2016

Hardware architecture support for mixed criticality and real-time systems

Chetan Kumar Nagamangala Govindaiah
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd
Part of the Computer Engineering Commons

Recommended Citation
Nagamangala Govindaiah, Chetan Kumar, 'Hardware architecture support for mixed criticality and real-time systems' (2016). Graduate Theses and Dissertations. Paper 15087.

This Dissertation is brought to you for free and open access by the Graduate College at Digital Repository @ Iowa State University. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Hardware architecture support for mixed criticality and real-time systems

by

Chetan Kumar Nagamangala Govindaiah

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Phillip H. Jones, Major Professor
Joseph A. Zambreno
Arun K. Somani
Manimaran Govindarasu
Nicola Elia

Iowa State University
Ames, Iowa
2016

Copyright © Chetan Kumar Nagamangala Govindaiah, 2016. All rights reserved.
DEDICATION

To my loving parents.

Govindaiah and Bhagyamma
TABLE OF CONTENTS

LIST OF TABLES vi
LIST OF FIGURES vii
ACKNOWLEDGEMENTS xii
ABSTRACT xiii

CHAPTER 1. INTRODUCTION 1
  1.1 Summary of Contributions 4
  1.2 Organization 4

PART I  HYBRID PRIORITY QUEUE AND SCHEDULER ARCHITECTURE 6

CHAPTER 2. BACKGROUND AND MOTIVATION 7
  2.1 Related Work 8
    2.1.1 Hardware Priority Queues 8
    2.1.2 Hardware Schedulers 9

CHAPTER 3. HYBRID PRIORITY QUEUE ARCHITECTURE 11
  3.1 Background 11
  3.2 Overview 11
  3.3 Hardware Priority Queue 13
    3.3.1 Enqueue 13
3.3.2 Dequeue ................................................................. 14
3.3.3 Decrease-Key and Delete ......................................... 15
3.4 Hybrid Priority Queue Management ............................... 15
  3.4.1 Enqueue .......................................................... 16
  3.4.2 Dequeue ......................................................... 17
3.5 Evaluation Methodology ............................................. 18
3.6 Results and Analysis ................................................ 21

CHAPTER 4. HARDWARE SCHEDULER ...................................... 24
  4.1 Overview .......................................................... 24
  4.2 Architecture ...................................................... 25
  4.3 Modes of Operation ............................................... 25
  4.4 Evaluation Methodology .......................................... 26
  4.5 Results and Analysis ............................................. 27

PART II  CACHE ARCHITECTURE FOR MIXED CRITICALITY SYSTEMS  31

CHAPTER 5. INTRODUCTION TO MIXED CRITICALITY SYSTEMS ....... 32
  5.1 Background ...................................................... 32
  5.2 Motivation ....................................................... 33
  5.3 Related Work .................................................... 35

CHAPTER 6. CRITICALITY AWARE CACHE DESIGN ..................... 38
  6.1 Least Critical Cache ............................................ 38
    6.1.1 Hardware Implementation .................................. 39
    6.1.2 Application-level usage model of LC cache ............. 42
  6.2 Impact on WCET Analysis ....................................... 43
    6.2.1 LC Cache Semantics ....................................... 43
    6.2.2 Cache analysis of a single task ......................... 44
6.2.3 Analysis of inter-task cache conflicts ......................... 45
6.3 Evaluation Methodology ........................................... 46
6.3.1 Hardware Platform and Configuration ......................... 46
6.3.2 Workload and Metrics ......................................... 48
6.4 Results and Analysis ............................................. 48
6.4.1 Experiment 1 - Two-task Setup ................................. 49
6.4.2 Experiment 2 - Five-task Setup ................................. 50
6.4.3 Hardware Resource Utilization ................................. 56
6.4.4 Other Considerations ......................................... 57

CHAPTER 7. DYNAMIC CACHE MANAGEMENT - A CASE STUDY 59
7.1 Introduction ...................................................... 59
7.2 Multi Criticality Workload .................................... 59
7.3 Results and Analysis ........................................... 60
7.3.1 Key Observations ........................................... 68
7.4 Hardware Monitor Infrastructure ............................... 69
7.5 Runtime Reconfiguration of the LC Cache ......................... 70

CHAPTER 8. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS 73
8.1 Conclusions ...................................................... 73
8.2 Future Research Directions ................................... 74
8.2.1 Extend LC Cache Analysis to Instruction Cache .......... 74
8.2.2 Application of LC Cache in Real-Time Scheduling ........ 74
8.2.3 Heuristics and Search Algorithms for Dynamic Cache Management ... 75
8.2.4 Towards a Criticality Aware Adaptive Hardware Platform ... 75

BIBLIOGRAPHY .......................................................... 77
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Priority increment distributions used in our evaluation.</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>FPGA resource utilization of the proposed priority queue design for different queue sizes.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>FPGA resource utilization of shift register and systolic array based priority queue architectures [57] in comparison with proposed priority queue design.</td>
<td>23</td>
</tr>
<tr>
<td>6.1</td>
<td>Characteristics of benchmark programs used to evaluate our LC cache design.</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Maximum observed cache miss rate of the critical task when using a LC cache in comparison with a LRU cache. 4KB 4-way set associative cache.</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>FPGA resource utilization of the proposed cache design in comparison with LRU cache for different cache sizes.</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary of the task set adapted from the generic avionics software specification described in [52].</td>
<td>60</td>
</tr>
<tr>
<td>7.2</td>
<td>Mapping of each task to mode of operation.</td>
<td>61</td>
</tr>
<tr>
<td>7.3</td>
<td>Normalized cache miss rate of critical data for different modes of operation. LRU cache is used as a baseline for comparison.</td>
<td>71</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1 Hardware architecture support for mixed critical and real-time systems.

(Blocks with dotted lines indicate the parts of the platform where I contribute.)

Figure 2.1 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.

Figure 3.1 Array representation of a binary heap.

Figure 3.2 A high level block diagram of the hardware-base priority queue interface.

Figure 3.3 The hardware priority queue architecture.

Figure 3.4 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.

Figure 3.5 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. In worst case there will be $log(n)$ swap operations.

Figure 3.6 (a) Memory mapped interface provides access to priority queue elements stored in block RAM. (b) Virtual address space showing extended priority queue.
Figure 3.7  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.

Figure 3.8  Steps of dequeue operation in hybrid mode. In this example we assume that the first 3 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.

Figure 3.9  FPGA-based evaluation platform.

Figure 3.10  Performance comparison between the software and hybrid implementation of a priority queue. Evaluated using the Classic Hold Model, for 4 different priority increment distributions.

Figure 3.11  Performance comparison between the software and hybrid implementation of a priority queue. Evaluated using the Up/Down Model, for 4 different priority increment distributions.

Figure 3.12  Comparing FPGA look-up table utilization of the proposed priority queue design against shift register and systolic array based priority queue architectures[57] for different queue sizes. Flip-flop utilization also shows a similar trend.

Figure 4.1  A high level architecture diagram of the hardware scheduler along with the custom instruction interface.

Figure 4.2  Performance of the software scheduler compared with hardware scheduler for task sizes less than or equal to 255.
Figure 4.3 Performance of software scheduler compared with hybrid scheduler for task sizes greater than 255.

Figure 4.4 Variation in execution times of software and hardware scheduler.

Figure 4.5 Variation in execution times of software and hybrid scheduler.

Figure 6.1 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.

Figure 6.2 High level block diagram of the Least Critical (LC) Cache Controller. Dotted blocks are registers which can be configured through software instructions.

Figure 6.3 High level block diagram of the non-intrusive hardware cache profiler.

Figure 6.4 Critical Task: Cache performance of LC cache when compared to LRU cache. Critical task: Inverted Pendulum Controller (IPC) being run pair-wise with CRC, FDCT, Compress, and FIR.

Figure 6.5 Overall Application: Cache performance of LC cache when compared to LRU cache. Critical task: Inverted Pendulum Controller (IPC) being run pair-wise with CRC, FDCT, Compress, and FIR.

Figure 6.6 Critical Task: Performance of LC cache when compared to LRU cache. Critical task run with four non-critical tasks: CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms.

Figure 6.7 Overall Application: Performance of the LC cache when compared to the LRU cache. Critical task run with four non-critical tasks: CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms.

Figure 6.8 Critical Task: Performance of the LC cache when compared to the LRU cache for different cache sizes.

Figure 6.9 Overall Application: Performance of the LC cache when compared to the LRU cache for different cache sizes.
Figure 6.10 Decrease in maximum observed execution time (MOET) of the critical task when using a LC cache in comparison with a LRU cache. 4KB 4-way set associative cache.

Figure 7.1 Mode - Surveillance, Cache Size: 4K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.2 Mode - Surveillance, Cache Size: 8K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.3 Mode - Surveillance, Cache Size: 4K Bytes: Normalized task execution times with LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.4 Mode - Surveillance, Cache Size: 8K Bytes: Normalized task execution times with LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.5 Mode - Tracking, Cache Size: 4K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.6 Mode - Tracking, Cache Size: 8K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.7 Mode - Tracking, Cache Size: 4K Bytes: Normalized task execution times with LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.
Figure 7.8 Mode - Tracking, Cache Size: 8KBytes: Normalized task execution times with LC cache when compared to LRU cache. In \textit{Config} – \textit{C}, only critical tasks’ data tagged as critical. In \textit{Config} – \textit{CE}, critical and essential tasks’ data tagged as critical.

Figure 7.9 Mode - Engage, Cache Size: 4KBytes: Performance of LC cache when compared to LRU cache. In \textit{Config} – \textit{C}, only critical tasks’ data tagged as critical. In \textit{Config} – \textit{CE}, critical and essential tasks’ data tagged as critical.

Figure 7.10 Mode - Engage, Cache Size: 8KBytes: Performance of LC cache when compared to LRU cache. In \textit{Config} – \textit{C}, only critical tasks’ data tagged as critical. In \textit{Config} – \textit{CE}, critical and essential tasks’ data tagged as critical.

Figure 7.11 Mode - Engage, Cache Size: 4KBytes: Normalized task execution times with when compared to LRU cache. In \textit{Config} – \textit{C}, only critical tasks’ data tagged as critical. In \textit{Config} – \textit{CE}, critical and essential tasks’ data tagged as critical.

Figure 7.12 Mode - Engage, Cache Size: 8KBytes: Normalized task execution times with LC cache when compared to LRU cache. In \textit{Config} – \textit{C}, only critical tasks’ data tagged as critical. In \textit{Config} – \textit{CE}, critical and essential tasks’ data tagged as critical.

Figure 7.13 A high level block diagram of the hardware cache monitor infrastructure.
ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor Dr. Phillip Jones for encouraging me to pursue my Ph.D. and providing this great opportunity. I would like to thank him for giving me the freedom to explore my ideas while providing excellent guidance and constructive feedback throughout the course of this study. I greatly appreciate his caring nature towards his students and have always admired his patience and his ability to listen reflectively.

I would also like to thank Dr. Joseph Zambreno, who was like a co-advisor to me and provided critical feedback on my research work throughout my Ph.D years. I would like to thank my committee members Dr. Arun Somani, Dr. Manimaran Govindarasu and Dr. Nicola Elia for their guidance and feedback during the final years of my Ph.D. I would also like to thank all the members of Reconfigurable Computing Laboratory for their support and good times in the lab.

I have been very fortunate to have great roommates and friends, who made my journey at Iowa State University an exciting and fun experience. I would like to thank you all for your support, encouragement and all the pep talks you gave during difficult times.

Lastly, I would like to thank my parents and family members for all the sacrifices they have made over the years to provide me the opportunity to pursue graduate education. This would not have been possible without your encouragement and continued support.
ABSTRACT

The use of hardware-based solutions for accelerating real-time and embedded system applications is limited by the scarceness of hardware resources. By their nature, being limited by the silicon area available, hardware solutions cannot scale in size as easily as their software counterparts. I assert a hardware-software co-design approach is required to elegantly overcome these limitations. In the first part of this dissertation, I demonstrate the feasibility of this approach by presenting a new hybrid priority queue architecture, which can be managed in hardware and/or software. As an application of this hybrid architecture, I then present a scalable task scheduler for real-time systems that reduces scheduler processing overhead and improves timing determinism of the scheduler. Performance evaluations of our Field Programmable Gate Array (FPGA)-based system-on-chip prototype shows up to a 90% reduction in scheduling overhead and a 98% decrease in scheduler execution time variation, when the scheduler is managed by hardware as compared to software.

As recent trends in real-time and embedded systems show, applications of different criticality are being executed on a single hardware platform driven by the need to reduce size, cost and power requirements. In these mixed criticality systems, it is necessary to ensure the non-critical tasks do not interfere with the timing behavior of safety-critical tasks. In the second part of this dissertation, I investigate hardware architectures that are aware of application criticality and can adapt to changing operating conditions to provided better timing guarantees for critical tasks, while improving overall resource utilization. In support of this approach, I present a criticality aware cache architecture for mixed criticality real-time systems. As a part of the proposed cache design, a new cache replacement policy called Least Critical (LC) is presented, where critical tasks’ data is least likely to be evicted from the cache. Experimental results illustrate the impact of the LC cache replacement policy on the response time of critical tasks, and on overall application performance.
CHAPTER 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 Ariel Vehicles (UAVs) is an example of where such need exists. The size of these vehicles have rapidly decreased, while the capabilities users wish to deploy continue to explode. As recent as June of 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 [16]. 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) [21][26] 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, due transistors approaching atomic scales and thermal constraints [34]. 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 [23, 63], to numerical simulation [66], to control systems [55, 41, 56].

I assert that the boundaries of software and hardware must be reexamined and I believe a fruitful realm for research is the hardware-software co-design of functionality that has been traditionallly implemented in software. Such a co-design is needed to balance the cost of dedicating limited silicon resources for high-performance fixed hardware functionally, with the flexibility and scalability offered by software. Additionally, I claim seamless migration between software and hardware implemented functionality is required to allow systems to adapt to the dynamic needs of applications. In the first part of this dissertation, I explore how hardware-software co-design of functionality can overcome the size and scalability limitation of hardware-only solutions. To demonstrate the feasibility of this approach, I present a hybrid priority queue.
architecture that can be managed in both hardware and software. The hybrid priority queue architecture is then evaluated in the context of real-time scheduling. It is shown that the co-design approach combines the benefit of high-performance fixed hardware functionality with the flexibility of software solutions.

In recent years, there has been increasing demand to reduce cost and power requirements of embedded and real-time systems in areas such as avionics and automotive control. To meet these demands, functionality of different criticality are being implemented on a shared hardware platform [30], which are called mixed criticality systems. In mixed criticality systems, it is necessary to provide temporal and spacial isolation guarantees for critical tasks to ensure their timing constraints are met under all conditions. Traditional priority-based scheduling techniques, which provide temporal isolation, assume all tasks are equally critical. To ensure schedulability of every task, conservative estimates of worst case resource demands should be taken, which leads to poor resource utilization under normal operating conditions, as the worst case behavior occurs rarely [85]. Under overload conditions, these techniques cannot guarantee non-interference from lower criticality tasks on the timing behavior of higher criticality tasks. Enforcing criticality as priority may avoid criticality inversion, but may not yield optimal priority assignment to maximize processor utilization. Hence, a research goal in mixed criticality systems is to schedule resources to maximize average resource utilization and enforce task’s criticality when a system is overloaded, to ensure no higher criticality activity’s constraint is violated because of the actions of a lower-criticality activity [30].

Several mixed criticality task models and scheduling algorithms [82, 22, 10, 45, 24] have been proposed to address this issue with respect to CPU scheduling. While such work is important, the CPU is neither the main performance bottleneck nor the most unpredictable aspect of many modern computing platforms. The storage hierarchy (from registers to pages) often can be the limiting performance factor and source of unpredictability for many applications [85]. Inter-task cache conflicts in mixed criticality systems is one such source of unpredictability that can affect the performance and response time of critical tasks and leads to increased WCET estimation pessimism. In the second part of this dissertation, I propose a cache architecture for mixed criticality systems to reduce inter-task cache conflicts and improve the response time of
critical tasks. The implementation of the proposed cache architecture additionally provides the flexibility to change the cache configuration including the cache replacement policy at runtime.

The ability to monitor application performance during runtime enables a system to re-assess computation requirements and change its configuration dynamically to better utilize system resources [8]. Previous research [11] has shown platforms that can monitor task execution times enable the use of adaptive scheduling algorithms, which improves schedulability of mixed criticality systems. I investigate the use of lightweight non-intrusive hardware monitors to observe the performance of the cache, and to provide runtime feedback. I propose mechanisms that dynamically change a cache configuration to improve the cache utilization of systems where computation and other resource requirements can change during runtime.

A high level block diagram of my envisioned hardware architecture support for mixed criticality and real-time systems is shown in Fig 1.1. The blocks with dotted lines indicate where I contribute.

![Hardware architecture support for mixed critical and real-time systems. (Blocks with dotted lines indicate the parts of the platform where I contribute.)](image)
1.1 Summary of Contributions

In Chapter 3:

- A hardware accelerated binary min heap design is presented, which 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. [44]

- A scalable hardware-software priority queue architecture that enables fast and low-overhead transitions of queue management between hardware and hybrid software-hardware modes of operation is proposed and evaluated. [44]

In Chapter 4, a hybrid software-hardware scheduler architecture that reduces scheduling overhead and improves predictability is presented. [44]

In Chapter 6:

- A new cache replacement policy called Least Critical (LC), in which critical data is given preference in the cache is presented. [43]

- A configurable cache architecture for mixed criticality systems that reduces the response time and improves predictability of critical tasks is presented. [43]

In Chapter 7, the feasibility of using lightweight hardware cache monitors to provide runtime feedback is evaluated and mechanisms for dynamic cache management are proposed.

1.2 Organization

This dissertation is divided into two parts. In Part I, the use of a hardware-software co-design approach to overcome limitations of purely hardware solutions is explored. Chapter 2 provides the motivation for a co-design approach along with the review of related work on hardware accelerated priority queues and hardware schedulers. My hardware-software priority queue architecture, hardware implementation, and evaluation methodology is presented in Chapter 3. Chapter 4 details the hardware scheduler architecture, which uses the priority
queue design from Chapter 3. Part II of this dissertation explores the use of criticality aware hardware in mixed criticality systems. Chapter 5 provides an introduction to mixed criticality systems and a review of real-time cache management techniques. The design, implementation and evaluation of the Least Critical (LC) cache architecture is presented in Chapter 6. A case study on dynamic cache management is described in Chapter 7. Chapter 8 concludes this dissertation and presents possible directions for future research.
PART I

HYBRID PRIORITY QUEUE AND SCHEDULER
ARCHITECTURE
CHAPTER 2. BACKGROUND AND MOTIVATION

In the past, many researchers have shown the benefit of migrating functionality from software to hardware. Implementing functionality in hardware improves performance, predictability, and application response time. However, lack of flexibility in hardware solutions limit their widespread use. There have been efforts to overcome this limitation by making the hardware configurable [59, 42]. For example, [42] implemented a configurable hardware scheduler that provided support for three scheduling disciplines, configurable during run-time. However, the maximum number of tasks supported is fixed once the hardware is fabricated. I believe a hardware-software co-design approach can overcome the size and scalability limitation of hardware solutions. To demonstrate the feasibility of this approach, I present a hybrid priority queue architecture which can be managed in both hardware and software. I then evaluate this hybrid 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 a 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 [64] [4].

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 associ-
Figure 2.1: 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.

| a) Worst case | T1 | T2 | T3 | T4 | T5 |
| b) Average case | T1 | T2 | T3 | T4 | T5 | Idle |
| c) Best Case | T1 | T2 | T3 | T4 | T5 | Idle |

- $T_i$ - Task
- $-$ - Scheduler Execution Time
- Time

Ated 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. 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 2.1 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 dissertation, I examine how 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 these efforts are increased CPU utilization, better system predictability, and finer schedule and timing resolution.

## 2.1 Related Work

### 2.1.1 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 [57, 12, 33]. [57] 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. [12] 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 [33] 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 [88], 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 [19], where a shift-register based priority queue [57] was extended by appending a software binary heap. [13] presented an exception based mechanism for handling overflows in hardware priority queue, where additional data is moved to secondary storage by the exception handler.

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 my 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 utilization of the proposed priority queue increases logarithmically with the queue size and avoids complex pipelining logic.

### 2.1.2 Hardware Schedulers

Several architectures [4, 17, 72, 42, 27, 40] 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 [4]. The scheduler of FASTHARD can handle 256 tasks and 8 priority levels. The Spring scheduling coprocessor [17] was built to accelerate scheduling algorithms used in the Spring kernel [74], which was used to perform feasibility analysis of the schedule. [42] implemented a configurable hardware scheduler that provided support for three scheduling disciplines, configurable during runtime. A slack stealing scheduling algorithm was implemented in hardware [72] to support scheduling of tasks (periodic and aperiodic) and to reduce scheduling overhead. [62] implemented most of the μITRON 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 [27], 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 [40]. The RTM supports static priority-based scheduling and is implemented as an on-chip peripheral that communicates with the processor through a memory mapped interface. The SERRA run-time scheduler synthesis and analysis tool was presented in [58]. The tool automatically generated a run-time hardware-software scheduler from system level specification. A hardware-software kernel was presented in [60], which implemented a scheduling co-processor running earliest deadline first scheduling algorithm. A Hardware Real-Time Scheduler Coprocessor (HRTSC) architecture for NIOS II processor was described in [78], which could be configured to run any priority based scheduling discipline.

One of the limitations of the hardware schedulers described above is that, once deployed, they can only support a fixed number of tasks. My 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.
CHAPTER 3. HYBRID PRIORITY QUEUE ARCHITECTURE

3.1 Background

A priority queue (PQ) is an abstract data structure in which each element has an associated priority. The PQ at minimum supports two operations: 1) Enqueue - which inserts a new element with an associated priority into the queue, and 2) Dequeue - which removes the element with highest priority from the queue. 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 as shown in Figure 3.1, 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.

![Array representation of a binary heap](image)

Figure 3.1: Array representation of a binary heap.

3.2 Overview

Here I 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 can 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 utilize 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.

![Figure 3.2: A high level block diagram of the hardware-base priority queue interface.](image)

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 3.2 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.
3.3 Hardware Priority Queue

A high level architecture diagram for the priority queue is shown in Figure 3.3. 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 [12, 33]. The address decoder generates addresses and control signals for the BRAM blocks. Queue operations in hardware mode are explained in detail next, using a min-heap example, where a lower key value corresponds to a higher priority.

3.3.1 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 behavior 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, my 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 3.3, inserts a new element in constant time. This is similar to the shift-register priority queue described in [57]. 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 3.4. 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.

### 3.3.2 Dequeue

Figure 3.5 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 $O(\log n)$.

![Figure 3.5: Steps of dequeue operation in hardware mode.](image)

b) New root is swapped with highest priority child. c) No more swap operations as the heap property is restored. In worst case there will be $\log(n)$ swap operations.

### 3.3.3 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.

### 3.4 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 3.6(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 3.6(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.

### 3.4.1 Enqueue

Figure 3.7 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 3.3.1. 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 3.3. This first stage is performed in constant time.
Figure 3.7: 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.

as explained in Section 3.3.1. 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.

### 3.4.2 Dequeue

Figure 3.8 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 3.3.2. 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.
Algorithm 1 Pseudocode of Hybrid Priority Queue’s Enqueue Operation

1: procedure HYBRID_PQ.Enqueue(queue, elem)
2:     if Queue = Full then
3:         throw exception
4:     end if
5:     hardware_pq_enqueue(elem)
6:     queue.size +=
7:     if queue.size > queue.hw_limit then
8:         index = queue.size
9:         Copy overflowed hardware element to the end of software queue.
10:        queue.data[index] = overflow_cell
11:        while index ≥ queue.hw_limit do
12:            if queue.data[index] < queue.data[parent(index)] then
13:                swap_queue_data(queue, index, parent(index))
14:                index = parent(index)
15:            end if
16:        end while
17:     end if
18: end procedure

The software implementation of dequeue operation is outlined in Algorithm 2.

Comparing our approach with the related work reported in Section 2.1, 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.

3.5 Evaluation Methodology

Platform The hybrid priority queue was deployed and evaluated on the Reconfigurable 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 3.9.

![Figure 3.8: Steps of dequeue operation in hybrid mode. In this example we assume that the first 3 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.](image)

**Architecture Configuration**  The priority queue was 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 8192 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.
Algorithm 2 Pseudocode of Hybrid Priority Queue’s Dequeue Operation

1: procedure HYBRID_PQ_DEQUEUE(queue)
2:     if Queue = Empty then
3:         throw exception
4:     end if
5:     result = queue.top;
6:     if queue.size < queue.hw_limit then
7:         hardware_pq_dequeue()
8:     else
9:         Replace root with last element of heap array.
10:        queue.data[0] = queue.data[size]
11:        Execute hardware dequeue and return position of newly inserted element.
12:       new_index = hardware_pq_dequeue()
13:       Continue heap restoration in software from the position returned.
14:      restore_sw_heap(new_index)
15:     end if
16:     queue.size --;
17: end procedure

Workload and Metrics  The performance of the priority queue was evaluated using the Classic Hold model [79] [35], where a priority queue of a given size is initialized 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.

The distributions used in our evaluation are listed in Table 3.1, which is similar to those used in [79] and [70]. The transient behavior of the priority queue is measured using the Up/Down model [71], where the queue is initialized to a given size by series of enqueue operation and then emptied by series of dequeue operation.

Table 3.1: Priority increment distributions used in our evaluation.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Expression to generate random values(^1)</th>
<th>Bias</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>-ln(rand)</td>
<td>0.05</td>
</tr>
<tr>
<td>Uniform 0.0 - 2.0</td>
<td>2 * rand</td>
<td>0.66</td>
</tr>
<tr>
<td>Bimodal</td>
<td>0.95238 * rand + if rand &lt;0.1 then 9.5238 else 0</td>
<td>0.13</td>
</tr>
<tr>
<td>Triangular</td>
<td>1.5 * rand</td>
<td>0.80</td>
</tr>
</tbody>
</table>

\(^1\) rand returns a random value uniformly distributed between 0 and 1.
3.6 Results and Analysis

This section presents the results of our hybrid priority queue versus software priority queue evaluation experiments.

![Figure 3.10: Performance comparison between the software and hybrid implementation of a priority queue. Evaluated using the Classic Hold Model, for 4 different priority increment distributions.](image)

**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 3.10 and 3.11. 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 6 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 8192 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.

**Resource Utilization and Scalability**  We implemented our hardware priority queue design on an Altera Cyclone III (EP3C25) FPGA. The resource utilization of the priority queue for different queue lengths is shown in Table 3.2. Each priority queue element is 64
Table 3.2: FPGA resource utilization of the proposed priority queue design for different queue sizes.

<table>
<thead>
<tr>
<th>Size</th>
<th>Look-up tables(LUTs)</th>
<th>Flip-flops</th>
<th>Memory(bits)</th>
<th>Block RAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1,411(5.73%)</td>
<td>906(3.68%)</td>
<td>1,920(0.32%)</td>
<td>8(12.12%)</td>
</tr>
<tr>
<td>63</td>
<td>1,996(8.1%)</td>
<td>1,048(4.25%)</td>
<td>3,968(0.65%)</td>
<td>10(15.15%)</td>
</tr>
<tr>
<td>127</td>
<td>2,561(10.4%)</td>
<td>1,182(4.8%)</td>
<td>8,064(1.32%)</td>
<td>12(18.18%)</td>
</tr>
<tr>
<td>255</td>
<td>3,161(12.84%)</td>
<td>1,330(5.4%)</td>
<td>16,256(2.67%)</td>
<td>14(21.21%)</td>
</tr>
</tbody>
</table>

1 Altera Cyclone III FPGA contains: 24,624 LUTs, 24,624 Flip-flops and 66 Block RAMs.

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 block RAM with a 64 bit wide port, a minimum of 2 M9K blocks are used per level. The block RAM usage can be optimized 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 [57]. The resource utilization of both architectures are shown in Table 3.3. These architectures use distributed memory instead of block RAMs to store queue elements. Figure 3.12 shows that our queue architecture scales well for large
Table 3.3: FPGA resource utilization of shift register and systolic array based priority queue architectures [57] in comparison with proposed priority queue design.

<table>
<thead>
<tr>
<th>Size</th>
<th>Shift Register</th>
<th>Systolic Array</th>
<th>Proposed Design</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUTs</td>
<td>Flip-flops</td>
<td>LUTs</td>
</tr>
<tr>
<td>31</td>
<td>4,995 (20.29%)</td>
<td>2077 (8.43%)</td>
<td>8560 (34.76%)</td>
</tr>
<tr>
<td>63</td>
<td>10,275 (41.73%)</td>
<td>4221 (17.14%)</td>
<td>17520 (71.15%)</td>
</tr>
<tr>
<td>127</td>
<td>20835 (84.61%)</td>
<td>8509 (34.56%)</td>
<td>—</td>
</tr>
<tr>
<td>255</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

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

Figure 3.12: Comparing FPGA look-up table utilization of the proposed priority queue design against shift register and systolic array based priority queue architectures [57] for different queue sizes. Flip-flop utilization also shows a similar trend.

queues, as compared to shift-register and systolic array based architectures [57] in which the combinational logic required increases linearly with queue size.
CHAPTER 4. HARDWARE SCHEDULER

4.1 Overview

As an application of the hybrid priority queue design described in Chapter 3, I 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 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.

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.

![Figure 4.1: A high level architecture diagram of the hardware scheduler along with the custom instruction interface.](image)
4.2 Architecture

A high level block diagram of the hardware scheduler is shown in Figure 4.1.

**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).

**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.

**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.

**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.

4.3 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 Chapter 3 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.

4.4 Evaluation Methodology

**Platform** The scheduler was deployed and evaluated on the Reconfigurable Autonomous Vehicle Infrastructure (RAVI) board, an in-house developed FPGA prototyping platform which was detailed in 3.5.

**Architecture Configuration** The scheduler was 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 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 2048, which is sufficient to support a vast majority of embedded systems. The scheduler can be configured to use Earliest Deadline First (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 (0.1ms, 1ms, 10ms), 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 Earliest Deadline First (EDF) [50] 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.
Workload and Metrics  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 utilization 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.

4.5 Results and Analysis

This section details the results of our hybrid and hardware scheduler evaluation experiments. For our analysis we have considered the following three configurations of a EDF scheduler.

- Software Scheduler: Used as the baseline for evaluating our hybrid and hardware scheduler. Evaluated for up to 2048 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 2048 tasks.

Scheduler Overhead  The overhead of the scheduler was measured for different sets of tasks and timer tick resolutions. Figure 4.2(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 [64], [4], since finer granularity time ticks
(a) Software Scheduler

![Diagram](image1)

(b) Hardware Scheduler

![Diagram](image2)

Figure 4.2: Performance of the software scheduler compared with hardware scheduler for task sizes less than or equal to 255.

(a) Software Scheduler

![Diagram](image3)

(b) Hybrid Scheduler.

![Diagram](image4)

Figure 4.3: Performance of software scheduler compared with hybrid scheduler for task sizes greater than 255.

lead to closer to ideal implementation of such schedulers.

Figure 4.2(b) 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 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.

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 4.3.

![Figure 4.4: Variation in execution times of software and hardware scheduler.](image)

**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 4.4. The difference between the best case and worst case execution time of the software scheduler is 50 times larger than the hardware implementation as shown in Figure 4.4. 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 behavior to ensure task deadlines are met, increases in execution time variation reduces CPU task utilization (i.e. CPU becomes underutilized). On the other hand, the execution times of the hardware scheduler show more deterministic behavior 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 utilization. Figure 4.5 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 4.5.
PART II

CACHE ARCHITECTURE FOR MIXED CRITICALITY SYSTEMS
CHAPTER 5. INTRODUCTION TO MIXED CRITICALITY SYSTEMS

5.1 Background

Safety critical systems are an important part of our daily lives. From braking control in our cars to aircraft controllers, we depend on these critical systems every day, where failure could result in loss of life or a significant damage to the environment. In safety critical systems, it is important to provide guarantees on the worst case response time when a critical event happens (e.g. Airbag deployment, emergency braking). Hence, systems which involve applications of different criticality are subject to strict certification requirements which follows guidelines such as DO-178B for avionic systems and the ISO 26262 functional safety standard for road vehicles. The term "criticality" often refers to the classification of tasks or functionality based on the consequence of failure or the level of assurance required against failure. For example, in DO-178B, applications are classified according to five different safety levels - Catastrophic, Hazardous, Major, Minor and No Effect.

Systems that involve applications of different criticality (e.g. avionics, automotive control) have been traditionally designed by separating safety-critical and non-critical functionality. In the past, this separation has been realized by providing safety-critical applications their own hardware-software platform [9]. This physical separation prevents a non-critical function from adversely affecting the behavior of critical applications, and simplifies the certification process. For example, in an autonomous Unmanned Aerial Vehicle (UAV), flight control, engine and actuation control are safety critical functions, while the imaging sub-system, displays and communication can be grouped under non-critical functions.

In recent years, there has been increasing demand to reduce cost, size and power requirements of embedded and real-time systems in areas such as avionics and automotive control.
One example where this trend is evident is the area of UAVs. The size of UAVs have rapidly decreased with the advent of micro drones and nano air vehicles [21], which are being used in reconnaissance and surveillance. UAVs are no longer limited to military use, as drone cameras which are being used for aerial photography [3] are becoming increasingly common in the consumer market. Reduced size and power requirements often translates to increased flight time of UAVs. To meet the stringent size and power requirements of these systems, functionality of different criticality are implemented on a shared hardware platform, which are called mixed criticality systems [9].

5.2 Motivation

In mixed criticality systems, it is necessary to provide temporal and spacial isolation guarantees for critical tasks to ensure their timing constraints are met under all conditions. Traditional real-time scheduling techniques do not take into account the criticality of a task and assumes all tasks are equally important. The worst case execution time (WCET) estimate of each task is used to determine the schedulability of the task set, according to a scheduling discipline (e.g. RMS, EDF). The real-time scheduling algorithms can work in mixed criticality systems if a highly assured WCET can be obtained for each task in the task set. However, estimation of worst case execution time is a complicated process. Depending on the level of assurance required, the methods and amount of effort involved in WCET estimation varies. For example, the following methods can be used to obtain WCET estimates for the following task criticalities:

- Non-critical tasks: WCET observed during the tests (high water mark).

- Moderately-critical tasks: WCET measured during extensive experimentation constructed for WCET analysis.

- Safety-critical tasks: WCET obtained through code flow analysis and cycle counting under pessimistic assumptions.

To provide guarantees in terms of a task set’s scheduling feasibility in mixed criticality systems, conservative estimates of worst case resource demands should be determined for every task.
This not only increases the effort involved in WCET estimation, but also leads to poor resource utilization under normal operating conditions as the worst case behavior occurs rarely [85].

Having a highly assured WCET estimate for low assurance software is not always possible. The software which implements non-critical functionality are not subjected to the same constraints as critical software. For example, non-critical software may contain code with unknown loop bounds, recursion or runtime memory allocation which are often avoided in critical software. It is also not possible to have tight WCET bounds when we have environment dependent execution time (e.g. execution time dependent on number of obstacles to avoid).

Under overload conditions, no guarantees are provided in traditional real-time scheduling algorithms to ensure higher criticality tasks get preference over lower criticality tasks. Enforcing criticality as priority may avoid criticality inversion, but may not yield optimal priority assignment to maximize processor utilization.

Various mixed criticality models and scheduling algorithms [82, 22, 10, 45, 24] have been proposed to address this issue with respect to CPU scheduling. There have also been efforts to investigate new CPU architectures for mixed criticality systems, which can provide hardware-based isolation to critical tasks without under-utilizing hardware resources. FlexPRET, a processor architecture for mixed criticality systems was proposed in [89] which adds new timing instructions to an existing (RISC-V) ISA and supports flexible scheduling by allowing arbitrary interleaving of threads instructions subject to avoiding hazards. However, in modern computing platforms, often the most unpredictable aspect of a platform is not the CPU, but other components such as storage hierarchy, which can be a performance limiting factor of many applications [85]. Inter-task cache conflicts in mixed criticality systems is one such source of unpredictability that can affect the performance and response time of critical tasks and lead to WCET pessimism. In the second part of this dissertation, I aim to mitigate inter-task interference arising from critical tasks sharing cache with non-critical tasks.

Cache memory greatly improves the overall performance of processors by bridging the increasing gap between processor and memory speed. But, the unpredictable behavior of cache complicates WCET analysis [85]. This complexity is often reduced by making conservative WCET estimates, which is at the cost of processor utilization. In mixed criticality systems,
critical task’s timing behavior is impacted by the inter-task cache interference of non-critical tasks. Thus, reducing the performance of the critical task, and adding complexity to its WCET analysis. Various techniques such as cache locking and partitioning have been proposed to make cache more predictable in real-time systems. Cache locking allows certain lines of the cache to be locked in place, which enables accurate calculation of memory access times. In cache partitioning, a portion of the cache is allocated to a specific task, which eliminates inter-task conflicts. Improved predictability is often achieved at the cost of reduced overall application performance.

In the next chapter, I 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. I assume data is either critical or non-critical. An extension of the least recently used (LRU) cache replacement policy, called Least Critical (LC), is proposed and implemented as part of my proposed cache design. In the LC cache replacement policy, critical data is given preference in the cache. Data can be tagged as critical in two ways: 1) based on task ID, where all data allocated to a task is tagged as critical, and 2) based on memory region, where data from certain address spaces are given preference in the cache. These critical address spaces are defined by critical address range (CAR) registers, which are configurable during run-time. My design provides flexibility and 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.

5.3 Related Work

For the context of cache within multi-tasking real-time systems, various cache locking and partitioning schemes [39, 80, 76, 15, 18, 53, 51, 65, 67, 81, 75] have been proposed to improve the predictability and overall performance of real-time tasks.

In cache partitioning, a task (or set of tasks) is restricted to exclusively use an assigned region of the cache, thus removing inter-task cache conflicts. Software-based partitioning approaches such as [67, 37, 29, 61, 15, 86, 38] use static cache analysis and compiler support to restrict task access to certain regions of the cache. These software-based partitioning techniques require changing from address-based to cache-line-based data mapping to eliminate inter-task
cache conflicts, which makes it difficult for system-wide application. Hardware based cache partitioning techniques, which require additional platform support, have been proposed by [49, 39, 76]. SMART, a hardware based cache partitioning scheme was proposed by [39] where the cache is divided into small fixed sized partitions to be used by performance critical real-time tasks and a large partition shared by non-critical service tasks. A prioritized cache model targeting set associative caches in real-time systems was proposed in [76]. In the prioritized cache model, the cache is partitioned column-wise and each partition can be assigned to a task or marked as “shared”. In addition, a higher priority task can use all partitions owned by lower priority tasks. The use of these hardware based techniques is limited by fixed partition sizes and coarse grained configurability, which may reduce cache utilization.

Cache locking allows an application to load certain data into cache and prevents it from being evicted. Several static and dynamic cache locking schemes [18, 81, 65, 7, 75, 87] have been proposed to improve timing predictability of tasks. While cache locking provides fine grained control over task data, it leads to poor utilization when data does not fit in the cache [81]. Dynamic cache locking also increases overhead and can affect overall task performance if cache lines are locked unnecessarily.

Explicit reservation of cache memory to reduce worst case cache-related preemption delay (CRPD) was proposed in [84], where the state of cache is saved to the stack during preemption and restored when the task continues execution. This technique induces a constant CRPD regardless of the task being preempted, but increases CRPD when no or few cache blocks are shared between pre-empting and pre-empted tasks.

More recently, cache management techniques for mixed critical real-time systems have been proposed to improve the timing predictability and performance of critical tasks. PRETI, a partitioned real time cache scheme was presented in [48], where a critical task is assigned a private cache space to reduce inter-task conflict. The cache lines not claimed by a critical task are marked as shared, and can be used by all tasks. By design, PRETI ensures the $N$ most recently used blocks are present in the cache, where $N$ is the set associativity of the partition reserved for a task. [53] proposed a cache management framework for multi-core architectures. Frequently accessed memory pages called hot pages are determined by profiling
the application and a combination of page coloring and dynamic cache locking mechanisms are used to provide a deterministic cache hit rate for a set of hot pages used by a task. The $MC^2$ (mixed-criticality on multicore) scheduling framework [54] was used in [36] to explore cache scheduling and locking techniques for managing shared caches in multicore systems. For cache scheduling, cache was viewed as a preemptive resource that is schedulable and for cache locking, cache was viewed as a non-preemptive resource that can be accessed via a locking protocol.

In addition to the real-time cache management schemes described, there has been numerous advancements in static WCET analysis techniques for architectures using general purpose caches. These techniques provide a tighter WCET bound on platforms with generic cache (e.g. LRU cache) by predicting the worst case cache behavior of memory references [25, 77, 47, 32, 73] and taking into account cache-related preemption delay [5, 6, 20] in multi-tasking preemptive real-time systems.

Existing hardware approaches partition cache at a column granularity, which limits the number of cache partitions. In our proposed cache design, we allow fine grained control over task data by providing a mechanism to tag critical data based on address space ranges that can be configured at run-time. This enables applications to better utilize cache. This article is an extension of our previous work [43], which only supported address based tagging of critical data. In this work, we additionally provide the flexibility to tag critical data based on task ID, which is useful for tasks that have a small memory footprint. The cache lines which are non-critical are shared by all tasks. By tagging critical data based on task ID or address range, which are given preference in the cache, the overhead involved in locking/unlocking individual cache lines is eliminated.

The dynamic cache locking approaches described consider each task in isolation and look to improve or accurately estimate the WCET of a single task. In preemptive multi-tasking real-time systems, additional overhead is introduced due to cache related preemption delay, which needs to be accounted for. In our cache design, we reduce the inter task conflicts arising from non-critical tasks sharing cache with critical tasks.
CHAPTER 6. CRITICALITY AWARE CACHE DESIGN

We present a criticality aware cache design for mixed criticality real-time systems to reduce inter-task cache conflicts and decrease the response time of critical tasks. Our cache architecture assumes data is either critical or non-critical. The core of the design is a new cache replacement policy, called Least Critical (LC), in which data tagged as critical is given preference in the cache. We provide two mechanisms to tag critical data: 1) based on Task ID, where all memory addresses accessed by a specified task are considered critical, and 2) based on address space, where data from specified address ranges are considered critical. Task ID based tagging is suitable for tasks with small memory footprints, while address based tagging allows fine grained control over critical data by giving preference to data from certain address ranges. Our flexible cache architecture enables switching the cache replacement policy between LRU and LC at run-time. The LC cache replacement policy and its hardware implementation is described in detail next.

6.1 Least Critical Cache

Our Least Critical cache (LC cache) replacement policy targets set associative 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 the number of lines that have critical data. We also maintain the LRU order for critical and non-critical lines in each cache set.

During a cache hit, the LRU order of either the 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 criteria in the following order: 1) Empty cache line,
2) Least recently used non-critical cache line, and 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, the number of critical cache lines in that set is updated accordingly. A critical cache line is 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 are no critical data lines in a cache set.

**Cache bypass for non-critical data.** The cache can be configured to enable cache bypass for non-critical data. This avoids the eviction of a critical cache line by non-critical data when all lines in a cache set are critical. If disabled, the least recently used critical line can be evicted by non-critical line when a cache set has only critical lines. This ensures at-least one cache-way is available for non-critical tasks in the worst case. This cache configuration can be changed at run-time. In Section 6.4, the effect of this configuration on the trade-off between improving predictability and overall task-set performance will be examined.

A working example of the LC cache policy in a 4-way set associative cache is shown in Figure 6.1. In its initial state, a set contains three non-critical lines (A, B, C) with line A being least recently used and one critical line (D^C). The LRU order of the non-critical lines is changed after a cache hit on line 'A'. A cache miss on line 'E^C', results in the eviction of non-critical line 'B', which was the least recently used. The number of critical lines is increased to two and line 'E^C' is made most recently used (MRU). A cache hit on line 'D^C' changes the LRU order of the critical lines. Finally, a cache miss of line 'F' results in the eviction of the LRU non-critical line 'C' and the LRU order of the non-critical lines is updated accordingly.

### 6.1.1 Hardware Implementation

Figure 6.2 depicts a high level block diagram of the LC cache architecture. The primary components of the cache controller are described in detail next:

**Task Criticality Register.** The task criticality register enables task ID based tagging of critical data. When this register is set, all data accessed by the active critical task are labeled as
critical. The operating system should update this register during a context switch based on the criticality of the task that is activating.

**Critical Address Range (CAR) Comparator.** CAR registers are used to identify critical data based on a memory address range. 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.

**Access History Module.** 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 Comparator.** Generates cache hit/miss signals by comparing requested memory addresses with tag bits associated with each cache line.

**Non-Critical (NC) Data Bypass Register.** Enable/disable cache bypass of non-critical data, when all lines in a cache set are critical.
Soft Reset Register. 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 and used when changing the cache replacement policy at run-time.

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

The implementation of our architecture additionally allows switching between our LC cache policy and a conventional LRU policy at run-time.

Switching from LC to LRU. The replacement policy can be changed from LC to LRU by 1) Clearing the task criticality and CAR registers and 2) Triggering a soft-reset, which resets the critical data line count of each cache set to zero. After a soft-reset of the LC cache, the cache lines that where critical are made the most recently used non-critical lines.

Switching from LRU to LC. This requires the modification of task criticality and CAR registers. However, modifying CAR registers at run-time can cause the critical-cache-line count to become incoherent. This occurs when the data from a new CAR is already present in the cache as

Figure 6.2: High level block diagram of the Least Critical (LC) Cache Controller. Dotted blocks are registers which can be configured through software instructions.
non-critical or when the new CAR partially overlaps an existing CAR. A cache flush must be performed before changing the replacement policy from LRU back to LC, which ensures consistency of critical-cache-line count.

6.1.2 Application-level usage model of LC cache

To make use of the LC cache, task ID based tagging provides a convenient way to tag critical data. This allows all memory allocated to a task to be critical. This approach is useful if the critical task has a small memory footprint, which is often the case for real-time tasks. If the critical task has a large memory footprint, this approach would result in deprivation of the cache for non-critical tasks. To allow fine grained control over tagging critical data, we provide a mechanism to tag critical data based on address space. The developer can manually tag critical data variables and the compiler places those variables in a separate section of memory. In GCC, this can be accomplished using the "section" attribute, which specifies that a variable resides in a particular section. For example,

```
int cdata __attribute__((section("critical")));
```

places the variable cdata into a memory region called “critical”.

The problem of selecting critical data is similar to selecting/allocating optimal data for cache locking and scratch pad memory. A number of approaches have been proposed to identify the optimal set of variables as locked contents and map the selected data to cache memory [53, 81, 87, 83]. Such methods could be leveraged to help automate the process of selecting what data to make critical. Instead of allocating the data set to cache or scratch pad memory, the selected data could be tagged as critical by storing it in a separate address range. When compared to cache locking, our technique avoids the run-time overhead of locking mechanisms, while still allowing critical data to stay in cache. When using a single critical address range (CAR) to tag critical data, as long as critical data size does not exceed the cache size, our LC cache policy behaves similar to cache locking, as critical data is not evicted by non-critical memory requests. In addition to that, the LC cache also reduces inter-task cache interference. When a critical task is preempted by a higher priority non-critical task, the critical cache lines will not be evicted by the preempting task, which reduces cache related preemption delay (CRPD).
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.

### 6.2 Impact on WCET Analysis

In a single processor systems with cache memory, there are two main types of cache conflicts that introduce unpredictable behavior and complicate the worst case execution time analysis: 1) Intra-task cache conflicts, which occur when a task evicts its own data from the cache, and 2) Inter-task cache conflicts, which occur in preemptive multi-tasking systems, when a preemption task evicts the cache lines used by the preempted task. This is also known as cache related preemption delay (CRPD). In this section, we will describe how the state of art in worst case execution time analysis applies to our LC cache replacement policy. We limit our discussion to data caches only.

#### 6.2.1 LC Cache Semantics

A set associative LC cache can be modeled by assigning ages to each cache line in a set and keeping track of the number of critical cache lines. For an $A$-way set associative cache, let the set of ages be $0, \ldots, A - 1$. The age function, $a(m)$, returns the age of a memory block if it is present in the cache set, else returns $-1$. The age of the least recently used block will be $A - 1$ and the age of most recently used block will be $0$. The ages of critical memory blocks will always be less than the non-critical memory blocks. The number of critical memory blocks in a cache set is given by $ch_{count}$. The function $C(m)$ defined in Equation 6.1, is used to determine if a memory block, $m$, is critical or not. When a memory block $m_a$ is accessed, the ages of the memory blocks in its cache set are updated according to the function $a'(m)$ defined in Equation 6.2.

$$
C(m) = 1, \quad \text{if } m \text{ is critical}
$$

$$
= 0, \quad \text{if } m \text{ is not critical}
$$

(6.1)
\[ a'(m) = 0 + cb_{\text{count}}, \quad \text{if } m_a = m \text{ and } C(m_a) \neq 1 \]
\[ = 0, \quad \text{if } m_a = m \text{ and } C(m_a) = 1 \]
\[ = a(m) + 1, \quad \text{if } m_a \neq m \text{ and } a(m_a) > a(m) \text{ and } C(m_a) > C(m) \quad (6.2) \]

In case of a cache miss, the memory block with age \( A - 1 \) is replaced with the newly accessed block. The ages and \( cb_{\text{count}} \) of that cache set is updated accordingly.

### 6.2.2 Cache analysis of a single task

Several techniques [25, 77, 47, 32, 73, 80] have been proposed to estimate the WCET of a task on architectures with cache memory. A well known method for WCET estimation on architectures with caches is based on the theory of abstract interpretation [77, 32]. In this static program analysis method, the control flow graph of a program is traversed and abstract cache states are used to safely predict the possible cache contents at every execution point of the program. Based on this approach, three analysis techniques have been developed [77] to tell if a memory block is always present in the cache (must analysis), if a memory block may be in the cache (may analysis) or if a memory block will not be evicted once it is loaded to cache (persistence analysis). Each memory reference is then categorized as always hit, always miss, first miss or not classified, which represents its worst case behavior. This information is then used to compute the impact of memory references on WCET estimation.

The replacement policy used has a strong influence on the precision of WCET analysis. A detailed study [68] that evaluated the timing predictability of four cache replacement policies (LRU, MRU, PLRU, FIFO) showed that LRU cache allows for more precise WCET estimation when compared to other cache replacement policies. Our LC cache replacement policy is an extension of a conventional LRU cache and the WCET analysis is similar to that of an LRU cache with minor modifications. The LC cache architecture has an option to enable or disable non-critical data bypass, which alters the behavior of the cache replacement policy. In the following section, we explain how the abstract interpretation based cache analysis proposed in [77] can be applied to LC cache for different configurations.
To begin with we will assume non-critical data bypass is enabled, which means critical data cannot be evicted by non-critical memory requests.

**Task ID based tagging of critical data.** When tagging critical data based on task ID, all data used by the task is considered as critical and all cache lines are available for the task. In this case, the cache analysis used to estimate the WCET for LRU cache also applies to LC cache. Since our LC cache replacement policy acts as a conventional LRU cache when all lines in a cache set are critical, the LRU must, may and persistence analysis described in [77] applies to LC cache as well.

**Address range based tagging of critical data.** When tagging critical data based on address range, a task can access both critical and non-critical data. The cache might also contain critical data belonging to other tasks along with its own critical data. Since non-critical data cannot evict critical cache lines, which might belong to other tasks, no guarantees can be provided about the availability of cache lines to non-critical data (without knowing the total critical data used by all other tasks and the critical address ranges). Hence, the abstract interpretation methods (must, may and persistence analysis) should only consider critical memory references made by the task. If the task’s critical data is less than the cache size, then the critical data will not be evicted from the cache once loaded, assuming there is no intra-task cache conflicts.

### 6.2.3 Analysis of inter-task cache conflicts

In multi-tasking systems, inter task cache conflicts occur when a high priority task preempts a lower priority task. The preemting task can evict the cache lines used by the preempted task, which causes cache misses in addition to those caused by intra-cache misses when the preempted task was the only task in the system. The upper bound on these additional cache misses due to preemption is called cache related preemption delay (CRPD). Statically bounding CRPD involves analyzing both preempted and preemting tasks. The concept of useful cache blocks (UCBs) was introduced in [46, 5] to analyze the effect of preemption on a preempted task. A memory block which is cached before preemption and may be used later is called an UCB. At a given program point, the number of UCBs can be predicted through data flow
analysis, which gives the upper bound on additional cache misses due to preemption. The memory blocks accessed by a preemting task during execution are referred to as evicting cache blocks (ECBs). A preemting task can increase CRPD only when its ECB can evict a UCB of the preempted task. Several methods have been proposed [5, 6, 20], which combines the notion of ECB and UCB for CRPD analysis when using a set associative LRU cache.

Priority based scheduling algorithms assume all tasks are equally important. In mixed criticality real-time systems, this means a critical task might be assigned a lower priority than a non-critical task. Our LC cache is designed to reduce the inter-task cache conflicts of critical tasks caused by non-critical task preemption. A critical task, when preempted by a higher priority non-critical task, will not cause any CRPD. This is due to the LC cache replacement policy not allowing critical data to be evicted by non-critical memory requests. When using the LC cache replacement policy, only higher priority critical tasks can cause inter-task cache conflicts. The CRPD is bounded by the analysis described in [20].

**Non-critical data bypass disabled.** When this configuration option is disabled, one way in a $A$ - way set associative cache will be shared between critical and non-critical data. Hence, only $A - 1$ ways will be available exclusively for critical data. This affects single task cache analysis of critical tasks when using address based tagging of critical data. The abstract interpretation based methods should be applied by treating the cache as an $A - 1$ way set associative cache, instead of $A$ - way cache. The size of the cache used in the analysis will be reduced to $A - 1/A \times \text{total cache size}$. This would also mean that one cache way is available for non-critical tasks, which acts as a direct mapped cache of size $1/A \times \text{total cache size}$.

### 6.3 Evaluation Methodology

#### 6.3.1 Hardware Platform and Configuration

Our LC cache was evaluated on a XUPV5-LX110T, a Xilinx Field Programmable Gate Array (FPGA) development 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 [2], was used to implement our cache design. Leon3 features a 7-stage instruction
pipeline and separate instruction and data caches. We limit analysis to data cache only. Our cache design was implemented as an 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 similar to the one presented in [31] was designed to accurately measure the performance of the data cache. A high-level block diagram of the hardware cache profiler is shown in Figure 6.3. The profiler is configurable to measure data cache hits/misses and execution time statistics for each task, along with overall application statistics. The profiler sends the data collected to a server through a UART interface for offline analysis.

![Figure 6.3: High level block diagram of the non-intrusive hardware cache profiler.](image-url)
6.3.2 Workload and Metrics

To evaluate the performance of our LC cache design, we used a set of five real-time benchmark programs. The critical task was an inverted pendulum controller (IPC). We varied the computation 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 worst case execution time (WCET) project [28] and consisted of CRC (cyclic redundancy check), FDCT (discrete cosine), FIR (finite impulse response filter), and data compress. The characteristics of these programs are shown in Table 6.1. FreeRTOS [1], 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. We also measured the maximum observed cache miss rate, along with the maximum observed execution time (MOET) of the critical task.

Table 6.1: Characteristics of benchmark programs used to evaluate our LC cache design.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Code Size (bytes)</th>
<th>Data Size (bytes)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller (IPC)</td>
<td>1092</td>
<td>256 - 8192</td>
<td>0.11 - 4.87</td>
</tr>
<tr>
<td>CRC</td>
<td>1216</td>
<td>1048</td>
<td>0.16</td>
</tr>
<tr>
<td>FDCT</td>
<td>2940</td>
<td>132</td>
<td>0.49</td>
</tr>
<tr>
<td>FIR</td>
<td>572</td>
<td>2948</td>
<td>54.06</td>
</tr>
<tr>
<td>Compress</td>
<td>3316</td>
<td>2416</td>
<td>18.52</td>
</tr>
</tbody>
</table>

1 Execution time for task running alone.

6.4 Results and Analysis

To verify the effectiveness of our LC cache, we conducted two sets of experiments, which are described in detail next. For both experiments non-critical data bypass mode is disabled to allow non-critical tasks access to at least one-way of the 4-way set associative cache.
6.4.1 Experiment 1 - Two-task Setup

In the first experiment, the critical task and one of the non-critical tasks listed in Table 6.1 are executed in a round-robin fashion for different critical data sizes. The amount of critical data processed by the inverted pendulum controller is varied by changing the size of primary matrix it processes to compute control values. When the size of the primary matrix is increased, the total memory requests by the critical task also increases, which affects the cache hit/miss rate of the task. This experiment is conducted for each non-critical task in Table 6.1. The data used by the critical task is placed in a separate section of memory and tagged as critical using the critical address range (CAR) registers.

![Graph 1](image1)

(a) LRU Cache

![Graph 2](image2)

(b) LC Cache

Figure 6.4: Critical Task: Cache performance of LC cache when compared to LRU cache. Critical task: Inverted Pendulum Controller (IPC) being run pair-wise with CRC, FDCT, Compress, and FIR.

![Graph 3](image3)

(a) LRU Cache

![Graph 4](image4)

(b) LC Cache

Figure 6.5: Overall Application: Cache performance of LC cache when compared to LRU cache. Critical task: Inverted Pendulum Controller (IPC) being run pair-wise with CRC, FDCT, Compress, and FIR.
Figures 6.4(a) and 6.4(b) can be used to examine the difference in cache miss rate of the critical task when using an LRU (Figure 6.4(a)) versus an LC (Figure 6.4(b)) cache replacement policy. When using the LC cache replacement policy, the critical task's cache miss rate is reduced by around 50 - 80% for critical data sizes less than the cache size (4KB). This decrease in cache miss rate is due to the LC policy reducing inter-task cache interference by giving critical data preference in the cache over non-critical data. When the critical task’s critical data is larger than the cache, increased intra-task interference negates the benefit of using the LC cache. In other words, the misses associated with the critical task are primarily due to its critical data evicting other members of its critical data set. At this point the critical task’s cache performance when using an LC cache is similar to when using an LRU cache.

Figures 6.5(a) and 6.5(b) shows the overall application’s cache performance when using an LRU (Figure 6.5(a)) in comparison with our LC (Figure 6.5(b)) cache replacement policy. Three key observations are: 1) while the amount of critical data is less than the cache size there is marginal difference in overall application cache performance between the two cache replacement policies, 2) when the amount of critical data is close to the cache size, overall application cache performance degrades when using the LC cache replacement policy, however the critical task still benefits (Figure 6.4(b)), and 3) when the amount of critical data exceeds the cache size, the overall application cache performance is less than when using a LRU replacement policy and the critical task no longer benefits from using the LC cache replacement policy. The primary reason for the overall application cache performance decreasing as the critical task’s memory footprint approaches and exceeds the cache size is that only one cache-way remains available for non-critical tasks. As will be seen in our second experimental setup, similar trends are observed and their implications discussed.

### 6.4.2 Experiment 2 - Five-task Setup

In the second experiment, the benchmark tasks listed in Table 6.1 were executed together 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). For the LC cache, we measured the cache performance using both address range based
and task ID based tagging of critical data. For address range based tagging, the data used by the task is placed in a separate section of memory and tagged as critical using CAR registers. When tagging critical data using task ID, all accesses to memory by the critical task, including its stack, are considered critical. Figure 6.6 shows the cache miss rates for the critical task as the size of its critical data increases. With the LC policy, the critical task’s references are favored and we see 60% - 80% reduction in the critical task’s cache miss rate when compared to the LRU policy. When using task level tagging of critical data, the cache miss rate for the critical task is further reduced as access to the task’s stack is also given preference in cache. When the size of the critical tasks’ data reaches the cache size (4KB), the performance benefit of using the LC cache replacement policy is reduced to 30% - 60%. This is due to increased intra-task cache conflicts. At 8KBs of critical data, we exceed the size of the cache, at which point the LRU and LC cache are indistinguishable for the critical task.

The cache miss rate for the overall application (critical and noncritical tasks) is shown in Figure 6.7. Overall performance is not adversely affected by the LC cache favoring the critical task, until we reach the size of the L1 cache at 4KB. Comparing across Figures 6.7(a), 6.7(b) and 6.7(c) at 4KBs of critical data, the cache miss rate of the critical task is reduced at the expense of the noncritical tasks’ cache performance, which reduces overall application cache performance. However, at 8KBs of critical data, favoring the critical task benefits neither it or any other task, thus at this point LRU would be a better choice.

Effect of Critical Tasks’ Period. When using the LRU cache, the miss rate of the critical task increases with its period as shown in Figure 6.6. This is due to inter-task cache interference increasing as the critical task is executed less 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 cache conflicts on the critical task by giving preference to that task’s critical data. Thus, even though the non-critical task may have equal or higher priority in RM scheduling, the critical task is still given preference in the cache.

Effect of Cache Size. The benchmark tasks were also executed using different cache sizes. For these experiments, the periods of both critical and non-critical tasks were kept constant at
Figure 6.6: Critical Task: Performance of LC cache when compared to LRU cache. Critical task run with four non-critical tasks: CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms
Figure 6.7: Overall Application: Performance of the LC cache when compared to the LRU cache. Critical task run with four non-critical tasks: CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms
200ms and critical data was tagged based on address range. Figure 6.8 shows the performance of the critical task for each cache size. For any given set of tasks, the intra-task interference reduces as the cache size is increased. Figure 6.8(a) shows that even when the critical task has a small memory footprint, there is significant performance variation when using LRU cache. While Figure 6.8(b) shows the LC cache policy has a more predictable miss rate and consistently outperforms LRU when the critical task’s data is less than the cache size. The overall application cache performance for each cache size is shown in Figure 6.9. When compared to the LRU cache (Figure 6.9(a)), the cache miss rate of the LC cache (Figure 6.9(b)) rises steeply when the critical tasks’ data exceeds the cache size. This is because only one way of the cache set will be available for non-critical tasks in our LC cache policy when critical data occupies all
cache lines. The non-critical tasks are deprived the cache, which decreases the overall cache performance. As the cache size is increased, availability of the cache for non-critical tasks increases which improves overall application performance as shown in Figure 6.9.

Table 6.2: Maximum observed cache miss rate of the critical task when using a LC cache in comparison with a LRU cache. 4KB 4-way set associative cache.

<table>
<thead>
<tr>
<th>Critical Data in Bytes</th>
<th>Maximum Observed Cache Miss Rate</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>LC</td>
</tr>
<tr>
<td>256</td>
<td>9.52%</td>
<td>2.95%</td>
</tr>
<tr>
<td>512</td>
<td>5.39%</td>
<td>1.17%</td>
</tr>
<tr>
<td>1024</td>
<td>3.91%</td>
<td>0.54%</td>
</tr>
<tr>
<td>2048</td>
<td>3.32%</td>
<td>0.28%</td>
</tr>
<tr>
<td>4096</td>
<td>2.92%</td>
<td>0.99%</td>
</tr>
<tr>
<td>8192</td>
<td>2.74%</td>
<td>2.66%</td>
</tr>
</tbody>
</table>

Figure 6.10: Decrease in maximum observed execution time (MOET) of the critical task when using a LC cache in comparison with a LRU cache. 4KB 4-way set associative cache.

**Observed Worst Case Behavior of Critical Task.** During Experiment 2 (i.e. five task setup), we also measured the worst case cache performance and maximum observed execution time (MOET) of critical task. In our experiment, the worst case behavior is observed when all non-critical tasks are executed before a critical task instance. The worst case behavior is the same for all critical task periods and occurs every 200ms, which is the period of the non-critical tasks. To avoid cold start cache behavior, the measurements were taken after warming up the
Table 6.3: FPGA resource utilization of the proposed cache design in comparison with LRU cache for different cache sizes.

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>LRU LUTs</th>
<th>LRU Flip-flops</th>
<th>LC LUTs</th>
<th>LC Flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>1040 (1.5%)</td>
<td>510 (0.74%)</td>
<td>1459 (2.11%)</td>
<td>702 (1.02%)</td>
</tr>
<tr>
<td>8KB</td>
<td>1246 (1.8%)</td>
<td>832 (1.20%)</td>
<td>2024 (2.93%)</td>
<td>1216 (1.76%)</td>
</tr>
<tr>
<td>16KB</td>
<td>1693 (2.45%)</td>
<td>1474 (2.13%)</td>
<td>2969 (4.30%)</td>
<td>2242 (3.24%)</td>
</tr>
<tr>
<td>32KB</td>
<td>2568 (3.72%)</td>
<td>2756 (3.99%)</td>
<td>4292 (6.21%)</td>
<td>4860 (7.03%)</td>
</tr>
</tbody>
</table>

1 Xilinx Virtex V FPGA contains: 69,120 LUTs and 69,120 Flip-flops.

cache for 10 task set scheduling iterations. The observed worst case cache miss rate for each critical data size when using LC and LRU cache is shown in Table 6.2. When the critical task’s data is less than or equal to the cache size (4KB), we see about a 65-90% decrease in maximum observed cache miss rate using a LC cache in comparison with a LRU cache. This is due to the LC cache reducing inter-task cache conflicts by giving preference to the critical task’s data.

The percentage decrease in MOET when using an LC cache, as compared to an LRU cache, for each critical data size is shown in Figure 6.10. The MOET shows a 4% decrease when compared to using a LRU cache, regardless of critical task period. When the amount of critical data is greater than the cache size, the LC cache behaves as LRU and the critical task does not benefit from the LC cache policy. It should be noted that Leon3 has a write-through cache with no write allocate. We believe the MOET when using LC cache would further improve over a LRU cache if a write back cache was used, which we plan to explore in the future.

### 6.4.3 Hardware Resource Utilization

We implemented our LC cache design as a L1 data cache in the Leon3 soft-core processor, which was deployed on an Xilinx Virtex-5 FPGA platform. The design was implemented as a 4-way set associative cache with a line size of 16 bytes. The hardware resource utilization of the LRU and LC cache controller for different cache sizes is shown in Table 6.3. Our LC cache
requires additional combinational logic to identify critical memory accesses. However, this overhead is constant and does not depend on the size of the cache. For a 4-way set associative cache, both the LRU and LC cache required 5 bits per set to store cache line access history. In addition, our LC cache requires 3 additional bits per cache set to store the critical lines count (0-4). Thus, the access history storage overhead for our LC cache is 60% greater than Leon3’s traditional LRU cache.

6.4.4 Other Considerations

Critical data selection.

The results presented give evidence that an LC cache can reduce inter-task cache conflicts and improve the response time of critical tasks. Task ID based tagging of critical data is useful for tasks that have a small memory footprint. But as critical data size increases, non-critical tasks become cache deprived, which degrades overall application performance. It was also observed that the critical task does not benefit from using a LC cache over a LRU cache when intra-task cache interference is high. This occurs when a critical task’s data is much larger than the cache size. Hence, to efficiently utilize the cache and improve WCET the choice of critical data is very important for tasks with large memory footprint. Existing static program analysis based techniques [53, 81, 87, 83] for optimally selecting data for cache locking and scratch pad memory can be leveraged to identify and tag critical data off-line. Our LC cache design also provides the flexibility to change the critical address range and cache policy during run-time, which could be used as a mechanism to better utilize cache by adapting to changing operating conditions. This is useful in systems designed to execute in different criticality modes [24].

Shared data. In the experiments presented, we assumed no data is shared between tasks. When using task ID based tagging of critical data, the data shared between a critical and non-critical task could result in inconsistent cache behavior. To avoid this, the critical address range (CAR) registers should be used to tag shared critical data.

Cache bypass for non-critical data. When critical data occupies all lines of a cache block, if cache bypass for non-critical data is enabled, then a non-critical task will be completely
deprived of the cache. This can dramatically degrade the performance of non-critical tasks when compared to allowing them access to one-way of the cache set (i.e. disabling cache bypass for non-critical data).
CHAPTER 7. DYNAMIC CACHE MANAGEMENT - A CASE STUDY

7.1 Introduction

Dynamic reconfiguration is desirable in many mixed criticality systems (e.g. avionics) where computation and other resource requirements can change at runtime. This may be due to mode changes in response to changing operating conditions, or in response to threats or faults. I assert that a hardware platform that is aware of an application’s criticality can provide better response times for critical tasks in a dynamic environment. A hardware platform that can adapt to different operating modes to refocus platform resources also enables better resource utilization. Such a platform requires an infrastructure to monitor resource utilization at runtime and configurable hardware components that can adapt to changing conditions. The implementation of our cache architecture, which allows switching between our Least Critical (LC) and a conventional Least Recently Used (LRU) policy at run-time, is a step towards this direction. In this chapter, I investigate the use of non-intrusive lightweight hardware monitors to observe the performance of the cache, and to provide runtime feedback for dynamically changing the cache configuration to improve cache utilization. The primary goals of this study are: 1) identify the metrics that can be used to measure the runtime performance of our LC cache, 2) implement the hardware monitor infrastructure needed to provide runtime feedback to the real-time operating system, and 3) explore heuristics for dynamic cache management.

7.2 Multi Criticality Workload

To investigate the feasibility of dynamic cache management, the task set from the informal partial specification for a hypothetical avionics mission control computer system described in [52] was used. The specification describes the timing constraints and computation requirements
of functionalities found in a typical fighter aircraft, which includes navigation, control, displays, tracking, and weapon control. It is noted that processing is often organized into modes (e.g. navigation, tracking) and a change of mode typically modifies the computational and period requirements of some functions. Each function is also classified based on its importance as either Critical, Essential or Background.

For our case study, we chose a set of eight periodic tasks from the mission control task set described in [52], which are either Critical or Essential. The specification does not provide any details on individual task characteristics (e.g. code size, memory and stack usage) apart from the computation time. Hence, for each mission control task, we selected a benchmark program from the worst case execution time (WCET) project [28] and modified the task properties to match the computation requirements given in [52]. In addition, two programs were chosen to run as background tasks. A summary of the task set properties is given in Table 7.1. To evaluate the behavior of the cache when computation requirements of a system change dynamically, three different modes of operation were defined: 1) Surveillance, 2) Tracking and 3) Engage. The mapping of each task to these modes of operation is given in Table 7.2.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Execution Time (ms)</th>
<th>Period (ms)</th>
<th>Utilization</th>
<th>Importance</th>
<th>Program(^1)</th>
<th>Data Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radar Tracking</td>
<td>2</td>
<td>40</td>
<td>0.05</td>
<td>critical</td>
<td>jfdct</td>
<td>1024</td>
</tr>
<tr>
<td>Target Tracking</td>
<td>4</td>
<td>40</td>
<td>0.1</td>
<td>critical</td>
<td>jfdct</td>
<td>2048</td>
</tr>
<tr>
<td>Aircraft Flight Data</td>
<td>8</td>
<td>50</td>
<td>0.16</td>
<td>critical</td>
<td>ndes</td>
<td>1544</td>
</tr>
<tr>
<td>HUD Display</td>
<td>6</td>
<td>50</td>
<td>0.12</td>
<td>essential</td>
<td>cnt</td>
<td>3632</td>
</tr>
<tr>
<td>MPD Tactical Display</td>
<td>8</td>
<td>50</td>
<td>0.16</td>
<td>essential</td>
<td>cnt</td>
<td>4928</td>
</tr>
<tr>
<td>Steering</td>
<td>6</td>
<td>80</td>
<td>0.075</td>
<td>critical</td>
<td>qurt</td>
<td>64</td>
</tr>
<tr>
<td>Weapon Trajectory</td>
<td>7</td>
<td>100</td>
<td>0.07</td>
<td>critical</td>
<td>qurt</td>
<td>64</td>
</tr>
<tr>
<td>Poll RWR</td>
<td>2</td>
<td>200</td>
<td>0.01</td>
<td>essential</td>
<td>fcntl</td>
<td>144</td>
</tr>
<tr>
<td>BG1</td>
<td>58</td>
<td>400</td>
<td>0.147</td>
<td>background</td>
<td>adpcm</td>
<td>1904</td>
</tr>
<tr>
<td>BG2</td>
<td>5</td>
<td>50</td>
<td>0.102</td>
<td>background</td>
<td>hudcmp</td>
<td>20800</td>
</tr>
</tbody>
</table>

\(^1\) Programs taken from WCET project [28].

7.3 Results and Analysis

The platform setup for the conducted experiments was similar to the setup detailed in Section 6.3.1. Rate monotonic (RM) scheduling was used to schedule the task set on FreeRTOS, which was configured to run a preemptive priority based scheduling algorithm. The background tasks were assigned a priority lower than all critical and essential tasks. The overall cache miss
Table 7.2: Mapping of each task to mode of operation.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Surveillance</th>
<th>Tracking</th>
<th>Engage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radar tracking</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Target tracking</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Aircraft flight data</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HUD display</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MPD tactical display</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Steering</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Weapon trajectory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poll RWR</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BG1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BG2</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

The rate along with the miss rates of individual tasks were used as the metrics to evaluate the cache performance, which was measured using the non-intrusive cache profiler detailed in Section 6.3.1. The LRU cache performance was used as a baseline for comparison. In each mode of operation, the performance of our LC cache was measured for two different configurations:

- **Config – C**: Only data used by critical tasks tagged as critical.
- **Config – CE**: Data used by both critical and essential tasks tagged as critical.

These experiments were conducted using data cache sizes of 4K bytes and 8K bytes. The overall cache miss rate along with the cache miss rates and normalized execution times for each critical and essential task in all three modes of operation is shown in Figures 7.1 to 7.12.

**Surveillance Mode**: In surveillance mode, there are two critical tasks (Aircraft flight data and Steering) and two essential tasks (HUD display and Poll RWR) along with two background tasks (BG1 and BG2). Figures 7.1 and 7.2 shows the performance of the LC cache in comparison with the LRU cache for a 4KB and 8KB cache respectively. When using an LC cache in **Config – C**, the cache miss rate of critical tasks is reduced by 20 - 80% as seen in Figures 7.1(a) and 7.2(a). In **Config – C**, only data from the critical tasks are tagged as critical, which reduces the cache interference on critical tasks from essential and background tasks. When using LC cache in
Figure 7.1: Mode - Surveillance, Cache Size: 4K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.2: Mode - Surveillance, Cache Size: 8K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.
Figure 7.3: Mode - Surveillance, Cache Size: 4K Bytes: Normalized task execution times with LC cache when compared to LRU cache. In *Config − C*, only critical tasks’ data tagged as critical. In *Config − CE*, critical and essential tasks’ data tagged as critical.

Figure 7.4: Mode - Surveillance, Cache Size: 8K Bytes: Normalized task execution times with LC cache when compared to LRU cache. In *Config − C*, only critical tasks’ data tagged as critical. In *Config − CE*, critical and essential tasks’ data tagged as critical.
Config – CE, the amount of critical data is increased, as data from both critical and essential tasks are tagged as critical. This increases the inter-task cache interference between critical and essential tasks. This is evident in Figure 7.1(b). The amount of critical data being much greater than the 4K byte cache causes a significant degradation in the critical data’s cache performance. However, when using a 8K byte cache, we see in Figure 7.2(b) that using the LC cache in Config – CE improves the cache performance of both critical and essential tasks, due to the increased cache size reducing the inter task cache interference.

![Figure 7.5: Mode - Tracking, Cache Size: 4K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.](image)

![Figure 7.6: Mode - Tracking, Cache Size: 8K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.](image)
Figure 7.7: Mode - Tracking, Cache Size: 4K Bytes: Normalized task execution times with LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.8: Mode - Tracking, Cache Size: 8K Bytes: Normalized task execution times with LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.
Tracking Mode: In tracking mode, in addition to the tasks in surveillance mode, a radar tracking critical task is added to the task set. Figures 7.5 and 7.6 compare the performance of the LC cache with the LRU cache in tracking mode. The results are similar to that in surveillance mode. When using the LC cache in Config – C, the cache miss rate of all critical tasks are reduced as seen in Figures 7.5(a) and 7.6(a). However, Figure 7.5(b) shows that using a 4K byte LC cache in Config – CE is not beneficial to any critical task and the overall cache miss rate also increases by around 17%. When using a 8K byte LC cache in Config – CE, the overall cache miss rate reduces and all critical and essential tasks except 'steering task', shows reduction in cache miss rate as shown in Figure 7.6(b).

Engage Mode: In engage mode, there are four critical tasks, two essential tasks, and two background tasks. The performance of a LC cache when compared to a LRU cache in engage mode are shown in Figures 7.9 and 7.10. When using a 4K byte LC cache in engage mode, on average we see a minor advantage for critical tasks in Config – C and no advantage in Config – CE. This is due to increased cache interference within critical tasks. When using a 8K byte LC cache, we do see an advantage over using a LRU cache as the overall cache performance also improves along with critical task performance.

Execution times: From the results presented in Figures 7.3, 7.4, 7.7, 7.8, 7.11 and 7.12, we do not see a significant improvement in execution times of tasks when using LC cache in
Figure 7.10: Mode - Engage, Cache Size: 8K Bytes: Performance of LC cache when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.

Figure 7.11: Mode - Engage, Cache Size: 4K Bytes: Normalized task execution times with when compared to LRU cache. In Config – C, only critical tasks’ data tagged as critical. In Config – CE, critical and essential tasks’ data tagged as critical.
comparison with LRU cache. This is mainly due to the task set used in these experiments being computationally intensive. Hence, the improvement in execution time due to better data cache performance when using LC cache accounts for a small part of total execution time.

7.3.1 Key Observations

The following key observations were made from the results presented in section 7.3:

- An LC cache can help reduce inter-task interference of non-critical tasks on critical tasks and improve the cache performance of critical tasks in certain configurations and modes of operation.

- When critical data is large compared to the size of cache, the interference among critical data will be high, which negates the benefits of using an LC Cache.

- Using an LC cache policy can increase the cache miss rate of some critical tasks when compared to LRU cache in certain configurations as shown in Figures 7.6(b) and 7.10(b).

- The LC cache architecture supporting changing cache configuration at runtime can be used to improve cache utilization when mode changes occur.
7.4 Hardware Monitor Infrastructure

From the experimental results presented in section 7.3, it is clear that a single cache configuration may not be optimal for all modes of operation. We can better utilize the cache by dynamically changing the configuration based on mode changes. Measuring the performance of the cache at runtime and dynamically adapting to changing operating conditions requires a hardware monitor infrastructure and feedback mechanisms.

In mixed criticality systems, the LC cache policy is used to give preference to critical data in the cache. To determine if we are benefiting from the current configuration of the LC cache, we need to know the cache miss rate of critical data. Knowing the overall cache miss rate of the application helps us evaluate the impact of the LC cache policy on the performance of non-critical tasks. A more fine grained tuning of cache is possible if we know the cache miss rate of individual tasks. However, having separate monitors for each task will incur additional resource overhead. One way to reduce the resource overhead of individual task monitors is to share a single set of monitors between multiple tasks. This involves saving and restoring the register values to either on-chip memory or main memory by the operating system during a context switch, which will increase the context switch overhead. In our experiments, separate hardware monitors were used to keep track of the cache performance of each task.

Based on the above premise, hardware monitors were implemented to measure the following metrics at runtime:

- Total cache hits and misses
- Critical cache hits and misses
- Cache hits and misses of each task

A high level block diagram of the hardware cache monitor infrastructure is shown in Figure 7.13. The real-time operating system can access and configure the cache monitors through a memory mapped interface.
7.5 Runtime Reconfiguration of the LC Cache

The multi-criticality workload (Table 7.1) used in this case study consists of eight periodic tasks with a hyper-period of 400ms. The experiments described in Section 7.3 were re-run and the cache miss rate during each hyper-period were measured over the course of the experiment, using the hardware cache monitors detailed in section 7.4. The hyper-period of the task set is the least common multiple of the tasks’ periods. The results showed, apart from the first hyper-period which includes the cold start behavior of the cache, the cache miss rate was the same for every hyper-period throughout the experiment. This behavior was observed for each mode of operation and all cache configurations. This occurs due to the periodic pattern of all tasks repeating after every hyper-period. Hence, the hyper-period of the task set can be used as a time-window to measure the cache performance in a given configuration.

The goal is to choose the best configuration for LC cache at runtime, which improves the performance of critical tasks when compared to using a LRU cache. Cache performance of critical data can be the primary metric for evaluation. Table 7.3 shows the normalized cache miss rates of critical data measured during the experiments detailed in Section 7.3. By looking
Table 7.3: Normalized cache miss rate of critical data for different modes of operation. LRU cache is used as a baseline for comparison.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Cache Size – 4K Bytes</th>
<th>Cache Size – 8K Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>LC - Config – C</td>
</tr>
<tr>
<td>Surveillance</td>
<td>100</td>
<td>49.87</td>
</tr>
<tr>
<td>Tracking</td>
<td>100</td>
<td>51.12</td>
</tr>
<tr>
<td>Engage</td>
<td>100</td>
<td>86.64</td>
</tr>
</tbody>
</table>

at the improvement in the overall cache miss rate of critical data shown in Table 7.3 and the cache performance individual tasks shown in Figures 7.1 to 7.10, these general observations were made:

- We see a clear advantage in using a configuration of LC cache when the overall miss rates of critical data is reduced by 40% or more in comparison with LRU cache. This generally results in improved cache performance for all critical tasks.

- When the improvement in overall cache miss rate of critical data is less than 20%, generally we see a minor or no improvement in cache performance of critical tasks.

Based on these observations, we suggest two ways to monitor and change the LC cache configuration at runtime:

1. **Offline Analysis** - The optimal cache configuration for each mode of operation can be calculated off-line and the runtime reconfiguration capability of our LC cache architecture can be used to change configuration when mode changes occur. The overhead in changing cache configuration at runtime may include flushing the cache, which can be accounted for during schedulability analysis. The hardware monitors could still be used to verify the desired cache behavior at runtime.
2. **Online Adaptation** - Starting with a base configuration (e.g. LRU mode), measure the cache performance for different LC cache configurations to find the best cache configuration in a given mode of operation based on a given set of metrics. The search space of different LC cache configurations can be decided off-line to limit the runtime overhead involved. In this method, depending on the hyper-period of the task set and number of different configurations to search, the time taken to achieve a steady state and the overhead involved may vary. Since changing cache configuration usually involves a cache flush, to exclude the cold start behavior in performance measurements we need to increase the time-window to at least two hyper-periods.

**Limitations.** The online adaptation method uses a time-window based approach to measure the performance of cache in a given configuration. This may not be feasible for systems with large hyper-periods, as hyper-periods with integer constraints are exponentially bounded with respect to the largest period. However, task models and algorithms like the ones proposed in [69, 14] can be used to reduce the hyper-period in periodic task systems. The online adaptation approach also does not work for sporadic or aperiodic task systems, where it is difficult to have a bounded time-window.
CHAPTER 8. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

8.1 Conclusions

Migrating functionality to hardware has shown to improve performance and predictability of real-time systems. But lack of flexibility and scalability of hardware solutions limits their wide spread use. The solutions proposed in the first part of this dissertation addressed these limitations through a hardware-software co-design approach. I presented a new hybrid priority queue architecture, where the hardware priority queue extends to software memory when its size exceeds hardware limits. I utilize 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, which can either execute in hardware or hybrid mode and support an arbitrarily large number of tasks was presented. The scheduler, when managed in hardware, showed up to 90% reduction in overhead and 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.

In the second part of this dissertation, a criticality aware cache design for mixed criticality real-time systems was presented. The cache architecture presented mitigates the inter-task interference arising from critical tasks sharing cache with non-critical tasks. A new cache replacement policy, called Least Critical, was proposed and implemented where critical tasks’ data is given higher preference in the cache. My design enables fine grained control over classifying task data as critical using critical address range (CAR) registers. The experimental results showed that the cache miss rate of a critical task was reduced by up to 70% when using an LC cache in comparison with an LRU cache. It was also observed that the critical task
does not benefit from using a LC cache over a LRU cache when intra-task cache interference is high, which occurs when a critical tasks' data is much larger than the cache size. This results in non-critical tasks being deprived of the cache which decreases overall application performance. I explored the feasibility of dynamic cache management using the runtime configuration capability of my LC cache architecture, which allows switching cache replacement policy between LRU and LC during runtime. I demonstrated the feasibility of using lightweight hardware cache monitors to observe the runtime performance of a cache configuration and to provide feedback for the operating system. I proposed mechanisms for runtime cache management, which decreased the response time of critical tasks and improved cache utilization under changing operating conditions.

8.2 Future Research Directions

8.2.1 Extend LC Cache Analysis to Instruction Cache

Instruction caches have been studied extensively in the context of real-time systems, which tends to show better spacial and temporal locality when compared to data caches. In this dissertation, we limited the analysis of my LC cache architecture presented in Chapter 6 to data caches only. A future work could be to evaluate the use of an LC Cache architecture in instruction caches. Extending the WCET analysis techniques described in [6, 20] to the LC cache replacement policy is another direction for future research.

8.2.2 Application of LC Cache in Real-Time Scheduling

The hybrid priority queues used in a scalable hardware-software scheduler design has demonstrated the benefit of migrating RTOS scheduling functionality to hardware, while supporting arbitrarily large number of tasks. Improved performance of hardware data structures can be attributed to two main factors: 1) acceleration of data structure operations 2) predictable and fast access to data stored in on-chip memory. Results presented in Chapter 6 have shown that the LC cache can provide fast and predictable access to critical data. The LC cache has a potential to support real-time scheduling in a similar manner and complement the hybrid priority
queue design. A comparative study could be conducted between using a hardware accelerated priority queue vs using a LC Cache with queue elements tagged as critical, in the context of real-time scheduling. This would help us understand the trade-offs between using hardware data structures vs guaranteeing better access times through caching schemes. The LC cache can also be used to further improve the performance of the hybrid priority queue presented in Chapter 3. This can be accomplished by tagging extended queue elements as critical, which will provide better access times even to the extended priority queue elements.

8.2.3 Heuristics and Search Algorithms for Dynamic Cache Management

The ability to measure the runtime performance of the cache through hardware cache monitors and the flexibility of the LC cache architecture, which allow us to configure the cache in multiple ways, provides an interesting dimension to extend this work. In the case study described in Chapter 7, two simple configurations of the LC cache were considered where either all critical tasks’ data was tagged as critical or both critical and essential tasks’ data was tagged as critical. This is a simplistic assumption, which may not always yield an optimal cache configuration. One way to extend this work is to relax this assumption and come up with heuristics to find the optimal set of tasks’ that can be tagged as critical given some bounds on overall cache performance. Another potential path for future work could be to consider sporadic and aperiodic task sets, where a time-window based measurement approach may not be feasible. This would enable the use of the dynamic cache management techniques for a wider variety of applications.

8.2.4 Towards a Criticality Aware Adaptive Hardware Platform

Mixed criticality systems (e.g. avionics, automotive control) often operate in a dynamic environment, where computing and other resource requirements can change during runtime. Adaptability, survivability, and graceful degradation are important aspects of any mixed criticality system. Developing hardware components that are aware of application criticalities and support dynamic reconfiguration enable us to build more adaptable systems, which improves system robustness [9]. The LC cache architecture presented in Chapter 6, which supports run-
time reconfiguration is a step towards this direction. The outcomes of the case study presented in Chapter 7 indicate that a criticality aware platform component, which supports runtime reconfiguration can improve adaptability of mixed criticality systems. There have been efforts to investigate new CPU architectures for mixed criticality systems [89]. Future research could explore criticality aware architectures for platform components such as the memory management unit, memory controller and shared buses.
Bibliography


