture artifacts on a system's physical stability.

As a motivating example, Fig. 1 shows the timing response of an inverted pendulum model as we vary the computational delay (the time between receiving a sensor sample and sending the response), while holding sensor sample rate constant. In Fig. 1(a), a controller computing delay that is 15% of the state sampling rate has negligible impact on the system's stability. As the delay increases to 65% of the sample period [see Fig. 1(b)], some ringing in the control signal becomes apparent. Progressing to a delay of 85% of the sample period [see Fig. 1(c)] causes the plant to become less stable with oscillations that are now more pronounced. It is interesting to note that the state of the plant (i.e., cart position and pendulum angle) still appears stable. A further increase in the computational delay [see Fig. 1(d)] leads to loss of controller stability resulting in an eventual fall for the pendulum.

Previous work has identified jitter in cyber-physical systems (CPS) as a significant research challenge. The authors in [11] worked on characterizing Linux for real-time applications and found that the sources of jitter were implicit to the processor and were not completely correctable through software. A detailed analysis of branch-prediction schemes [8] concludes that static branching schemes work better for real-time systems than dynamic branch prediction. In [4], the authors compare several scheduling methods and concluded that deadline advancement was the most consistent, with minimal degradation in performance of controllers as the number of tasks increased and had relatively consistent low jitter. Controls experts are developing toolflows, like TrueTime-JitterBug, to evaluate the impact of a controller's response-time jitter on closed-loop stability [5]. In [6], [9] the authors have developed a set of stability criteria for closed-loop systems in which the sample rate contains jitter. In [7], a quantitative metric similar to the concept of phase margin is proposed, called jitter margin, which is the upper-bound of delay that a control loop can tolerate before going unstable. In an approach closely related to ours, the delay and period of control loops are used in a cost function, which is then treated as a minimization problem [3], and later a convex optimization problem

Manuscript received March 27, 2013; accepted May 04, 2013. Date of publication May 30, 2013; date of current version February 24, 2014. This work is supported in part by the National Science Foundation (NSF) under Award CNS-1060337, and by the Air Force Office of Scientific Research (AFOSR) under Award FA9550-11-1-0343. This manuscript was recommended for publication by Ramesh S.

S. Vyas, C. Kumar N. G., J. Zambreno, and P. Jones are with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011 USA (e-mail: spvyas@iastate.edu; ckng@iastate.edu; zambreno@iastate.edu; phjones@iastate.edu).

C. Gill and R. Cytron are with the Department of Computer Science and Engineering, Washington University, Saint Louis, MO 63130 USA (email: cdgill@wustl.edu; cytron@wustl.edu).

Digital Object Identifier 10.1109/LES.2013.2262107

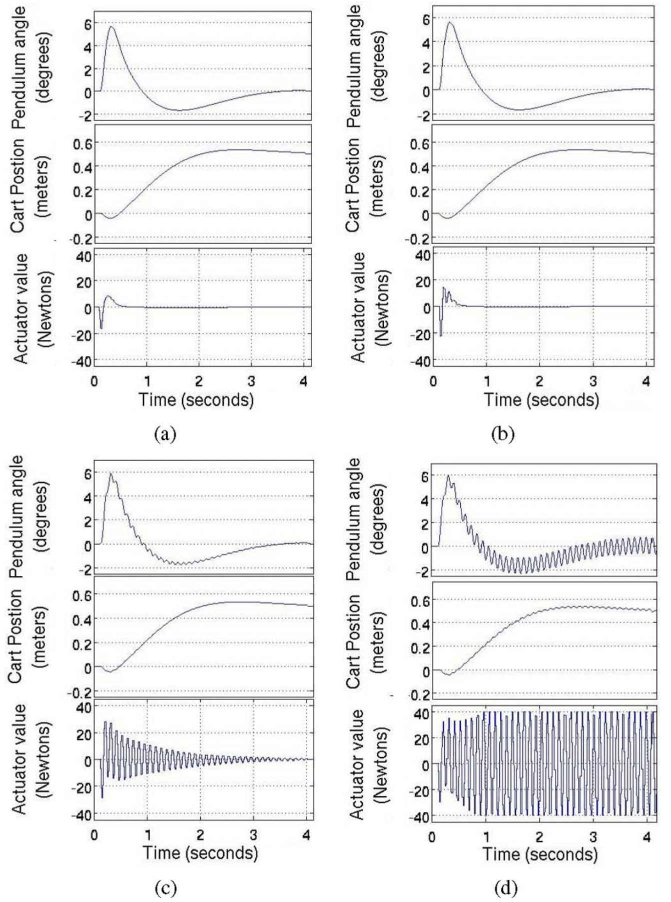


Fig. 1. Effect of computational delay on a digital control system. With the sample period fixed at 15 ms, the delay is varied from 15% (a) to 90% (d). At 65%, a ringing begins to appear in (b), which becomes more pronounced at 85% (c). Finally, at 90% (d), the plant remains stable while the controller continuously oscillates.

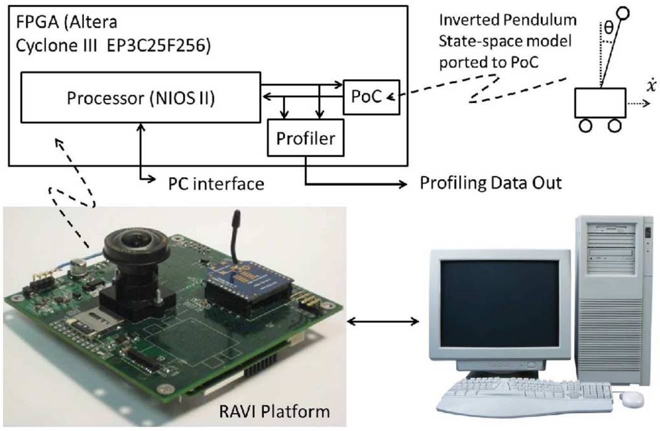

FPGA (Altera Cyclone III EP3C25F256)
Processor (NIOS II)
PoC
Profiler
Inverted Pendulum State-space model ported to PoC
θ
ẋ
PC interface
Profiling Data Out
RAVI Platform

Fig. 2. Our experimental setup implemented on our in-house reconfigurable platform, RAVI. Note θ and $\dot{x}$ of the plant model.

[13]. A limitation of many of the previous approaches is their reliance on analytical tools and simulations of CPS which mask the jitter caused by hardware architectures.

In contrast, this letter presents the design and implementation of a control systems emulation framework that couples plant emulation hardware with an embedded processor, together on a field-programmable gate array (FPGA)-based platform. This hardware/software framework allows us to more accurately study the interaction between an actual processor and a plant-on-chip (PoC). Our experimental results, using a state-space model of an inverted pendulum as captured in the PoC hardware, indicate that this proposed framework both safely and accurately captures the nondeterministic effects of modern processor architecture on a physical plant. Since the setup uses the same interfaces that the actual system would use, once the PoC is replaced by the real plant, the input and output jitter from sampling and actuating are already accounted for in the platform. The PoC could be integrated via on-chip or off-chip networking interface to emulate plants being controlled over a network.

II. Architecture

Fig. 2 illustrates our FPGA-based infrastructure for CPS analysis. The FPGA is configured to implement the three main components: 1) an embedded processor (NIOS II) with conventional architectural features that is capable of running a modern operating system (OS); 2) a custom PoC emulator that implements a given model for the system under test; and 3) a profiler module that collects appropriate performance data and reports back to a host workstation. Our in-house reconfigurable platform, the reconfigurable autonomous vehicle infrastructure (RAVI) board, is also shown in Fig. 2. This small form factor (90 grams and 3.4" × 3.4") board was specifically designed and fabricated at Iowa State University to promote the development of efficient control systems for mobile autonomous vehicles, hosting an Altera Cyclone III FPGA for deploying the computational stack, an inertial measurement unit (IMU) for monitoring physical dynamics of vehicles, and other features that enable it to support a wide range of autonomous vehicles and applications.

Our proposed dedicated hardware (see Fig. 3) emulates the state-space model of the chosen physical plant. Our example plant is an inverted pendulum from [10], where the state-vector, X consists of four variables, the pendulum's angle θ and angular rate $\dot{\theta}$, and its cart's position x and velocity $\dot{x}$. u is the input variable that comes from the controller to stabilize the plant and is stored in the "Control Input reg." The previous state of X is stored in the "Old X RAM." The feed-back matrix A and input matrix B are constants and thus stored in "A ROM" and "B ROM." The new state of X is calculated by the hardware, with the help of a finite state machine (FSM) and internal timers, as follows. u is sequentially multiplied with the "B" matrix and the result stored in "uB RAM." Next, the dot products of X with each row of A is sequentially calculated with the help of the accumulator and stored in the "AX RAM." Then, the addition of vectors Bu and Ax is performed, resulting in the new, updated state X and stored in the "Xnew RAM." The processor may sample X at any time through the "Sample Reg." A hardware interface is dedicated to nonintrusive transmission of X, u, and their respective time stamps through the "UART Reg." Other important evaluation metrics like sample-to-actuation time delay and the energy consumed by actuators are performed during post processing from the recorded data.

We require a noise source to emulate a noisy environment and test robustness in the same manner as JitterBug [5]. This

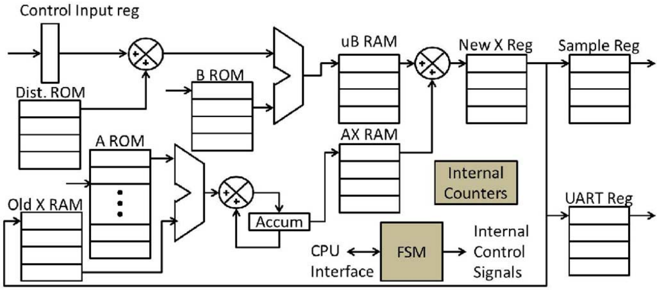


Fig. 3. Register Level architecture of the state-space based Plant-on-Chip emulator.

is implemented with the 'Dist ROM' which contains a sample array of a white-noise signal similar to JitterBug's disturbance. A value from the "Dist ROM" is periodically injected into the system by adding it to the input u before starting a state-update. This emulates an external force being exerted on the cart and can be enabled or disabled through software by an application designer.

The current hardware utilization is fairly small with 2900 LUTs, 800 flip-flops, 32 DSP blocks, and 1 K of RAM/ROM. With a 50 MHz clock source, the emulator updates its state every 100 μs, which is sufficient for emulating our example inverted pendulum plant. The advantage of our setup is that the states are periodically updated, independent of the processor controlling the hardware emulator. This eliminates the effects software simulations have on the computer they are usually running on (for example missing or late updates), especially when that computer is running the control algorithm, as well. The processor controlling this emulator cannot distinguish between the actual plant or its emulation, as the interface is unchanged and the hardware appears as an independent entity.

III. Experimental Setup and Results

In evaluating our framework, we attempted a validation of the PoC against known control system evaluation tools and standards. Control systems can be evaluated based on transient response, energy consumption, or other cost functions. These metrics correlate with the amount of effort the controller exerts to keep the system stable after receiving a change either in reference value, or when experiencing an external disturbance. We shall now refer to this metric as J. For Jitterbug, J is an "integration of square of error" [5], where error is the deviation of a designer specified variable from zero. The PoC's J is the energy (*Joules*) spent by the actuator. A secondary interest was in comparing the J's from JitterBug and the PoC. Initial experiments indicate that a JitterBug cost function of $J = \sum_{t=0}^{t=\mathrm{time}_{\mathrm{sim}}} ({e_{\dot{\theta}}}^2 + {e_x}^2 + {e_{\dot{x}}}^2)$ was closest to the amount of energy used by the actuator to keep the pendulum upright. Method 1 describes our routine for characterizing system costs. We explored the design space by varying sample period and computational delay and measured the cost in JitterBug and the energy in our setup to keep the system stable. The points where JitterBug's plots trend to infinity (equivalent to the plateau region of our setup's plots) correspond to the unstable regions of the system. To give a physical perspective, these regions correspond to our pendulum example losing balance.

Method 1: Control system cost profiling

```
for $period = 2ms \rightarrow 20ms$ do
    for $delay = 0\% \rightarrow 100\%$ do
        run simulation for $time_{sim}$;
        $J \leftarrow$calculate $energy$; /* or cost */
        $Array_J \leftarrow J$; $delay \leftarrow delay + 5\%$;
    end
    $Matrix_J \leftarrow Array_J$; $period \leftarrow period + 1ms$;
end
```

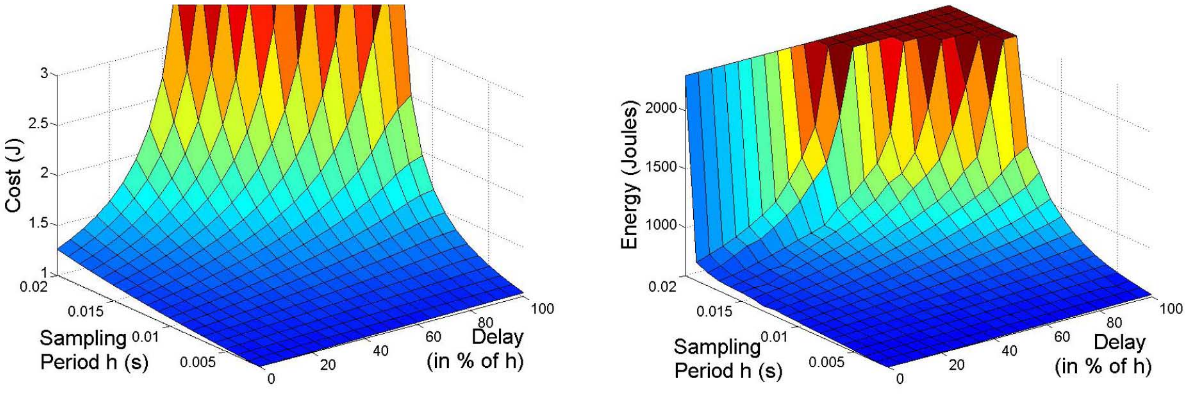
We conducted two sets of experiments. First, we attempted to maintain the pendulum cart at a fixed location, given various external disturbances. This can be done in JitterBug and in our setup. Next, we tested our setup with a step-response, which JitterBug does not permit. We performed a profiling of the relevant cost, as outlined in Method 1, and fed this data into Matlab to create the following surface plots.

Fig. 4 gives a summary of the first experiment's results. We see common trends in both setups. As we increase the computational delay from 0% of the sample period to a full sample period, the cost [see Fig. 4(a)] of keeping the system stable and the amount of energy [see Fig. 4(b)] needed by the system to keep the system stable increase in a similar fashion. Both setups show an increase in cost and energy as the sample period of the controller is increased. The region of instability is almost the same in both setups, with the PoC setup showing a slightly smaller region. An example point is where sample period is 15 ms and delay percentage is 70%. JitterBug shows that the system will be unstable whereas the PoC setup indicates that the system will be stable, but will spend more energy to maintain stability. This difference is because the pattern and magnitude of JitterBug's external disturbance is unknown and an estimated pattern is used in the PoC setup. The major difference between the setups is that JitterBug predicts that the system will be stable when the sample period is 20 ms and delay is roughly 40% or less. Since the PoC is a more realistic setup and shows that a 20 ms sample period even with no delay will be unstable, we can safely say that JitterBug's prediction is less accurate.

While analyzing our setup's step response (see Fig. 5) to different combinations of sample period and delay, we can refer back to Fig. 1 for additional clarity. Keeping the sample period fixed to 15 ms, let us observe the impact of increasing delay. At 15% delay, the system is very stable in the time response plot [see Fig. 1(a)] and is in the dark-blue plain of Fig. 5. As we increase delay, we start seeing a damp oscillation in the controller signal begin to increase in Fig. 1(b) and (c) and the energy increase and climb the cliff of the surface plot of Fig. 5. At 95%, the system is unstable [see Fig. 1(d)] and the corresponding point on the surface plot is on the plateau, further indicating instability. A JitterBug version of this test is not possible as the reference value cannot be set by a user to produce a step input.

IV. Conclusion

We presented a method for analyzing Cyber-Physical Systems using a hardware plant emulator we designed and



(a)

(b)

Fig. 4. Surface plots of cost (a) and energy (b) while injecting disturbance in cart position x. (a) JitterBug. (b) Plant-on-Chip.

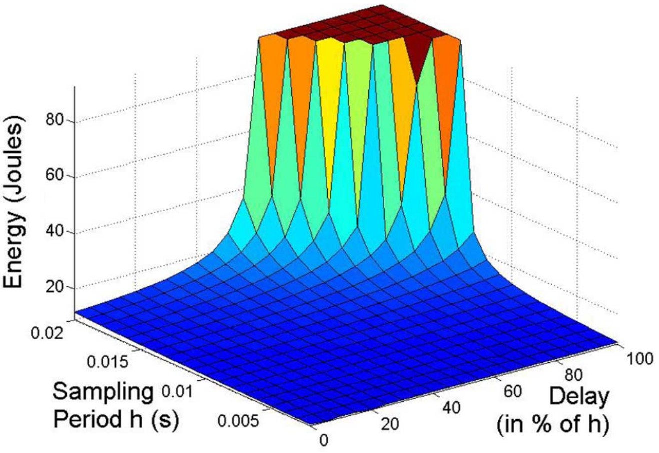


Fig. 5. Characterization plot of PoC's step-response.

integrated with an embedded processor in an FPGA-based platform. Our framework provides insight for embedded designers into how computer architecture can influence control loops. Though current simulation-based design tools provide a good approximation of a system's robustness to sample-period and delay, they work in environments and with assumptions that the delay can be modeled as a probability distribution function [4], [5]. Research in [8], [11], and [12] shows this to be not realistic and that computer elements cause nondeterministic time-varying delay and sample-period. With an actual processor under test, our setup inherently contains these nondeterministic sources of delay jitter and thus gives a more accurate result, when characterizing a system's robustness against sample period and delay variation.

In the future, we plan to control a plant-on-chip emulator while sharing processor resources with other tasks, using a real-time operating system (e.g., RT-Linux).

References

[1] K.-E. Arzen and A. Cervin, "Control and embedded computing: Survey of research directions," presented at the World Conf. Int. Federation Automic Contr. (IFAC), 2005.

[2] K. J. Astrom and B. Wittenmark, *Comput.-Controlled Syst.*, 3rd ed. Englewood Cliffs, NJ, USA: Prentice-Hall Inc., 1997.

[3] E. Bini and A. Cervin, "Delay-aware period assignment in control systems," presented at the Real-Time Syst. Symp., 2008.

[4] G. Buttazzo and A. Cervin, "Comparative assessment and evaluation of jitter control methods," presented at the Int. Conf. Real-Time Netw. Syst., 2007.

[5] A. Cervin, K.-E. Arzen, D. Henriksson, M. Lluesma, P. Balbastre, I. Ripoll, and A. Crespo, "Control loop timing analysis using Truetime and Jitterbug," presented at the Int. Conf. Contr. Appl., 2006.

[6] A. Cervin, "Stability and worst-case performance analysis of sampled-data control systems with input and output jitter," presented at the Amer. Contr. Conf., 2012.

[7] A. Cervin, B. Lincoln, J. Eker, K. Arzen, and G. Buttazzo, "The jitter margin and its application in the design of real-time control systems," presented at the Int. Conf. Real-Time Embed. Comput. Syst. Appl., 2004.

[8] J. Engblom, "Analysis of the execution time unpredictability caused by dynamic branch prediction," presented at the Real-Time Embed. Technol. Appl. Symp., 2003.

[9] H. Fujioka, "Stability analysis of systems with aperiodic sample-and-hold devices," *Automatica*, 2009.

[10] K. Ogata, *Modern Control Engineering*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2009.

[11] F. Proctor, "Timing studies of real-time linux for control," presented at the Proc. Design Eng. Techn. Conf., 2001.

[12] F. M. Proctor and W. P. Shackleford, "Real-time operating system timing jitter and its impact on motor control," presented at the SPIE Sensors Contr. Intell, Manufact. Conf,, 2001.

[13] Y. Wu, G. Buttazzo, E. Bini, and A. Cervin, "Parameter selection for real-time controllers in resource-constrained systems," *IEEE Trans. Ind. Inf.*, vol. 6, no. 4, pp. 610–620, Nov. 2010.

Hardware-software architecture for priority queue management in real-time and embedded systems

N.G. Chetan Kumar and Sudhanshu Vyas

Department of Electrical and Computer Engineering,
Iowa State University,
2215 Coover Hall, Ames, IA 50011, USA
E-mail: ckng@iastate.edu
E-mail: spvyas@iastate.edu

Ron K. Cytron and Christopher D. Gill

Department of Computer Science and Engineering,
Washington University in St. Louis,
1 Brookings Dr., St. Louis, MO 63130, USA
E-mail: cytron@cse.wustl.edu
E-mail: cdgill@cse.wustl.edu

Joseph Zambreno and Phillip H. Jones*

Department of Electrical and Computer Engineering,
Iowa State University,
2215 Coover Hall, Ames, IA 50011, USA
E-mail: zambreno@iastate.edu
E-mail: phjones@iastate.edu
*Corresponding author

Abstract: The use of hardware-based data structures for accelerating real-time and embedded system applications is limited by the scarceness of hardware resources. Being limited by the silicon area available, hardware data structures cannot scale in size as easily as their software counterparts. We assert a hardware-software co-design approach is required to elegantly overcome these limitations. In this paper, we present a hybrid priority queue architecture that includes a hardware accelerated binary heap that can also be managed in software when the queue size exceeds hardware limits. A memory mapped interface provides software with access to priority-queue structured on-chip memory, which enables quick and low overhead transitions between hardware and software management. As an application of this hybrid architecture, we present a scalable task scheduler for real-time systems that reduces scheduler processing overhead and improves timing determinism of the scheduler.

Keywords: priority queue; hardware-software co-design; real-time and embedded systems; hardware scheduler.

Reference to this paper should be made as follows: Kumar, N.G.C., Vyas, S., Cytron, R.K., Gill, C.D., Zambreno, J. and Jones, P.H. (2014) 'Hardware-software architecture for priority queue management in real-time and embedded systems', *Int. J. Embedded Systems*, Vol. 6, No. 4, pp.319–334.

Biographical notes: N.G. Chetan Kumar is a PhD student in the Department of Electrical and Computer Engineering, where he is working with Prof. Phillip Jones. He completed his BS in Electronics and Communication at Visveswaraya Technological University, Bangalore, India in 2007. His research interests include embedded and real-time systems and hardware/software co-design. His current research focuses on developing techniques to improve predictability in execution of core system operations in real-time systems, using hardware-software co-design approaches.

Sudhanshu Vyas is a graduate student pursuing his PhD at Iowa State University. He joined ISU in the Fall of 2009. Before joining, he received his BE from Birla Institute of Technology in Electronics and Communication Engineering in 2006 and worked at CG-CoreEl, an embedded systems company based in Bangalore. His research interests include reconfigurable architectures, embedded systems, control systems and FPGA fault tolerance.

Ron K. Cytron is a Professor of Computer Science and Engineering at Washington University. His research interests include optimised middleware for embedded and real-time systems, fast searching of unstructured data, hardware/runtime support for object-oriented languages, and computational political science. He has over 100 publications and ten patents. He has received the SIGPLAN Distinguished Service Award and is a co-recipient of SIGPLAN Programming Languages Achievement Award. He served as Editor-in-Chief of *ACM Transactions on Programming Languages and Systems* for six years. He participated in writing the Computer Science GRE Subject Test for eight years and chaired the effort for three years. He is a Fellow of the ACM.

Christopher D. Gill is a Professor of Computer Science and Engineering at Washington University in St. Louis. His research includes formal modelling, verification, implementation, and empirical evaluation of policies and mechanisms for enforcing timing, concurrency, footprint, fault-tolerance, and security properties in distributed, mobile, embedded, real-time, and cyber-physical systems. He developed the Kokyu real-time scheduling and dispatching framework used in several AFRL and DARPA projects and flight demonstrations, and led development of the nORB small-footprint real-time object request broker at Washington University. He has over 60 refereed technical publications and has an extensive record of service in review panels, standards bodies, workshops, and conferences for distributed, real-time, embedded, and cyber-physical systems.

Joseph Zambreno is an Associate Professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, where he has been since 2006. His research interests include computer architecture, compilers, embedded systems, and hardware/software co-design, with a focus on run-time reconfigurable architectures and compiler techniques for software protection. He was a recipient of a National Science Foundation Graduate Research Fellowship, a Northwestern University Graduate School Fellowship and a Walter P. Murphy Fellowship. He is a recent recipient of the NSF CAREER award (2012), as well as the ISU award for Early Achievement in Teaching (2012) and the ECpE Department's Warren B. Boast undergraduate teaching award (2009, 2011).

Phillip H. Jones received his BS in 1999 and MS in 2002 in Electrical Engineering from the University of Illinois at Urbana-Champaign, and his PhD in 2008 in Computer Engineering from Washington University in St. Louis. Currently, he is an Assistant Professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, where he has been since 2008. His research interests are in adaptive computing systems, reconfigurable hardware, embedded systems, and hardware architectures for application-specific acceleration.

This paper is a revised and expanded version of a paper entitled 'Improving system predictability and performance via hardware accelerated data structures' presented at Proceedings of the International Conference on Computational Science (ICCS), 2012, Omaha, Nebraska, USA, 4–6 June.

1 Introduction

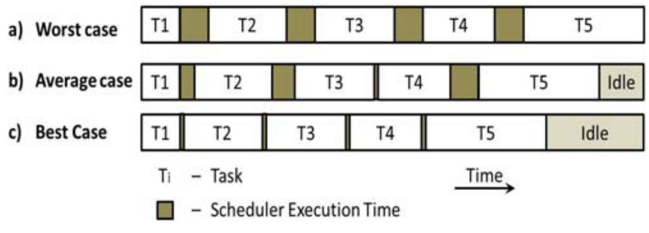
Deploying increasing amounts of computation into smaller form factor devices is required to keep pace with the ever increasing needs of real-time and embedded system applications. The area of micro unmanned aerial vehicles (UAVs) is an example of where such need exists. The size of these vehicles has rapidly decreased, while the capabilities users wish to deploy continue to explode. As recently as June 2011, the *New York Times* published several articles on the cutting-edge work being pursued by Wright Patterson Air Force Base to develop micro-drones to aid soldiers on the battlefield (Bumiller and Shanker, 2011). In February of 2011, the DARPA funded nano air vehicle (NAV) program demonstrated a humming bird form-factor UAV weighing less than 20 grams (e.g., less than an AA battery) (DARPA, 2011; Grossman et al., 2011) with video streaming capabilities. These real-time and embedded applications can no longer rely on manufacturing advances to provide computing performance at Moore's law rates, owing to transistors approaching atomic scales and thermal constraints (ITRS, 2009). Thus, more efficient use of the transistors available is needed. For example, use of application specific hardware has showed promise in accelerating various application domains, from cryptography (Eberle et al., 2008; Ors et al., 2008) to numerical simulation (Rahmouni et al., 2013) to control systems (Muller et al., 2013).

We assert that the boundaries of software and hardware must be reexamined and we believe a fruitful realm for research is the hardware-software co-design of functionality that has been traditionally implemented in software. Such a co-design is needed to balance the cost of dedicating limited silicon resources for high-performance fixed hardware functionality, with the flexibility and scalability offered by software. Additionally, we claim seamless migration between software and hardware implemented functionality is required to allow systems to adapt to the dynamic needs of applications. In this paper we examine a hybrid architecture for priority queue management and evaluate this architecture within a real-time scheduling context. The following motivates the importance of low processing overhead and timing predictably to a real-time scheduler's performance.

A real-time operating system (RTOS) is designed to execute tasks within given timing constraints. An important characteristic of an RTOS is predictable response under all conditions. The core of the RTOS is the scheduler, which ensures tasks are completed by their deadline. The choice of a scheduling algorithm is crucial for a real-time application. Online scheduling algorithms incur overhead, as the task queues must be updated regularly. This action is typically paced using a timer that generates periodic interrupts. The scheduler overhead generally increases with the number of tasks. A high resolution timer is required to distribute CPU load accurately based on a scheduling discipline in real-time systems, but such fine-grain time management increases the operating system overhead (Park et al., 2001; Adomat et al., 1996).

The extent to which a scheduler can ideally implement a given scheduling paradigm [e.g., earliest deadline first (EDF), rate monotonic (RM)], and thus provide the guarantees associated with that paradigm, is in part dependent on its timing determinism. A metric for helping quantify the amount of non-determinism that is introduced to the system by the scheduler is the variation in execution time among individual scheduler invocations. This can be roughly summarised by noting its best-case and worst-case execution times. Variations in scheduler execution time can be caused by system factors such as changes in task set composition, cache misses, etc. Reducing the scheduler's timing sensitivity to such factors can help increase deterministic behaviour, which in turn allows the scheduler to better model a given scheduling paradigm.

Figure 1 In order to allow analytical analysis of schedule feasibility, worst-case execution time (WCET) typically needs to be assumed (see online version for colours)



Note: Thus, scheduler execution time variations that cause large differences between WCET and typical case execution time reduce utilisation of system computing resources.

Figure 1 illustrates how the variation in scheduler overhead affects processor utilisation. To ensure that tasks meet their deadlines, the scheduler's worst-case execution times are often overestimated. This can cause a system to be underutilised and wastes CPU resources. In this paper, we examine how the scheduler overhead and its variation can be reduced by migrating scheduling functionality (along with time-tick interrupt processing) to hardware logic. The expected results of our efforts are increased CPU utilisation, better system predictability, finer schedule and timing resolution.

1.1 Contributions

The primary contributions of this paper are

1. a hardware accelerated binary min heap that supports enqueue and peek operations in O(1) time, returns the top-priority element in O(1) time, and completes a dequeue operation in $O(\log n)$ time
2. a scalable hardware-software priority queue architecture that enables fast and low-overhead transitions of queue management between hardware and hybrid modes of operation
3. a hybrid scheduler architecture that reduces scheduling overhead and improves predictability.

1.2 Organisation

The reminder of this paper is organised as follows. Section 2 describes the hardware-software priority queue architecture and implementation details. Section 3 describes the hardware scheduler architecture, which uses our priority queue design. The evaluation methodology and results are discussed in Sections 4 and 5. Section 6 presents related work on hardware accelerated priority queues and schedulers. Conclusions and future work are presented in Section 7.

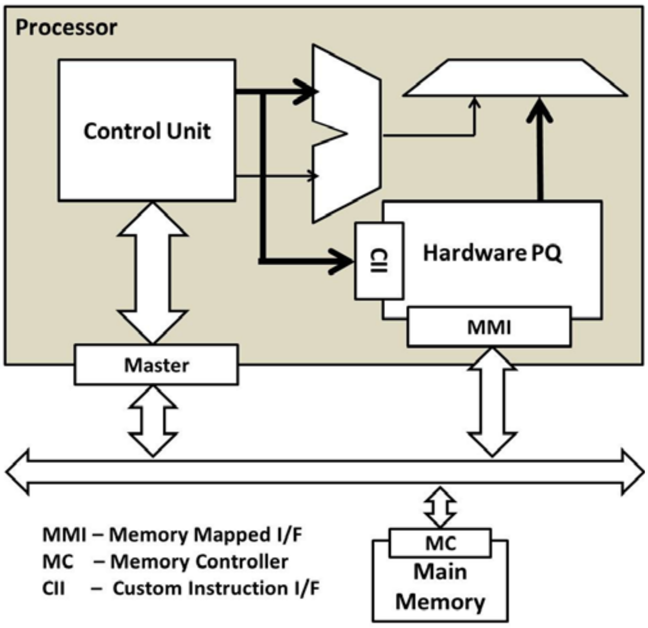
2 Hybrid priority queue architecture

Priority queues are commonly implemented using a binary heap data structure, which supports enqueue and dequeue operations in $O(\log n)$ time. A binary heap is constrained by the heap property, where the priority of each node is always less than or equal to its parent. In a binary min heap, lower key-value corresponds to higher priority and the root node has the highest priority (lowest key value). A binary heap can be stored as a linear array where the first element corresponds to the root. Given an index i of an element, $i/2$, $2i$ and $2i + 1$ are the indices of its parent, left and right child respectively.

Here we present a hybrid priority queue architecture that includes the hardware implementation of a conventional binary min heap (lower key value corresponds to higher priority), which can be managed in hardware and/or software. A binary heap could be stored compactly when compared to skip list, binomial heap and Fibonacci heap, without requiring additional space for pointers. Since the memory available in hardware (on-chip memory) is limited, the priority queue was implemented as a binary heap to better utilise the available resources. The priority queue operates in hardware mode when the queue size is less than a hardware limit threshold. When managed in hardware, the priority queue supports enqueue and peek operations in $O(1)$ time and dequeue operations in $O(\log n)$ time. Although the dequeue operation takes $O(\log n)$ time to complete, the top-priority (lowest key value) element can be returned immediately, allowing the dequeue operation to overlap its execution with the primary processor. Software issues custom instructions to initiate hardware-implemented enqueue and dequeue operations.

Once the priority queue size exceeds hardware limits, excess elements are stored in the system's main memory and managed by both hardware and software. Elements of the priority queue that are managed by hardware are memory mapped, providing software with direct access to these elements that are stored in a priority-queue-structured on-chip memory. Figure 2 illustrates this architecture. Memory mapping the priority-queue-structured on-chip memory additionally allows rarely used priority queue operations (e.g., delete element and decrease key) to be easily implemented in software, thus reducing the complexity of hardware control logic.

Figure 2 A high level block diagram of the hardware-base priority queue interface (see online version for colours)



2.1 Hardware priority queue

A high level architecture diagram for the priority queue is shown in Figure 3. Central to the priority queue is the queue manager, which provides the necessary interface to the CPU and executes operations on the queue. Elements in each level of the binary heap are stored in separate on-chip memories called block rams (BRAMs) to enable parallel access to elements, similar to Bhagwan and Lin (2000) and Ioannou and Katevenis (2007). The address decoder generates addresses and control signals for the BRAM blocks. Queue operations in hardware mode are explained in detail below, using a min-heap example, where a lower key value corresponds to a higher priority.

2.1.1 Enqueue

Enqueue operations in a software binary heap are accomplished by inserting the new element at the bottom of the heap and performing compare-swap operations with successive parents until the priority of the new element is less than its parent. In software, the worst-case behaviour of this operation occurs when the priority of the new element is greater than the rest of the nodes present in the heap. In this case, the new element bubbles-up all the way to the root of the heap [i.e., $O(\log n)$ time].

However, our hardware implementation can perform this operation in $O(1)$ time. We first calculate the path from the next vacant leaf node to the root. The index, i, of this leaf node is always one more than the current size of the queue, and each ancestor of this leaf node can be computed in parallel using a closed form equation (e.g., k^{th} parent is located at index $i/2^k$) in hardware. This path includes all ancestors from the leaf node to the heap's root. The heap property ensures that the elements in this path are in sorted order.

The shift register mechanism, shown in Figure 3, inserts a new element in constant time. This is similar to the shift-register priority queue described in Moon et al. (1997). Each level of the heap is mapped to an enqueue cell, which consists of a comparator, multiplexer and a register. The element to be inserted is broadcast to all the cells during an enqueue operation. The enqueue operation is then completed in the three steps shown in Figure 4. In the first step, all the elements in the path from the leaf node to the root node are loaded into the corresponding enqueue cells. The address for each BRAM block is generated by the address decoder. In the second step, the comparator in each enqueue cell compares the priority of the new element with the element stored locally and decides whether to latch the current element, new element or the element above it. In the final step, the elements along with the new entry are stored back into the heap.

Figure 3 The hardware priority queue architecture (see online version for colours)

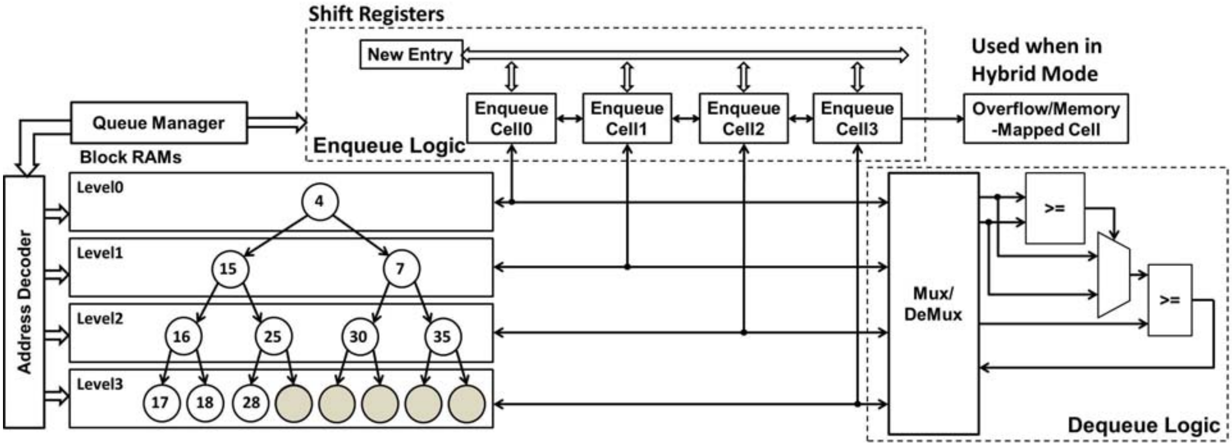


Figure 4 Steps of enqueue operation in hardware mode, (a) elements in the insertion path are loaded to enqueue cells (b) sorted insert of the new element to the enqueue cell array (c) elements in the enqueue cell array are stored back to the heap (see online version for colours)

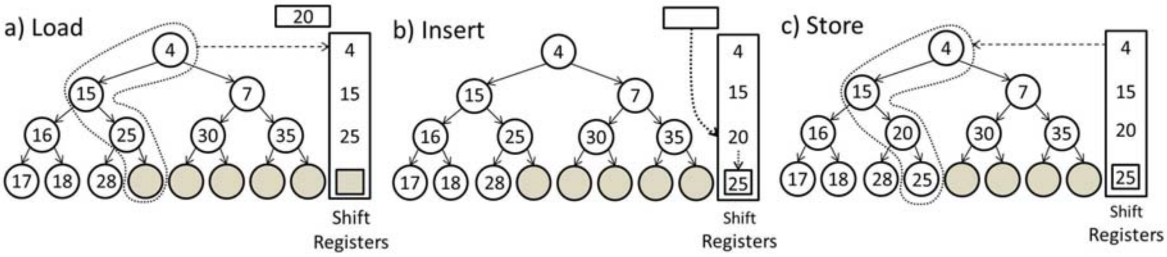
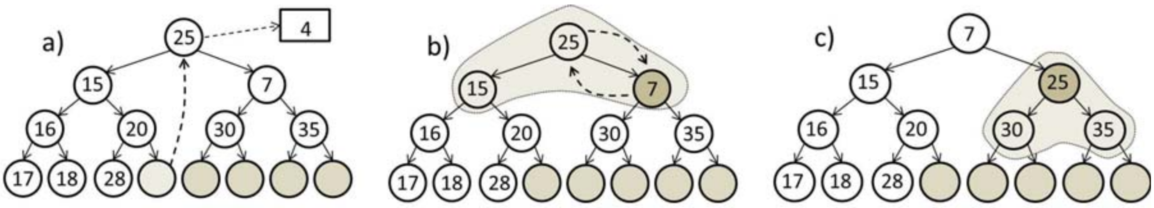


Figure 5 Steps of dequeue operation in hardware mode, (a) the root element is removed by replacing it with last element of the queue (b) new root is swapped with highest priority child (c) no more swap operations as the heap property is restored (see online version for colours)



Note: In worst case there will be log(n) swap operations.

2.1.2 Dequeue

Figure 5 illustrates an example of a dequeue operation in hardware mode. The dequeue operation can be divided into two stages: removing the root element from the queue (as the value to be returned by the dequeue call), and reconstruction of the heap. The root element is first removed by replacing it with the last element of the queue to keep the heap balanced. The new root element is then compared with its highest priority child and is swapped if its priority is less than that of its child. This operation is repeated until the priority of the new root element is greater than that of its children.

Note that the root element is returned immediately to the processor before restoring the heap property. The processor is not required to wait for the operation to complete, as the heap property of the queue is restored in hardware which executes in parallel to the CPU. Back-to-back dequeue operations would cause the processor to wait for the first operation to be completed in hardware before getting the result of the second request. Hence, the worst case execution time of a dequeue operation is $O(\log n)$.

2.1.3 Decrease-key and delete

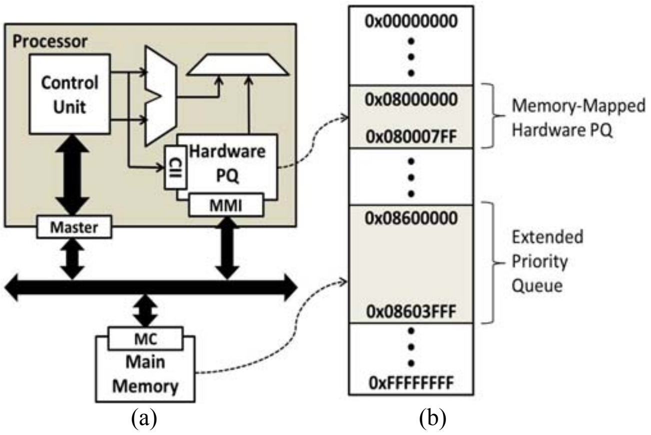
The decrease-key operation decreases the priority of a given queue element, and the delete operation removes a specified element from the queue. Supporting these rarely used operations in hardware adds considerable complexity to the hardware's control logic. To avoid this complexity, these operations have been implemented in software. Software accesses the hardware priority queue elements via a memory mapped interface as if they resided in main memory.

2.2 Hybrid priority queue management

The size of the hardware priority queue is limited by the available on-chip memory resources of the device. Gracefully handling size overflow situations allows the use of hardware data structures for a wider range of applications. We achieve this by extending the heap array to off-chip memory (i.e., main memory) and managing the queue in both hardware and software. In hybrid mode, the enqueue and dequeue operations are executed in two stages. The hardware executes a part of the queue operation in the first stage, and then control is returned to software, which completes the rest of the operation.

A memory mapped interface, shown in Figure 6(a), provides software access to on-chip priority queue elements as if they were resident in main memory. Since the address space of memory mapped hardware and the extended priority queue will typically not be part of the same continuous memory block, as shown in Figure 6(b). The queue algorithm needs to be modified accordingly to access the correct address depending on the array index of the element. The combination of memory mapping the hardware-base priority queue and implementing small modification to the queue algorithm enables our hybrid approach to have fast and low overhead transitioning between hardware and software management. The priority queue operations in hybrid mode are explained in detail below.

Figure 6 (a) Memory mapped interface provides access to priority queue elements stored in BRAM (b) Virtual address space showing extended priority queue (see online version for colours)



2.2.1 Enqueue

Figure 7 presents an example of the enqueue operation in hybrid mode. In the first stage of an enqueue operation, the new element is inserted into the hardware priority queue, which forms the top portion of the queue. This is similar to the hardware enqueue operation as explained in Section 2.1.1. Since we only go into hybrid mode when the queue size exceeds hardware limits, the lowest priority element in the hardware insertion path must be moved to the overflow buffer shown in Figure 3. This first stage is performed in constant time as explained in Section 2.1.1. Control is then returned to software. The overflow buffer is available to software through a memory mapped interface. In the second stage of the enqueue operation, the element in the overflow buffer is copied to the bottom of the extended queue and compare-swap operations are performed with successive parents until the heap property is restored. This stage is similar to the software enqueue operation and only the extended part of the queue (stored in main memory) is modified by software. The software implementation of enqueue operation is outlined in Algorithm 1.

Algorithm 1 Pseudocode of hybrid priority queue's enqueue operation

```
procedure HYBRiD_PQ_ENQUEUE(queue, elem)
    if Queue = Full then
        throwexception
    end if
    Hardware_pq_enqueue(elem)
    queue.size + +
    if queue.size > queue.hwlimit then
        index = queue.size
        Copy overflown hardware element to the end of software queue.
        queue.data[index] = overflow_cell
        while index > queue.hw_limit do
            if queue.data[index] <queue.data[parent(index)} then
                swap_queue_data(queue, index, parent(index))
                index = parent(index)
            end if
        end while
    end if
end procedure
```

Figure 7 Steps of enqueue operation in hybrid mode, in this example we assume that the first 3 levels of the heap are managed in hardware, (a) hardware elements in the insertion path are loaded to enqueue cells (b) sorted insert of the new element and the lowest priority element is moved to the overflow buffer (c) hardware stores back the elements in enqueue cells and the overflow buffer element is moved to the bottom of the queue by software (d) software performs compare-swap operation to restore heap property (see online version for colours)

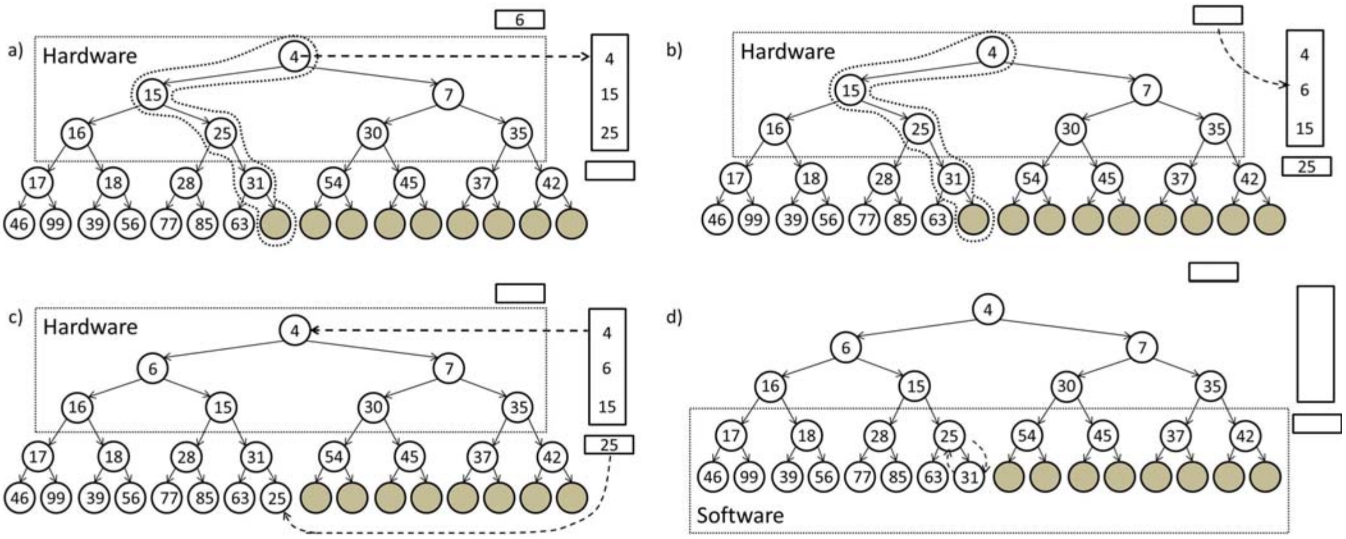
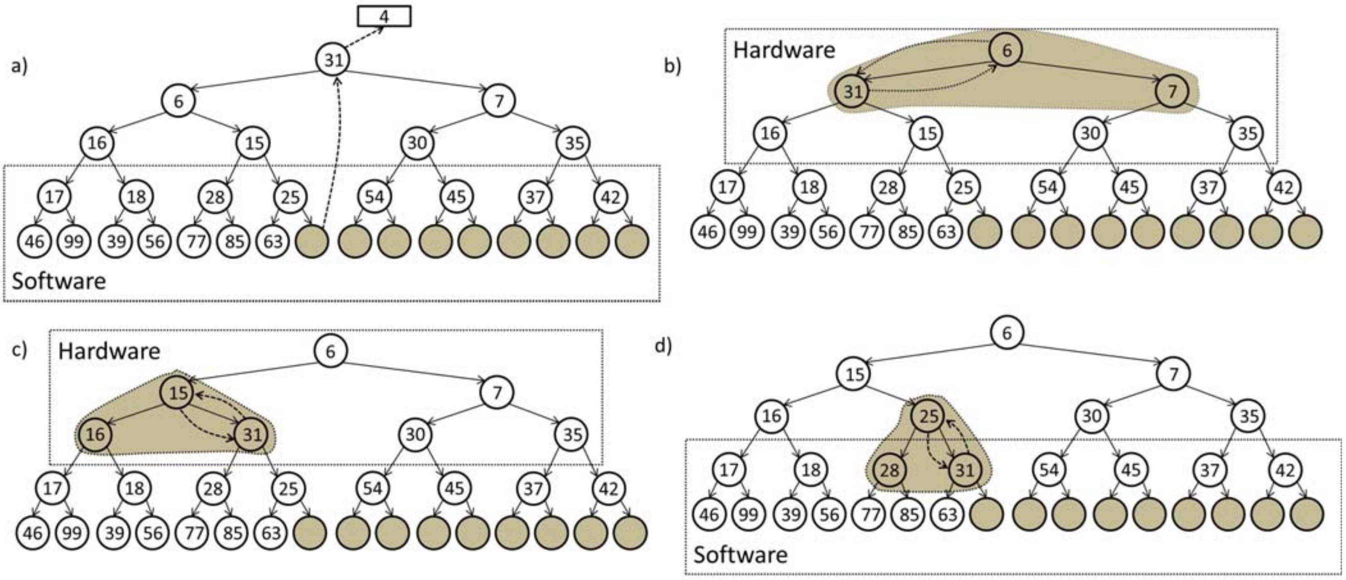


Figure 8 Steps of dequeue operation in hybrid mode, in this example we assume that the first three levels of the heap are managed in hardware, (a) the root element is removed by replacing it with the last element of the queue by software (b) the heap property is restored by swapping the new root (31) with highest priority child (c) hardware completes dequeue operation and returns the position of new root(31) (d) software continues restoring the heap property from the position returned (see online version for colours)



2.2.2 Dequeue

Figure 8 provides an example of the dequeue operation in hybrid mode. In the first stage of a dequeue operation, the root element of the queue is removed by replacing it with the last element of the queue. This operation should be performed by software, since the last element of the queue resides in main memory. The hardware dequeue operation is then initiated through a custom instruction, which restores the heap property of the hardware portion of the queue as explained in Section 2.1.2. The custom instruction when completed returns the position of the newly inserted element, which can be accessed by software through memory mapped interface. The software then continues restoring the heap property starting from the position returned. The software implementation of dequeue operation is outlined in Algorithm 2.

Comparing our approach with the related work reported in Section 6, our approach scales nicely without requiring complex hardware control logic to manage pipelining. Our hardware-software co-design approach overcomes the size limitations of hardware, enabling the support of arbitrarily large priority queues.

3 Hardware scheduler

3.1 Overview

As an application of the priority queue described above, we propose a hardware-software scheduler architecture designed to reduce the time-tick interrupt processing and scheduling overhead of a system. In addition, our hybrid architecture increases the timing determinism of the scheduler operations. The instruction set architecture of a processor was extended to support a set of custom instructions to communicate with the scheduler. The hardware scheduler executes the scheduling algorithm and returns control to the processor along with the next task to execute. Software then performs context switching before executing the next task.

Algorithm 2 Pseudocode of hybrid priority queue's dequeue operation

```
procedure HYBRiD_PQ_DEQUEUE(queue)
    if Queue = Empty then
        throw exception
    end if
    result = queue.top;
    if queue.size < queue.hw_limit then
        hardware_pq_dequeue()
    else
        Replace root with last element of heap array.
        queue.data[0] = queue.data[size]
        Execute hardware dequeue and return position of
        newly inserted element.
        new_index = hardware_pq_dequeue()
        Continue heap restoration in software from the
        position returned.
        Restore_sw_heap(new_index)
    end if
    queue.size – –;
end procedure
```

A software timer periodically generates interrupts to check for the availability of a higher priority task. The check is accomplished using a single custom instruction that returns a preempt flag, set by the hardware scheduler, based on which the processor chooses to continue executing the current task or preempts it to run a higher priority task. The following describes the functionality of the key components of the hardware accelerated scheduler.

3.2 Architecture

A high level block diagram of the hardware scheduler is shown in Figure 9.

3.2.1 Controller

The controller is the central processing unit of the scheduler. It is responsible for the execution of the scheduling algorithm. The controller processes instruction calls from the processor and monitors task queues (ready queue and sleep queue).

3.2.2 Timer unit

The timer unit keeps track of the time elapsed since the start of the scheduler. This provides accurate high-resolution timing for the scheduler. The resolution of the timer-tick can be configured at run time.

3.2.3 CPU interface

The interface to the scheduler is provided through a set of custom instructions as an extension to the instruction set architecture of the processor. This removes bus arbitration timing dependencies for data transfer. Basic scheduler operations such as run, configure, add task, and preempt task are supported.

3.2.4 Task queues

At the core of the scheduler are the task queues, which are implemented as priority queues. The ready queue stores active tasks based on their priority. The sleep queue stores inactive tasks until their activation time. The task with the earliest activation time is located at the front of the sleep queue.

3.3 Modes of operation

The scheduler is designed to operate in either hardware or hybrid mode, depending on the size of the hardware priority queues and the number of tasks in the system. Once the number of tasks exceeds the hardware limit, the queues extend to off-chip memory (i.e., main memory) and the scheduler starts operating in hybrid mode. In hybrid mode the scheduling algorithm is executed in software and the hybrid priority queues described in Section 2 are used to accelerate scheduler operations. This transition involves stalling the hardware scheduler through a co-processor call (custom instruction) and calling the software scheduler function. As the elements stored in the on-chip priority queues can be accessed by software via a memory mapped interface, it avoids the need to copy data between hardware and software memory when the scheduler changes modes. The proposed scheduler architecture scales to support an arbitrarily large number of tasks.

4 Evaluation methodology

4.1 Platform

The hybrid priority queue and the scheduler were deployed and evaluated on the re-configurable autonomous vehicle infrastructure (RAVI) board, an in-house developed FPGA prototyping platform. RAVI leverages field programmable gate array (FPGA) technology to allow custom hardware to be tightly integrated to a soft-core processor on a single computing device. It enables exploration of the software/hardware co-design space for designing system architectures that best fit an application's requirements. The portions of the RAVI board used for our experiments included the Cyclone III FPGA, the on-board DDR DRAM and the UART. The FPGA was used to implement the NIOS-II (Altera's soft-processor), the DDR stored software that was executed on the NIOS-II, and the UART supported data collection. A pictorial description of the setup is shown in Figure 10.

4.2 Architecture configuration

The priority queue and the scheduler were implemented as an extension to the instruction set architecture (using custom instructions) of a Nios II embedded processor running at 50 MHz on an Altera Cyclone III FPGA. The priority queue supported up to 255 elements in hardware mode and up an arbitrarily large number of elements in hybrid mode of operation. For our evaluation we limited the queue size to 8,192 elements. A binary heap-based priority queue implemented in software was used as a baseline to compare against the performance of our hybrid priority queue.

The scheduler can support up to 255 tasks when managed in hardware, and up to an arbitrarily large number of tasks when in hybrid mode. For our evaluation we limited the task set size to 2,048, which is sufficient to support a vast majority of embedded systems. The scheduler can be configured to use EDF or a fixed priority-based scheduling algorithm such as rate monotonic scheduling (RMS). The scheduler overhead was also measured using different timer-tick resolutions (0.1 ms, 1 ms, 10 ms), which is used to generate periodic interrupts for the scheduler. A software test bench was built to accurately measure the overhead of the scheduler for different task sets and timer resolutions. Hardware-based performance counters, supported by the NIOS II processor provided a relatively unobtrusive mechanism to profile software programs including interrupt service routines in real-time. An EDF (Liu and Layland, 1973) scheduler was deployed to measure the impact of running a dynamic scheduling algorithm on the processor. In EDF scheduling, task priorities are assigned based on the absolute deadline of the current request. At any given time, the task with the nearest deadline will be assigned the highest priority and executed. A software EDF scheduler implementation was used as a baseline to compare against our hybrid implementation.

Figure 9 A high level architecture diagram of the hardware scheduler along with the custom instruction interface (see online version for colours)

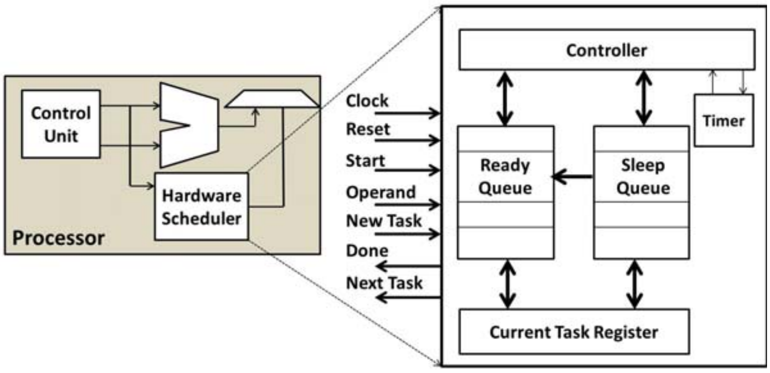
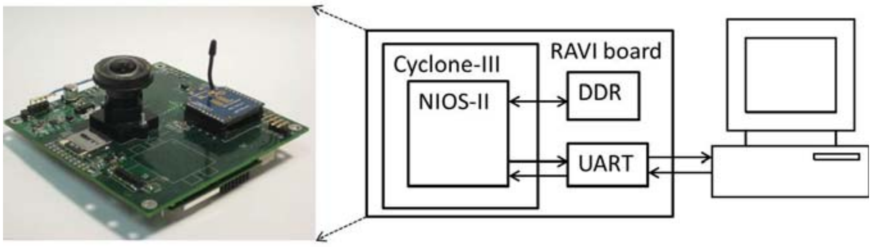


Figure 10 FPGA-based evaluation platform (see online version for colours)



4.3 Workload and metrics

The performance of the priority queue was evaluated using the classic hold model (Vaucher and Duval, 1975); Jones, 1986), where a priority queue of a given size is initialised and hold operations (dequeue followed by enqueue) are performed repeatedly on the queue. The size of the queue remains constant for the whole duration of the experiment. The access time measured by the hold model is dependent on the initial queue size and priority increment distribution. For our evaluation we used exponential, uniform, bimodal and triangular distributions, similar to those used in Vaucher and Duval (1975) and Ronngren and Ayani (1997). The transient behaviour of the priority queue is measured using the up/down model (Ronngren et al., 1991), where the queue is initialised to a given size by series of enqueue operation and then emptied by series of dequeue operation.

A set of periodic tasks with randomly generated parameters (i.e., task execution time and period) was used to evaluate the performance of the EDF scheduler. The relative deadline of the tasks were assumed to be equal to their period. The number of tasks in the task set were varied, keeping the utilisation factor constant at 80%. The metrics used to evaluate our scheduler were:

- scheduler overhead: time spent executing the scheduling algorithm
- timer-tick overhead: time taken to service the periodic timer interrupt
- predictability: variation in the execution time of individual scheduler invocations.

5 Results and analysis

This section presents the results of our hybrid priority queue versus software priority queue evaluation experiments. A discussion is then given on the results of our hybrid and hardware scheduler evaluation experiments.

5.1 Priority queue

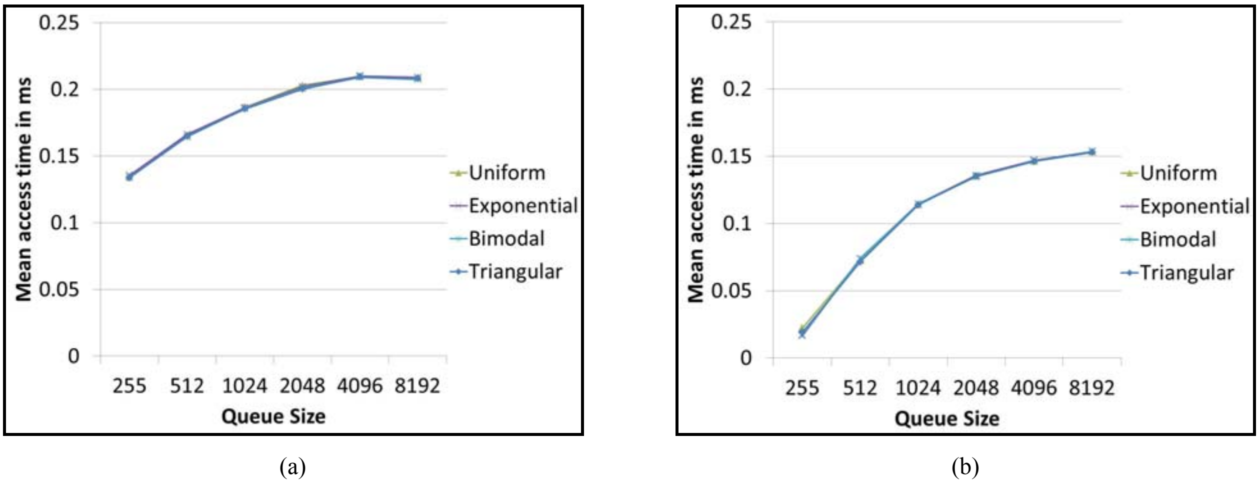
5.1.1 Mean access time

The mean access times of the hybrid and software priority queues measured using classic hold and up/down experiments are shown in Figures 11 and 12. The hybrid priority queue is fully managed in hardware when the queue size is 255 or less. The results show that the hybrid queue is six times faster than the software queue when the queue size is 255. The hybrid priority queue extends to software memory when the queue size exceeds 255 elements and the fraction of total work done in hardware decreases as more levels of heap are stored in software memory. Hence, the difference in performance between the hybrid and software priority queue decreases as the size of the queue increases. Even when the queue contains 8,192 elements, the hybrid priority queue performs close to 30% better than software priority queue. The performance of the hybrid and software priority queue is not very sensitive to priority increment distributions.

5.1.2 Resource utilisation and scalability

We implemented our hardware priority queue design on an Altera Cyclone III (EP3C25) FPGA. The resource utilisation of the priority queue for different queue lengths is shown in Table 1. Each priority queue element is 64 bits wide, with a 32 bit priority value. The amount of combinational logic required increases *logarithmically* with the size of priority queue. Since the number of elements doubles with each additional level, the combinational logic scales logarithmically with queue size. The device contains 66 M9K memory blocks, which can be used as on chip memory. Each M9K block can hold 8,192 memory bits with a maximum data port width of 36. Since each level of the heap is stored in a BRAM with a 64 bit wide port, a minimum of 2 M9K blocks are used per level. The BRAM usage can be optimised by moving the first 5 levels of the heap to memory mapped registers. We also implemented the shift-register and systolic array-based priority queue architectures described in Moon et al. (1997). The resource utilisation of both architectures are shown in Table 2. These architectures use distributed memory instead of BRAMs to store queue elements. Figure 13 shows that our queue architecture scales well for large queues, as compared to shift-register and systolic array-based architectures (Moon et al., 1997) in which the combinational logic required increases linearly with queue size.

Figure 11 Performance comparison between the software and hybrid implementation of a priority queue, (a) software priority queue (b) hybrid priority queue (see online version for colours)



Note: Evaluated using the classic hold model, for four different priority increment distributions.

Table 1 FPGA resource utilisation of the proposed priority queue design for different queue sizes

Size	*Resources*[1]			
	Look-up tables (LUTs)	*Flip-flops*	*Memory (bits)*	*BRAMs*
31	1,411 (5.73%)	906 (3.68%)	1,920 (0.32%)	8 (12.12%)
63	1,996 (8.1%)	1,048 (4.25%)	3,968 (0.65%)	10 (15.15%)
127	2,561 (10.4%)	1,182 (4.8%)	8,064 (1.325%)	12 (18.18%)
255	3,161 (12.84%)	1,330 (5.4%)	16,256 (2.67%)	14 (21.21%)

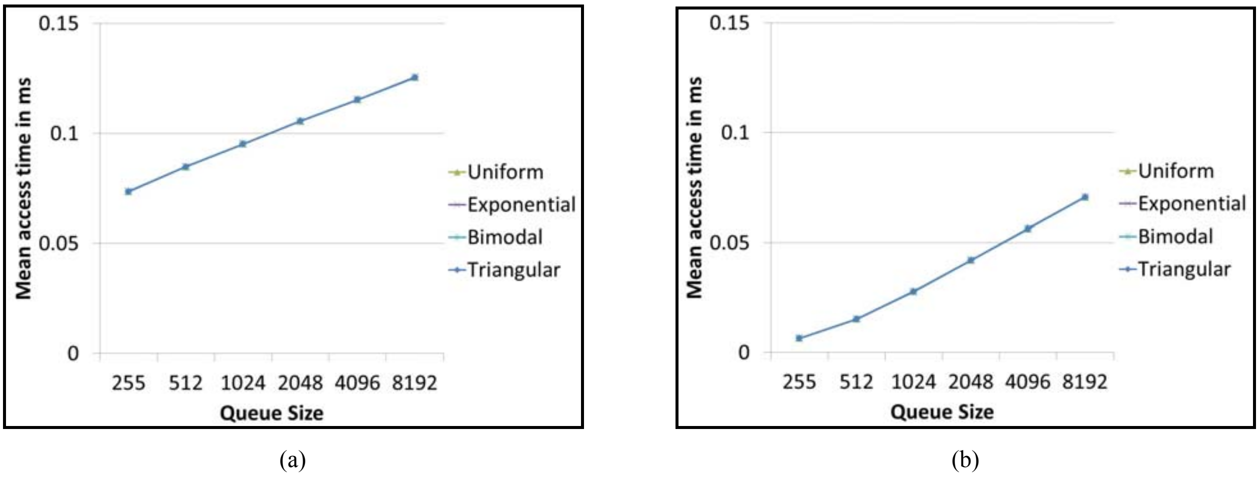
Note: [1]Altera Cylone III FPGA contains: 24,624 LUTs, 24,624 flip-flops and 66 BRAMs.

Table 2 FPGA resource utilisation of shift register and systolic array-based priority queue architectures (Moon et al., 1997) in comparison with proposed priority queue design

Size	*Shift register*		*Systolic array*		*Proposed design*	
	LUTs	*Flip-flops*	*LUTs*	*Flip-flops*	*LUTs*	*Flip-flops*
31	4,995 (20.29%)	2,077 (8.43%)	8,560 (34.76%)	3,999 (16.24%)	1,411 (5.73%)	906 (3.68%)
63	10,275 (41.73%)	4,221 (17.14%)	17,520 (71.15%)	8,127 (33.00%)	1,996 (8.1%)	1,048 (4.25%)
127	20,835 (84.61%)	8,509 (34.56%)	–	–	2,561 (10.4%)	1,182 (4.8%)
255	–	–	–	–	3,161 (12.84%)	1,330 (5.4%)

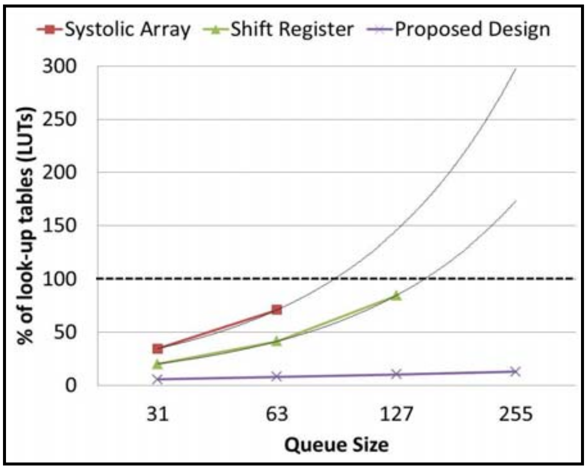
Note: – Configurations for which the priority queue resources do not fit in Altera Cyclone III FPGA.

Figure 12 Performance comparison between the software and hybrid implementation of a priority queue, (a) software priority queue (b) hybrid priority queue (see online version for colours)



Note: Evaluated using the up/down model, for four different priority increment distributions.

Figure 13 Comparing FPGA look-up table utilisation of the proposed priority queue design against shift register and systolic array-based priority queue architectures (Moon et al., 1997) for different queue sizes (see online version for colours)



Note: Flip-flop utilisation also shows a similar trend.

5.2 Scheduler

For our analysis we have considered the following three configurations of an EDF scheduler.

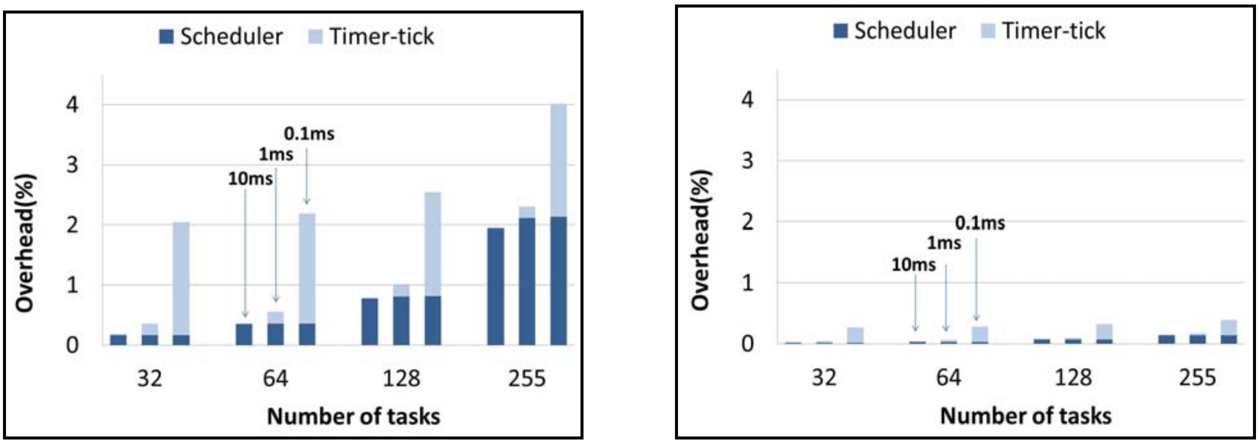
- Software scheduler: used as the baseline for evaluating our hybrid and hardware scheduler. Evaluated for up to 2,048 tasks.
- Hardware scheduler: executes scheduling algorithm, manages task queues, and supports up to 255 tasks in hardware.
- Hybrid scheduler: the task queues of the software scheduler is replaced by our hybrid priority queue to accelerate scheduler operations. Evaluated for up to 2,048 tasks.

5.2.1 Scheduler overhead

The overhead of the scheduler was measured for different sets of tasks and timer tick resolutions. Figure 14(a) shows the percentage overhead of software scheduler. The software scheduler overhead increases with the number of tasks and the timer-tick resolution. Most of this overhead results from time tick processing, where the scheduler periodically processes interrupt requests to check for new tasks and managing the task queues. This time-tick processing has been a limiting factor for implementing dynamic priority-based scheduling algorithms in embedded real time systems (Park et al., 2001; Adomat et al., 1996), since finer granularity time ticks lead to closer to ideal implementation of such schedulers.

Figure 14(b) shows the scheduling overhead when the hardware scheduler is used. The results show that when the timer tick resolution is set to 0.1 ms and with 255 tasks, the scheduler overhead is less than 0.4%. This is a 90% reduction in scheduler overhead as compared to the software implementation. Most of the scheduling overhead is eliminated by the hardware scheduler, as the time tick processing and a majority of the scheduling functionality is migrated to hardware. A call to the software scheduler is now replaced by a custom instruction call to obtain the next task for execution or to preempt the current task. The overhead of managing the task queues in software is removed, as the scheduler runs in parallel to the processor and hardware priority queues are used to accelerate task queue management. The time tick processing overhead is reduced considerably as the software interrupt service routine just needs to execute a single instruction to check the availability of a higher priority task in the hardware scheduler.

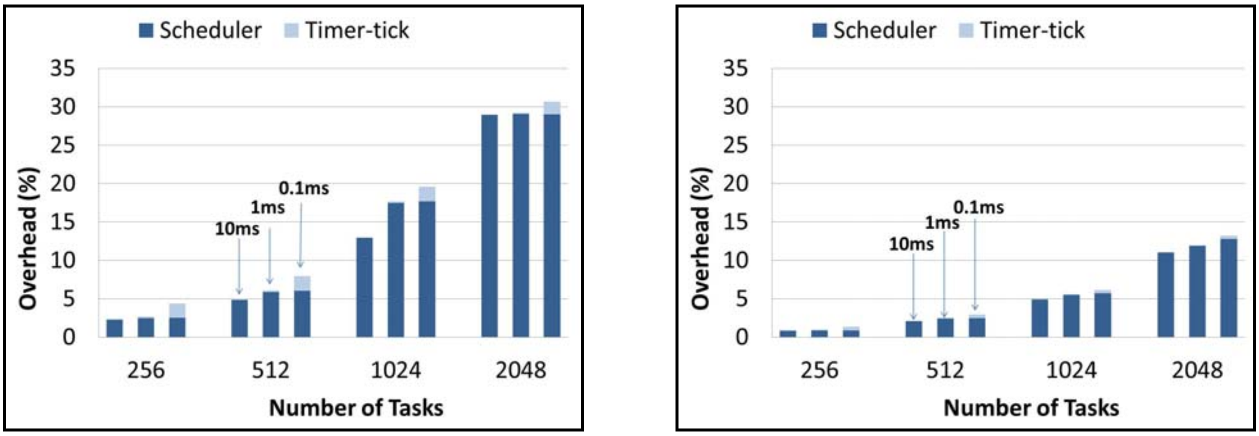
Figure 14 Performance of the software scheduler compared with hardware scheduler for task sizes less than or equal to 255, (a) software scheduler (b) hardware scheduler (see online version for colours)



(a)

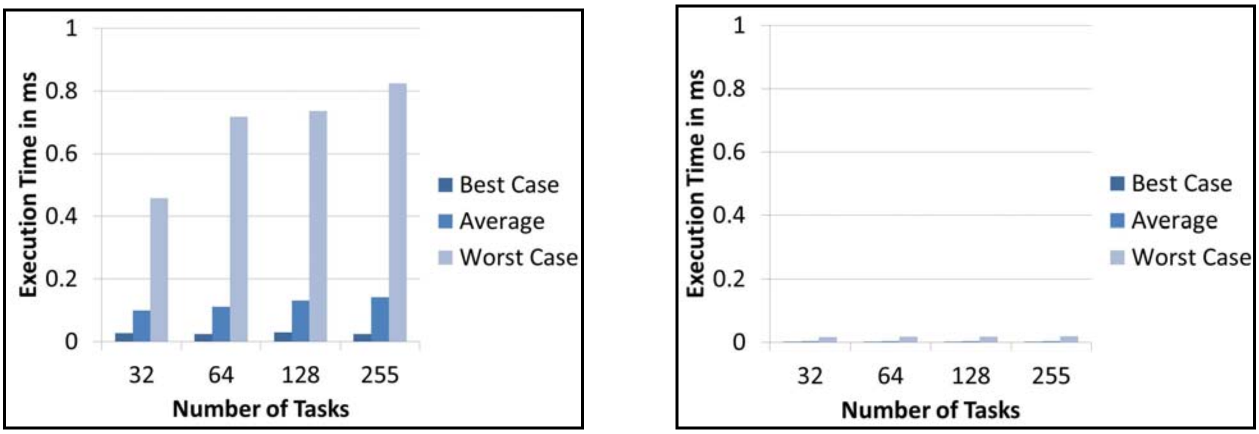
(b)

Figure 15 Performance of software scheduler compared with hybrid scheduler for task sizes greater than 255, (a) software scheduler (b) hardware scheduler (see online version for colours)



(a) (b)

Figure 16 Variation in execution times of software and hardware scheduler, (a) software scheduler (b) hardware scheduler (see online version for colours)



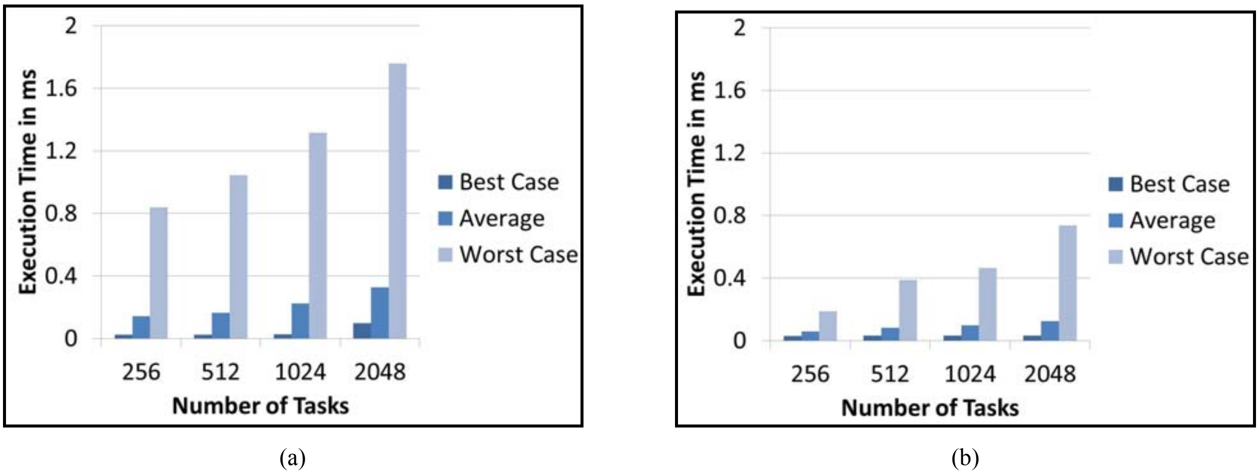
(a) (b)

Once the number of tasks exceeds 255, our scheduler executes in hybrid mode where the scheduling algorithm runs in software and queue operations are accelerated using our hybrid priority queues. The switching between hardware and hybrid scheduler mode is quick and has little or no overhead in part due to the hardware queues being memory mapped. The overhead of the scheduler in hybrid mode is 50% less than the software scheduler overhead as seen in Figure 15.

5.2.2 Predictability

The predictability of the scheduler can be measured as the variation in the execution time of a single call to the scheduler. The variation in execution times of the hardware and software scheduler is shown in Figure 16. The difference between the best case and worst case execution time of the software scheduler is 50 times larger then the hardware implementation as shown in Figure 16. This variation for the software implementation is due to system factors such as changes in task-set composition, cache misses, etc. The processing time of the software priority queues (task queues) varies, as it depends on the current queue size and task parameters. These variations can make the scheduler a significant source of non-determinism in real-time systems. Since the system must be designed for worst case behaviour to ensure task deadlines are met, increases in execution time variation reduces CPU task utilisation (i.e., CPU becomes underutilised). On the other hand, the execution times of the hardware scheduler show more deterministic behaviour with very little variation. Migrating time-tick processing to hardware and the use of hardware accelerated priority queues results in tighter worst-case execution time bounds for the scheduler. This in turn leads to higher CPU task utilisation. Figure 17 shows the variation in execution time of the hybrid scheduler in comparison with the software scheduler. The use of hybrid priority queues in the software scheduler reduces the variation in the scheduler execution time by more than 50% as shown in Figure 17.

Figure 17 Variation in execution times of software and hybrid scheduler, a) software scheduler (b) hardware scheduler (see online version for colours)



6 Related work

6.1 Hardware priority queues

Many hardware priority queue architectures have been implemented in the past, most of them in the realm of real-time networks for packet scheduling (Moon et al., 1997; Bhagwan and Lin, 2000; Ioannou and Katevenis, 2007). Moon et al. (1997) compared four scalable priority queue architectures: first-in-first-out, binary tree, shift registers and systolic array-based. The shift-register architecture suffers from bus loading, as new tasks must be broadcasted to all the queue cells. The systolic array architecture overcomes the problem of bus loading at the cost of doubling hardware storage requirements. The hardware complexity for both the shift register and systolic array architecture increases linearly with the number of elements, as each cell requires a separate comparator. This makes these architectures expensive to scale in terms of hardware resources.

Bhagwan and Lin (2000) proposed a new pipelined priority queue architecture based on p-heap (a new data structure similar to binary heap). A pipelined heap manager was proposed in Ioannou and Katevenis (2007) to pipeline conventional heap data structure operations. Both of these pipelined implementations of a priority queue are scalable and are designed to achieve high throughput, but at the expense of increased hardware complexity.

The size of the priority queues discussed above is limited by the availability of on-chip memory. A hybrid priority queue system (HPQS) was proposed in Zhuang and Pande (2006), where both SRAM and DRAM was used to store large priority queues used in high speed network devices. A java-based hardware-software priority queue was proposed in Chandra and Sinnen (2010), where a shift-register-based priority queue (Moon et al., 1997) was extended by appending a software binary heap. Bloom et al. (2012) presented an exception-based mechanism for handling overflows in hardware priority queue, where additional data is moved to secondary storage by the exception handler.

6.2 Hardware schedulers

Several architectures (Adomat et al., 1996; Burleson et al., 1999; Saez et al., 1999; Kuacharoen et al., 2003; Gupta et al., 2010; Kohout et al., 2003) have been proposed to improve the performance of schedulers using hardware accelerators. Most schedulers implement some kind of priority-based scheduling algorithm that requires a priority queue to sort the tasks based on their priority. A real time kernel called FASTHARD has been implemented in hardware (Adomat et al., 1996). The scheduler of FASTHARD can handle 256 tasks and eight priority levels. The Spring scheduling coprocessor (Burleson et al., 1999) was built to accelerate scheduling algorithms used in the Spring kernel (Stankovic and Ramamritham, 1991), which was used to perform feasibility analysis of the schedule. Kuacharoen et al. (2003) implemented a configurable hardware scheduler that provided support for three scheduling disciplines, configurable during runtime. A slack stealing scheduling algorithm was implemented in hardware (Saez et al., 1999) to support scheduling of tasks (periodic and aperiodic) and to reduce scheduling overhead. Nakano et al. (1995) implemented most of the/xITRON kernel functionality including tasks scheduling in a co-processor called STRON-1 which reduced the kernel overhead. A hardware scheduler for multiprocessor system on chip is presented in Gupta et al. (2010), which implements the Pfair scheduling algorithm. A real time task manager (RTM) that implements scheduling, time management, and event management in hardware is presented in Kohout et al. (2003). The RTM supports static priority-based scheduling and is implemented as an on-chip peripheral that communicates with the processor though a memory mapped interface. The SERRA run-time scheduler synthesis and analysis tool was presented in Mooney and Micheli (1997). The tool automatically generated a run-time hardware-software scheduler from system level specification. A hardware-software kernel was presented in Morton and Loucks (2004), which implemented a scheduling co-processor running EDF scheduling algorithm.

A hardware real-time scheduler coprocessor (HRTSC) architecture for NIOS II processor was described in Varela et al. (2012), which could be configured to run any priority-based scheduling discipline.

The hardware priority queues described above use on-chip memory to store data, which limits the size of the queue due to resource constraints of the device. In our hybrid priority queue architecture, the hardware priority queue can be extended into off-chip memory and managed in both hardware and software, when the queue size exceeds hardware limits. The priority queue, when managed in hardware, supports constant time enqueue operations and dequeue operations in $O(\log\ n)$ time. The hardware utilisation of the our priority queue increases logarithmically with the queue size and avoids complex pipelining logic.

One of the limitations of the hardware schedulers described above is that, once deployed, they can only support a fixed number of tasks. Our hybrid scheduler architecture overcomes this limitation by switching between hardware and software modes of operation depending on the number of tasks in the system. The transitions between hardware and software is fast and has low overhead. The hybrid priority queue is used as a part of our real-time scheduler to improve performance and timing predictability.

7 Conclusions and future work

A new hybrid priority queue architecture has been implemented, which can be managed in hardware and/or software. The priority queue when managed in hardware supports enqueue and peek operations in O(1) time, returns the top-priority element in O(1) time, and completes a dequeue operation in $O(\log\ n)$ time. The design enables quick and low overhead transition between hardware and software management. We utilise hardware logic to enhance the performance of queue operations even when managing the priority queue in software. As an application of the proposed priority queue architecture, a scalable hybrid scheduler is implemented that supports 255 tasks in hardware mode and up to an arbitrarily large number of tasks in hybrid mode. The scheduler when managed in hardware, showed up to 90% reduction in scheduler overhead when compared to the software scheduler. Our results show that the hardware scheduler has 98% less variation in execution time when compared to the software scheduler, thus giving more predictable execution times, which is necessary in high-performance real time systems.

Avenues of future work include,

1 reducing the rate of performance degradation as queue overflows into software,

2 evaluating the use of our hybrid priority queue in discrete event simulation and network optimisation algorithms

3 integrating our hybrid scheduler with Real-time Linux and characterising the scheduler performance.

Acknowledgements

This work is supported in part by the National Science Foundation (NSF) under award CNS-1060337, and by the Air Force Office of Scientific Research (AFOSR) under award FA9550-11-1-0343.

References

Adomat, J., Furunas, J., Lindh, L. and Starner, J. (1996) 'Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems', in *Real-Time Systems, Proceedings of the Eighth Euromicro Workshop on*, pp.164–168.

Bhagwan, R. and Lin, B. (2000) 'Fast and scalable priority queue architecture for high-speed network switches', in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings*, Vol. 2, pp.538–547.

Bloom, G., Parmer, G., Narahari, B. and Simha, R. (2012) 'Shared hardware data structures for hard real-time systems', in *Proceedings of the tenth ACM International Conference on Embedded Software, EMSOFT '12*, pp.133–142, ACM, New York, NY, USA.

Bumiller, E. and Shanker, T. (2011) 'War evolves with drones, some tiny as bugs' [online] http://www.nytimes.com/2011/06/20/world/20drones.html (accessed February 2014).

Burleson, W., Ko, J., Niehaus, D., Ramamritham, K., Stankovic, J., Wallace, G. and Weems, C. (1999) 'The spring scheduling coprocessor: a scheduling accelerator', *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol. 7, No. 1, pp.38–47.

Chandra, R. and Sinnen, O. (2010) 'Improving application performance with hardware data structures', in *Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW), IEEE International Symposium on*, pp.1–4.

DARPA (2011) 'Nano air vehicle (NAV)' [online] http://www.darpa.mil/NewsEvents/Releases/2011/11/24.aspx (accessed February 2014).

Eberle, H., Gura, N., Shantz, S.C. and Gupta, V. (2008) 'A cryptographic processor for arbitrary elliptic curves over gf (2m)', *International Journal of Embedded Systems*, Vol. 3, No. 4, pp.241–255.

Grossman, L., Brock-Abraham, C., Carbone, N., Dodds, E., Kluger, J., Park, A., Rawlings, N., Suddath, C., Sun, F., Thompson, M., Walsh, B. and Webley, K. (2011) 'The 50 best inventions', *Time Magazine*.

Gupta, N., Mandal, S., Malave, J., Mandal, A. and Mahapatra, R. (2010) 'A hardware scheduler for real time multiprocessor system on chip', in *VLSI Design, VLSID '10, 23rd International Conference on*, pp.264–269.

Ioannou, A. and Katevenis, M. (2007) 'Pipelined heap (priority queue) management for advanced scheduling in high-speed networks', *Networking, IEEE/ACM Transactions on*, Vol. 15, No. 2, pp.450–461.

ITRS (2009) 'The International Technology Roadmap for Semiconductors (ITRS)', *Lithography* [online] http://www.itrs.net/ (accessed February 2014).

Jones, D.W. (1986) 'An empirical comparison of priority-queue and event-set implementations', *Commun. ACM*, Vo. 29, No. 4, pp.300–311.

Kohout, P., Ganesh, B. and Jacob, B. (2003) 'Hardware support for realtime operating systems', in *Hardware/Software Codesign and System Synthesis, First IEEE/ACM/IFIP International Conference on*, pp.45–51.

Kuacharoen, P., Shalan, M.A. and Mooney III, V.J. (2003) 'A configurable hardware scheduler for real-time systems', in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp.96–101, CSREA Press.

Liu, C. and Layland, J. (1973) 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *Journal of the ACM (JACM)*, January, Vol. 20, No. 1, pp.46–61.

Moon, S-W., Shin, K. and Rexford, J. (1997) 'Scalable hardware priority queue architectures for high-speed packet switches', in *Real-Time Technology and Applications Symposium, Proceedings, Third IEEE*, pp.203–212.

Mooney, V.J. and Micheli, G.D. (1997) *Hardware/Software Co-design of Run-time Schedulers for Real-time Systems*, Technical report, Stanford, CA, USA.

Morton, A. and Loucks, W.M. (2004) 'A hardware/software kernel for system on chip designs', in *Proceedings of the 2004 ACM Symposium on Applied Computing, SAC '04*, pp.869–875, ACM, New York, NY, USA.

Muller, M., Klockner, J., Gushchina, I., Pacholik, A., Fengler, W. and Amthor, A. (2013) 'Performance evaluation of platform-specific implementations of numerically complex control designs for nano-positioning applications', *International Journal of Embedded Systems*, Vol. 5, No. 1, pp.95–105.

Nakano, T., Utama, A., Itabashi, M., Shiomi, A. and Imai, M. (1995) 'Hardware implementation of a real-time operating system', in *TRON Project International Symposium, Proceedings of the 12th*, pp.34–42.

Ors, B., Batina, L., Preneel, B. and Vandewalle, J. (2008) 'Hardware implementation of an elliptic curve processor over gf (p) with montgomery modular multiplier', *International Journal of Embedded Systems*, Vol. 3, No. 4, pp.229–240.

Park, T.R., Park, J.H. and Kwon, W.H. (2001) 'Reducing OS overhead for real-time industrial controllers with adjustable timer resolution', in *Industrial Electronics, ISIE, IEEE International Symposium on*, Vol. 1, pp.369–374.

Rahmouni, K., Chabanet, S., Lambelin, N. and Petrot, F. (2013) 'Design of a medium voltage protection device using system simulation approaches: a case study', *International Journal of Embedded Systems*, Vol. 5, No. 1, pp.53–66.

Ronngren, R. and Ayani, R. (1997) 'A comparative study of parallel and sequential priority queue algorithms', *ACM Trans. Model. Comput. Simul.*, Vol. 7, No. 2, pp.157–209.

Ronngren, R., Riboe, J. and Ayani, R. (1991) 'Lazy queue: an efficient implementation of the pending-event set', in *Proceedings of the 24th annual symposium on Simulation, ANSS '91*, pp.194–204, IEEE Computer Society Press, Los Alamitos, CA, USA.

Saez, S., Vila, J., Crespo, A. and Garcia, A. (1999) 'A hardware scheduler for complex real-time systems', in *Industrial Electronics, ISIE '99, Proceedings of the IEEE International Symposium on*, Vol. 1, pp.43–48.

Stankovic, J. and Ramamritham, K. (1991) 'The spring kernel: a new paradigm for real-time systems', *Software, IEEE*, Vol. 8, No. 3, pp.62–72.

Varela, M., Cayssials, R., Ferro, E. and Boemo, E. (2012) 'Real-time scheduling coprocessor for NIOS II processor', in *Programmable Logic (SPL), VIII Southern Conference on*, pp.1–6.

Vaucher, J.G. and Duval, P. (1975) 'A comparison of simulation event list algorithms', *Commun. ACM*, Vol. 18, No. 4, pp.223–230.

Zhuang, X. and Pande, S. (2006) 'A scalable priority queue architecture for high speed network processing', in *INFOCOM, 25th IEEE International Conference on Computer Communications. Proceedings*, pp.1–12.

Cache Design for Mixed Criticality Real-Time Systems

N G Chetan Kumar, Sudhanshu Vyas, Ron K. Cytron*, Christopher D. Gill*, Joseph Zambreno and Phillip H. Jones
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA
Email: {ckng, spvyas, zambreno, phjones}@iastate.edu
* Department of Computer Science and Engineering, Washington University, St. Louis, MO, USA
Email: {cytron, cdgill}@cse.wustl.edu

***Abstract*—Shared caches in mixed criticality systems are a source of interference for safety critical tasks. Shared memory not only leads to worst-case execution time (WCET) pessimism, but also affects the response time of safety critical tasks. In this paper, we present a criticality aware cache design which implements a Least Critical (LC) cache replacement policy, where a least recently used non-critical cache line is replaced during a cache miss. The cache acts as a Least Recently Used (LRU) cache if there are no critical lines or if all cache lines are critical in a set. In our design, data within a certain address space is given higher preference in the cache. These critical address spaces are configured using critical address range (CAR) registers. The new cache design was implemented in a Leon3 processor core, a 32bit processor compliant with the SPARC V8 architecture. Experimental results are presented that illustrate the impact of the Least Critical cache replacement policy on the response time of critical tasks, and on overall application performance as compared to a conventional LRU cache policy.**

I. Introduction

Cache memories greatly improve the overall performance of processors by bridging the increasing gap between processor and memory speed. In real-time systems, it is necessary to accurately estimate the worst-case execution time (WCET) of a task to ensure tasks are completed within certain deadlines. The unpredictable behavior of shared caches complicates WCET analysis [14], which leads to overestimation of WCET and decreases processor utilization. Various techniques such as cache locking and partitioning have been proposed to make shared caches more predictable in real-time systems. Higher predictability is often achieved at the cost of reduced application performance.

In mixed criticality real-time systems, where tasks of different criticalities are executed on the same platform, it is necessary to ensure the timing constraints of critical tasks are met under all conditions, while trying to maximize average processor utilization. To achieve this, we need to mitigate the interference of lower criticality tasks on the timing behavior of higher criticality tasks. Shared caches in mixed criticality systems is one such source of interference that can increase the response time of critical tasks.

In this paper, we present a cache design for mixed criticality real-time systems in which critical task data is least likely to be evicted from cache during a cache miss. We assume data is either critical or non-critical. An extension of the least recently used (LRU) cache replacement policy, called Least Critical (LC), is implemented as a part of our proposed cache design. In our LC cache replacement policy, data from certain address spaces are given preference in the cache. These critical address spaces are defined by a critical address range (CAR), which is configurable during run-time. Our design enables fine grained control over classifying task data as critical, and allows run-time configuration of a critical address space to better manage cache performance.

II. Related Work

In the context of shared caches in real-time systems, various cache locking and partitioning schemes have been proposed to improve predictability and overall performance of real-time tasks. In cache partitioning, a portion of the cache is assigned to a task and the task is restricted to only use that assigned partition. This removes inter-task cache conflicts. Software based partitioning techniques such as [5], [2], [15], [6] require changing from address to cache-line mapping to eliminate inter-task conflicts, which makes it difficult for system-wide application. The use of hardware based techniques [8], [12] is limited by fixed partition sizes and coarse grained configurability, which may reduce cache utilization. Cache locking allows certain lines of the cache to be locked in place, which enables accurate calculation of memory access times. While cache locking [13], [1], [11] provides fine grained control over task data, it will lead to poor utilization when data does not fit in the cache [13]. Dynamic cache locking also increases overhead and can affect overall task performance, if cache lines are locked unnecessarily.

More recently, cache management techniques for mixed criticality real-time systems have been proposed to improve predictability and performance of critical tasks. PRETI, a partitioned real time cache scheme was presented in [7], where a critical task is assigned a private cache space to reduce inter-task conflict. The cache lines not claimed by a task are marked as shared, and can be used by all tasks. [9] proposed a cache management framework for multi-core architectures to provide a deterministic cache hit rate for a set of hot pages used by a task. Cache scheduling and locking techniques to manage shared caches within the MC^2 scheduling framework [10] was presented in [4].

In our proposed cache design, we allow fine grained control over task data by providing a mechanism to store critical data in separate address spaces. This enables better cache utilization as the non-critical cache lines are shared by all tasks. By placing critical task data in separate address ranges, which are given preference in cache, the overhead involved in locking/unlocking individual cache lines is also eliminated. Our design provides the flexibility to change critical address ranges at run-time, which enables applications to better utilize cache.

III. Criticality Aware Cache Design

We present a criticality aware cache design for shared caches in mixed criticality real-time systems to reduce inter-task conflicts and decrease response time of critical tasks. The core of the design is a new cache replacement policy, called Least Critical, which is described in detail next.

A. Least Critical Cache

Our Least Critical cache (LC cache) replacement policy targets set associative shared caches in mixed criticality real-time systems. The LC policy is an extension of a conventional least recently used (LRU) cache. For each cache set, we keep a count of lines which have data from critical address range. We also maintain LRU order for critical and non-critical lines in each cache set. During a cache hit, the LRU order of either critical or non-critical lines in the cache set is modified based on the line being accessed. When there is a cache miss, the line to be replaced is selected based on the following order: 1. Empty cache line. 2. Least recently used non-critical cache line. 3. Least recently used critical cache line, if all the lines in a cache set are critical.

During a cache miss, if the data accessed or evicted is from a critical address range, then the number of critical cache lines in that set is updated. During a cache miss, a critical cache line gets evicted only when all lines in a cache set are critical. The LC cache replacement policy acts as LRU, if all the lines in a cache set are from a critical address range or if there is no critical data in a cache set. A working example of the LC cache policy is shown in Figure 1.

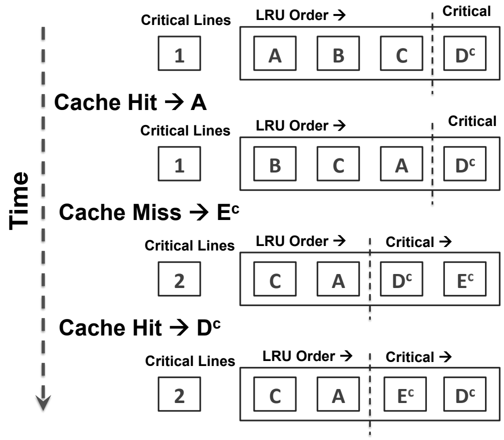


Fig. 1. A working example of our Least Critical cache replacement policy. The LRU order, for both critical and non-critical data, is maintained using a state transition table. ^{C} indicates critical cache lines.

B. Hardware Implementation

Figure 2 depicts a high level block diagram of the LC cache architecture. It is composed of four primary components: 1) CAR Compare, 2) Access History, 3) Tag Compare, and 4) Data Control.

Critical Address Range (CAR) Compare. CAR registers are used to identify critical data based on memory address. An application configures these memory-mapped registers to specify where critical data resides in memory. The architecture supports the use of multiple CAR registers sets, each defines an address space for holding critical data. The memory address is compared with CAR registers during cache access to identify critical cache lines. The implementation of our architecture additionally allows dynamically switching between our LC cache policy and a conventional LRU policy at run-time.

Access History. The LRU order of critical and non-critical lines along with the number of critical lines is maintained as an access history, which is updated on every memory access. In addition to the bits used to store the LRU order for each set, $\log A + 1$ bits are required to track the number critical lines in each set, where A is the cache set associativity.

Tag Compare. Generates cache hit/miss signals by comparing requested memory addresses with tag bits associated with each cache line.

Data Control. Provides an interface to the CPU to read/write data from cache or main memory.

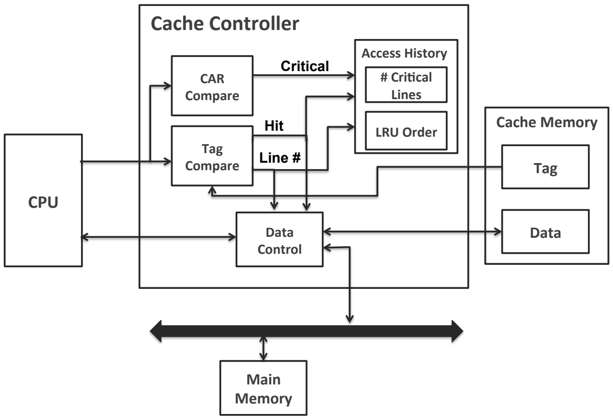


Fig. 2. High level block diagram of the Least Critical (LC) Cache Controller.

The design allows run-time modification of the critical address ranges. Since modifying CAR registers at run-time impacts the coherency of the critical-cache-line count, a mechanism is needed to restore coherency. For maintaining coherency, a *soft-reset* mechanism is used to clear the critical data line count of each cache set to zero. This is accomplished by the application writing to a specific memory mapped register. A *soft-reset* of the cache should be performed before updating CAR registers.

Switching between LC and LRU. The LC cache can be reverted to behaving as LRU by clearing the CAR registers and triggering a *soft-reset* of the critical line counts. After a *soft-reset* of the LC cache, existing critical cache lines default to most recently used non-critical cache lines.

Application-level usage model of LC cache. To make use of the LC cache, the application should tag critical data variables and the compiler should place those variables in a separate section of memory. In GCC, this could be accomplished using the ”*section*” attribute, which specifies that a variable resides in a particular section. $Ex.\ int\ cdata\ \_\_attribute\_\_\ ((section(”critical”)));$

Frequently used memory pages e.g., from the application could also be made critical by configuring CAR registers.

When compared to cache locking, our technique avoids the run-time overhead of locking mechanisms and allows critical data to stay in cache. We also provide graceful degradation when critical data is larger than the cache size, since the cache acts as LRU when all the lines in a set are critical.

IV. Evaluation Methodology

A. Hardware Platform and Configuration

The LC cache replacement policy was evaluated on a XUPV5-LX110T, a Xilinx FPGA development and evaluation platform that features a Virtex-5 FPGA, 256 MB RAM (DDR2), JTAG and UART interfaces. Leon3, a 32bit soft-core processor compliant with the SPARC V8 architecture, was used to implement our cache design. Leon3 features a 7-stage pipeline and separate instruction and data caches. In this paper, we limit the analysis to data cache only. Our cache design was implemented as a L1 data cache in the Leon3 processor running at 33MHz with no memory management unit (MMU). For our evaluation, we used a 4-way set associative data cache of size 4KB with 16 bytes/line. The LRU cache supported by Leon3 was used as the baseline to compare the performance of our LC cache design. A non-intrusive hardware cache profiler was designed to accurately measure the performance of the data cache unobtrusively. The profiler could be configured to measure data cache hits and misses for each task, along with overall application statistics. The profiler sends the data offline to a server through a UART interface.

TABLE I. CHARACTERISTICS OF BENCHMARK PROGRAMS USED TO EVALUATE OUR CACHE DESIGN.

Task Name	Code Size (bytes)	Data Size (bytes)	Execution Time (ms)[1]
CRC	1216	1048	0.16
FDCT	2940	132	0.49
FIR	572	2948	54.06
Compress	3316	2416	18.52
IPC	1092	256 - 8192	0.11 - 4.87

[1] Execution time for task running alone.

B. Workload and Metrics

To evaluate the performance of our cache design, we used a set of five real-time benchmark programs. The critical task was an inverted pendulum controller (IPC). We varied the resolution of the controller so that its critical data (matrix used in the control computation) ranged from 256 to 8K bytes. Background tasks were drawn from the WCET project [3] and consisted of CRC, FDCT (discrete cosine), FIR (finite impulse response filter), and compress. The characteristics of these programs are shown in Table I. FreeRTOS, an open source kernel designed for embedded real-time systems, was used to run the benchmark applications on Leon3. FreeRTOS was configured to execute a preemptive priority based scheduling algorithm. The cache miss rate of both the critical task and the overall application was measured for LC and LRU cache replacement policies.

V. Results and Analysis

To evaluate the performance of data cache, the benchmark programs were executed using rate monotonic (RM) scheduling. The period of non-critical tasks were kept constant at 200ms and the experiment was conducted for three different critical task periods (50ms, 100ms, 200ms). Figure 3 shows the cache miss rates for our critical task, as the size of its critical data increases. With the LC policy, its references are favored and we generally see a marked improvement over the LRU policy for the critical task. When the size of the critical task's bytes reach the cache size (4K), we see an increase in the critical task miss rate even for the LC replacement policy. This is because the critical tasks' references are due to matrix multiplications, which will incur misses once the cache size is exceeded. Finally at 8K critical data bytes, we have exceeded the size of the 4K-byte cache. Then, LRU and LC are indistinguishable for the critical task.

When using LRU, the miss rate of the critical task increases with its period as shown in Figure 3. This is due to inter-task interference increasing when the critical task is not executed often. In comparison, the LC cache shows a predictable miss rate for the critical task while performing 40% - 70% better than the LRU cache. The LC cache reduces the impact of inter-task conflicts on the critical task by giving preference to that task's critical data.

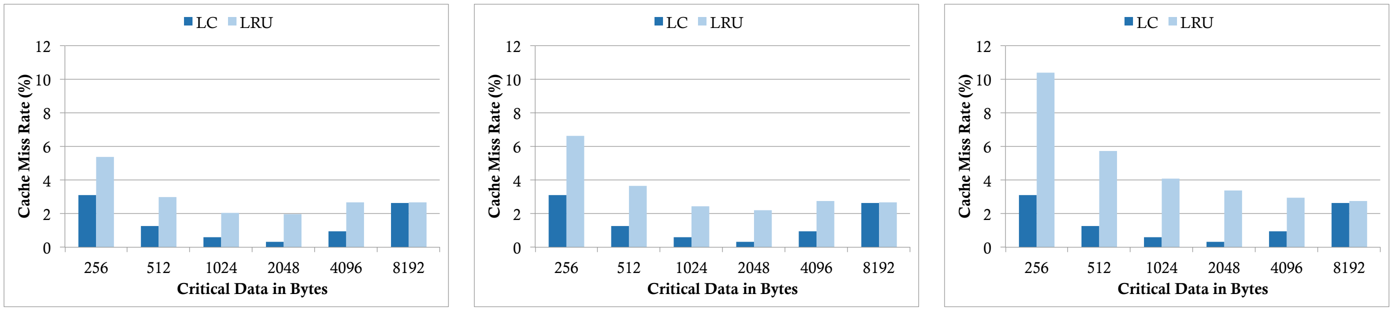
The cache miss rates for the overall application (critical and noncritical tasks) is shown in Figure 4. Overall performance is not adversely affected by LC's favoring the critical task, until we reach the size of the L1 cache at 4K bytes. Comparing across the figures, at 4K, we see reason to favor the critical task, improving its execution time at the expense of the noncritical tasks. However, at 8K, favoring the critical task benefits neither that task nor any other task. LRU would be a better choice at this point.

VI. Conclusion

In this paper, we presented a criticality aware cache design for mixed criticality real-time systems. A new cache replacement policy, called Least Critical, was proposed where data within a critical address space is given higher preference in the cache. Our design enables fine grained control over classifying task data as critical, and allows run-time configuration of a critical address space to better manage cache performance. Our experimental results show that the cache miss rate of a critical task is reduced by up to 70% when using LC cache in comparison with LRU cache. We also show that increasing critical data size deteriorates the performance of non-critical tasks. In order to manage overall performance of the application, we recommend limiting critical data size to less than cache size, or switching to a LRU cache policy at run-time when this threshold is surpassed. Avenues for future work include, 1) extending the analysis to instruction cache and 2) enabling support for data with multiple criticality levels.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under award CNS-1060337, and by the Air Force Office of Scientific Research (AFOSR) under award FA9550-11-1-0343.

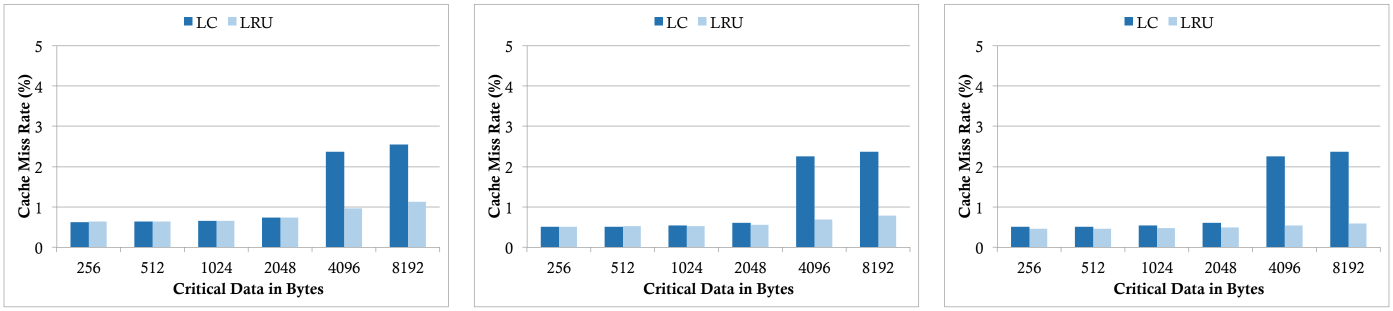


(a) Critical Task Period = 50 ms

(b) Critical Task Period = 100 ms

(c) Critical Task Period = 200 ms

Fig. 3. Critical Task: Performance of LC cache when compared to LRU cache. Critical task run with CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms



(a) Critical Task Period = 50 ms

(b) Critical Task Period = 100 ms

(c) Critical Task Period = 200 ms

Fig. 4. Overall Application: Performance of LC cache when compared to LRU cache. Critical task run with CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms

References

[1] A. Asaduzzaman, F. N. Sibai, and A. Abonamah, “A dynamic way cache locking scheme to improve the predictability of power-aware embedded systems,” in *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*. IEEE, 2011.

[2] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, “Impact of cache partitioning on multi-tasking real time embedded systems,” in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA’08. 14th IEEE International Conference on*. IEEE, 2008, pp. 101–110.

[3] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, “The Mälardalen WCET benchmarks – past, present and future.” OCG, 2010, pp. 137–147.

[4] C. J. Kenna, J. L. Herman, B. C. Ward, and J. H. Anderson, “Making shared caches more predictable on multicore platforms,” in *Euromicro Conference on Real-Time Systems*, 2013.

[5] H. Kim, A. Kandhalu, and R. Rajkumar, “A coordinated approach for practical os-level cache management in multi-core real-time systems,” in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013, pp. 80–89.

[6] J. Kim, I. Kim, and Y. I. Eom, “Code-based cache partitioning for improving hardware cache performance,” in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ACM, 2012, p. 42.

[7] B. Lesage, I. Puaut, and A. Seznec, “Preti: Partitioned real-time shared cache for mixed-criticality real-time systems,” in *Proceedings of the 20th International Conference on Real-Time and Network Systems*. ACM, 2012, pp. 171–180.

[8] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, “Enabling software management for multicore caches with a lightweight hardware support,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 14.

[9] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-time cache management framework for multi-core architectures,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 45–54.

[10] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, “Mixed-criticality real-time scheduling for multicore systems,” in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1864–1871.

[11] V. Suhendra and T. Mitra, “Exploring locking & partitioning for predictable shared caches on multi-cores,” in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 300–303.

[12] Y. Tan and V. Mooney, “A prioritized cache for multi-tasking real-time systems,” in *Proc., SASIMI*, 2003.

[13] X. Vera, B. Lisper, and J. Xue, “Data cache locking for tight timing calculations,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 1, p. 4, 2007.

[14] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, “The worst-case execution-time problem—overview of methods and survey of tools,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[15] X. Zhang, S. Dwarkadas, and K. Shen, “Towards practical page coloring-based multicore cache management,” in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 89–102.