PERFORMANCE AND COST EFFICIENT RELIABILITY FRAMEWORK FOR MULTICORE ARCHITECTURES

by

Sanem Arslan Yılmaz

B.S., Computer Engineering, Marmara University, 2009

M.S., Computer Engineering, Boğaziçi University, 2011

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2017

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis advisors Prof. Oğuz Tosun and Prof. Haluk Topcuoğlu for their support, encouragement, and guidance throughout my graduate study and completion of this thesis. I would like to thank my last-year advisor, Prof. Can Özturan for his help and suggestions. I also want to thank Prof. Mahmut Kandemir (from Pennsylvania State University) for his valuable feedback and guidance during my dissertation. I would also thank to Prof. Arda Yurdakul, Assoc. Prof. Alper Şen, Assoc. Prof. Deniz Turgay Altılar, and Assoc. Prof. Gürhan Küçük for their participation in my thesis jury, their useful comments, and feedback.

I gratefully acknowledge the financial support of The Scientific and Technological Research Council of Turkey (TUBITAK) BIDEB 2211. This work was also supported by TUBITAK through a research grant (Project Number:113E530).

I would like to thank my colleagues from the Department of Computer Engineering at Marmara University for their support and friendship. In particular, I would like to thank Betül Boz and Işıl Öz for their invaluable support and friendship throughout the years.

I would like to offer my thanks to my father Hüseyin Arslan and my sister Azime Evrim Arslan for their endless love and great support during my dissertation. I am eternally grateful to my mother Mukaddes Arslan (now deceased, greatly missed) who always believed in my ability to be successful in the academic arena. The biggest credit goes to my dear husband, Tanju Yılmaz. I am grateful for his love, patience, sacrifices, and understanding. This thesis (along with everything else I have in life) was made possible thanks to your support.

ABSTRACT

PERFORMANCE AND COST EFFICIENT RELIABILITY FRAMEWORK FOR MULTICORE ARCHITECTURES

Modern architectures become more vulnerable to soft errors with technology scaling. Enabling fault tolerance capabilities on all cache structures in a system is inefficient in terms of performance and power consumption. In this study, we propose an enhanced protection mechanism for code segments, which are critical in terms of reliability, by utilizing asymmetrically reliable cores under performance and power constraints. Our proposed system contains at least one high-reliability core, which has an ECC-protected L1 cache, and several low-reliability cores, which have no protection mechanisms. Our framework protects only reliability-based critical code regions of each application, which are determined based on critical data usage, user annotations, or static analysis. In our first attempt, the framework dynamically assigns the software threads executing critical code fragments to the protected core(s) by using the First Come First Served (FCFS) algorithm. Our experimental evaluation shows that the proposed approach takes advantage of protecting only critical code regions and presents comparable performance and reliability results with fully protected systems having lower power consumption and cost values for a set of applications. However, the FCFS-based scheduling algorithm may degrade the system performance and unfairly slow down applications for some workloads. Therefore, a set of scheduling algorithms is proposed to improve both the system performance and fairness perspectives. Various static priority techniques that require preliminary information about the applications and dynamic priority techniques that target to equalize the total time spent of applications on the protected core(s) are presented as part of this thesis. Extensive evaluations using multi-application workloads validate significant improvements of proposed scheduling techniques on system performance and fairness over the FCFS algorithm.

ÖZET

ÇOK ÇEKİRDEKLİ MİMARİLERE YÖNELİK PERFORMANS VE MALİYET ETKİN GÜVENİLİRLİK SİSTEMİ

Modern mimariler gelişen teknoloji ile geçici hatalara karşı daha savunmasız hale gelmiştir. Bir sistemdeki tüm önbellek yapılarını seçici olmaksızın korumak, performans ve enerji tüketimi açısından önemli bir ek yük getirir. Bu tez kapsamında performans ve güç tüketimi kısıtları altında asimetrik olarak güvenilir önbelleklere sahip çok çekirdekli bir sistem kullanılarak, yalnızca güvenilirlik açısından kritik olan kod parçalarını koruyan bir mekanizma önerilmiştir. Önerilen sistemimiz L1 önbellek yapılarında ECC korumasına sahip en az bir yüksek güvenilirlikli çekirdek ve önbellek yapılarında koruma bulunmayan birden fazla düşük güvenilirlikli çekirdeklerden oluşmaktadır. Bu tez kapsamında, güvenilirlik temelli kritik kod bölümleri, kritik veri kullanımı, kullanıcı ek açıklamaları ve statik analiz temel alınarak farklı yöntemlerle belirlenmiştir. İlk yaklaşımımızda, kritik kod bölümlerini çalıştıran uygulama iş parçacıkları First Come First Served (FCFS) tabanlı bir çizelgeleme algoritması ile dinamik olarak korunan çekirdeğe eşlenmiştir. Yapılan deneysel çalışma sonucunda, önerilen yaklaşımımız bir dizi uygulama için tamamen güvenilir sisteme yakın güvenilirlik ve performans sonuçları ile daha düşük güç tüketimi ve maliyet değerleri sunmuştur. Bununla birlikte FCFS tabanlı çizelgeleme algoritması bazı iş yükleri için düşük sistem performansı ve eşitlik sonuçlarına sahiptir. Bu tez kapsamında, sistem performansı ve eşitlik perspektiflerini iyileştirmek için, uygulamalar hakkında ön bilgi gerektiren önceliğe dayalı çizelgeleme teknikleri ve korunan çekirdek(ler) üzerindeki harcanan toplam süreyi eşitlemeyi hedefleyen dinamik çizelgeleme teknikleri sunulmuştur. Yapılan deneysel değerlendirme sonucunda, önerilen çizelgeleme tekniklerinin FCFS algoritmasına kıyasla sistem performansını ve eşitlik sonuçlarını önemli ölçüde iyileştirdiği gözlemlenmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

AS	Average speedup
$criticality_i$	Criticality value of function i
E_{bus}	Energy value for off-chip bus
E_{cod}	ECC encoder energy
E_{dec}	ECC decoder energy
E_{mem}	Memory access energy
E_{proc}	Processor energy
$Eread$	Energy value of read access
$E_{readECCoff}$	Energy values of the read accesses for the unprotected cache
$E_{readECCon}$	Energy values of the read accesses for the protected cache
ET_i	Execution time of function i
$Ewrite$	Energy value of write access
$fixity_i$	Fixity value of function i
HS	Harmonic speedup
M_i	Number of misses for cache i
$M_{protected}$	Number of misses for the protected cache
$N_{protected}$	Number of cache accesses for the protected cache
$Nread_i$	Number of read accesses for cache i
$N_{unprotected}$	Number of cache accesses for the unprotected cache
$Nwrite_i$	Number of write accesses for cache i
$OutDegree_i$	Number of callee functions in function i
$priority_i$	Priority value of function i
$R_{protected}$	Number of replacements for the protected cache
$slack_i$	Slack value of function i
$T_i(alone)$	Execution time of the application i with no scheduler
$T_i(approach)$	Execution time of the application i with the proposed scheduling technique
$T_i(baseline)$	Execution time of the application i with the FCFS technique
$vulnerability_i$	Vulnerability value of function i

$W_{protected}$	Number of write accesses for the protected cache
α	Weight of the vulnerability metric on the criticality metric

LIST OF ACRONYMS/ABBREVIATIONS

AFCFS	Adapted First Come First Served
CFS	Completely Fair Scheduler
DFT	Discrete Fourier Transform
DVFS	Dynamic Voltage and Frequency Scaling
ECC	Error Correcting Codes
EDC	Error Detection Codes
EPET	Earliest Possible Execution Time
EST	Earliest Start Time
FCFS	First Come First Served
FFM	Fast Multipole Method
FFT	Fast Fourier Transform
IPC	Instruction per Cycle
LGFS	Largest Gang First Served
LST	Latest Start Time
LTF	Largest Task First
RCS	Reliability-based Critical Section
RCSS	Reliability-based Critical Section Scheduler
SDR	Silent Data Corruption
SECDED	Single Error Correction and Double Error Detection
TMR	Triple Modular Redundancy
TVF	Thread Vulnerability Factor

1. INTRODUCTION

Modern architectures are progressively vulnerable to transient and permanent errors because of persistent decrease on transistor sizes, high transistor density per chip, and high operating frequencies [1,2]. The impact of such errors can be quite varied and dramatic depending on the target application domain. For instance, in a safety-critical application (e.g., a program that controls a nuclear plant or a missile), results of even a single transient error can be catastrophic. On the other hand, in a molecular dynamics application with self-correcting capability (e.g., one that uses iterative solvers), an error can notably increase the running time of the program, in spite of the fact that the program may still finish successfully. Based on above observations, reliability (fault tolerance) should be considered as a fundamental and first-class metric in hardware and software designs.

Previous studies explore the fault tolerance strategies in different layers of hardware and software. A certain level of success is accomplished by those efforts in predicting and mitigating hardware faults. Hardware-only approaches for reliability improvement lacks of application-level information. As a result, all memory accesses must be protected as they have the same criticality level (which is clearly not the case in reality). Because of this conservative situation, the provided reliability has overheads of cost, performance and power consumption. On the other hand, software-only approaches are not aware of the runtime information to enable making ideal protection decisions, dynamically.

A temporary condition in a semiconductor device may alter the stored data in memory, which results in a soft error [3]. This sort of error randomly occurs and may modify the data or terminate the execution of a target program. Alpha particles, low energy particles from the packaging material, and high energy particles from the cosmic rays are the main causes of this type of error [2]. Having large area of the logic relative to other parts of the chip [1,4] and high transistor density [5] makes cache structures more vulnerable to soft errors.

Utilizing Error Correction Codes (ECC) is a general method to ensure protected cache memories. However, applying ECC protection in all cache structures may result in significant overheads in terms of area, power, and cost [2].

The multicore architectures with the unprecedented levels of transistors will require more appropriate measures to be taken for the reliability problem. In the light of the observations specified, we propose a performance and cost efficient reliability framework aiming to maximize the reliability with using minimum reliability hardware under the performance, power and cost constraints. In our proposed framework, instead of protecting all data uniformly at the same level of reliability, we can determine the Reliability-based Critical Sections (RCS) which specifies the portions of the program that should be protected. Then, these portions are preserved more conservatively than the remaining parts.

We consider an asymmetric chip multiprocessor (or heterogeneous chip multiprocessor) consisting of at least one reliability-aware core and a set of less reliable cores. While the protected core(s) use most of the chip area by containing larger amount of fault-tolerant logic and components, the unprotected cores do not provide any protection mechanisms. An efficient way to achieve higher hardware reliability is to utilize protected core(s) for the Reliability-based Critical Sections and to map the less-critical parts of the program on the remaining unprotected cores. By using asymmetrically reliable cores we can provide maximum reliability enhancement using minimum reliability hardware. In our framework, the protected core is not reserved for a particular application or thread, and the application threads can visit the high reliability cores at runtime according to their critical sections of code. Therefore, an efficient scheduling method is required to dynamically map application threads to the protected cores. We started with a primitive scheduling technique based on First Come First Served (FCFS) policy. Then, alternative scheduling techniques with different characteristics are implemented and evaluated in terms of system performance and fairness.

1.1. Thesis Contributions

This thesis proposes an enhanced protection mechanism for the Reliability-based Critical Sections of the given program by utilizing an asymmetric chip multiprocessors containing cores with different protection levels. The contributions of the thesis can be listed in three categories:

Proposing Asymmetrically Reliable Caches for Critical Data [6, 7]:

- We propose two different heterogeneous chip multiprocessors in this part. In the first approach, a chip multiprocessor which has one high reliability and a set of low reliability cores is proposed. While ECC protection is applied on individual L1 cache structure of the high reliability core, there is no cache protection mechanism for the less reliable cores. An application input is analyzed statically a priori, and a portion of the input is determined as critical data specifically for the given application. The code fragments that access critical data are determined as the critical code fragments that need to be protected.
- In the second approach, we propose a chip multiprocessor which has one high reliability, one middle-level reliability and a set of low reliability cores. While ECC protection is applied on individual L1 cache structure of the high reliability core, parity check is utilized for the middle-level reliability core. An application input data is analyzed statically and classified as critical, semi-critical and non-critical. The applications can utilize high, medium-level or low reliability cores based on critical, semi-critical, and non-critical data.
- We present a framework which dynamically allocates application threads to asymmetrically reliable cores based on critical data usage for both cases. We present an FCFS-based scheduling method for the queue structure of the high reliability core.
- A simulation-based fault injection framework is implemented to perform a detailed experimental study on both proposed asymmetrically reliable caches and traditional caches.

Experimental studies performed on applications with various characteristics validate the proposed framework for asymmetrically reliable caches. The partially safe cache configuration performs significantly better than the unsafe configuration with respect to reliability, and the safe configuration with respect to performance and energy consumption.

Proposing Asymmetrically Reliable Caches for Critical Code Regions [8, 9]:

- We propose two different approaches to determine the reliability-based critical code regions of the applications. As a first approach, a programmer can annotate the reliability-based critical code regions of the application. Selected applications are profiled and the functions that cover 90% of total execution time (that might be three or four functions) are assumed as critical regions for each application. Then, each of these functions are protected using asymmetrically reliable caches.
- As a second approach, the critical code fragments are extracted by using static analysis. Each application is profiled, and the execution time percentages and the call graph of the functions are generated. Then, high-priority functions to be protected are determined with vulnerability and criticality analysis on the profiling results.
- For both cases, a comparative study is performed using different numbers of protected cores to show the efficiency of our framework by using a diverse set of applications from benchmarks. Our experimental evaluation shows that the proposed approach takes advantage of protecting only functions with higher priority and presents comparable performance and reliability results with fully protected systems, while providing lower power consumption and cost values for a set of functions.

Proposing Different Scheduling Techniques for Our Framework [10]:

- We propose different scheduling techniques with various characteristics to map application threads on the protected cores.

Six priority-based scheduling techniques are presented in which the applications are prioritized based on their execution order, cache miss rates, number of requests sent to the protected core(s), or total burst time spent on the protected core(s). These static priority techniques require executing the applications in advance to determine the priority-level of the applications.

- We propose two dynamic priority techniques: equal-time and equal-progress. The former one targets to equalize the amount of time for each application on the protected core(s) at every scheduling point whereas the latter one targets to equalize the progress of requests for each application on the protected core(s). Additionally, the priority-levels of the applications are updated dynamically at runtime in these techniques.
- We propose the threshold-based priority technique, which uses a static priority technique by considering last scheduled application.
- Evaluations with different workloads show that our priority-min-burst method that prioritizes the applications with low total burst time on the protected core(s) presents an average of 51.4% (up to 70.8%) better performance with respect to average speedup metric and an average of 36.7% (up to 65.4%) better fairness with respect to Jain's fairness index on a 16-core system with one protected core. On the other hand, our equal-time scheduling technique shows an average of 44.7% better performance with an average of 20.5% better fairness results relative to the FCFS algorithm. Overall, these results validate the usage of scheduling algorithms presented in this part for providing high performance and fairness values, in place of the FCFS algorithm.

1.2. Outline of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents a variety of studies proposed in the literature. We describe our framework in general in Chapter 3. Chapter 4 demonstrates the implementation and evaluation of the main mechanics of our proposed approach with critical data. In Chapter 5, we present our framework with asymmetrically reliable caches for critical code regions.

Chapter 6 demonstrates the scheduling opportunities of our framework by providing nine different methods. We conclude the document by providing summary and directions for future work in Chapter 7.

2. LITERATURE SURVEY

We present a summary of related work in two subsections. Hardware-based, software-based and hybrid approaches for the reliability problem are summarized in the first subsection. It also includes a review on data and code criticality approaches. In the second part, various scheduling methods that are frequently used in the literature are explained.

2.1. Literature Survey on Reliability

Hardware-based and software-based reliability approaches have been extensively studied in the literature. Physical duplication of hardware units is a common method in hardware-based techniques [11]. Triple Modular Redundancy (TMR) is a well-known method of three identical pieces of hardware components are used [12]. With the evaluation of majority voting system, the majority of these results are considered as accurate and the system is continued to the operation with this outcome. In such a case, errors may occur in a part of the system without affecting the correct functioning of it. While such techniques use fault masking methods, there are studies dealing with error detection and error correction techniques. Error Correcting Codes (ECC) based techniques are widely deployed technology in the industry that are designed in different ways to protect on-chip memory and cache structures as well as communication links of NoC. Variable Strength Error Correcting Codes (VS-ECC) change ECC strength on different cache lines with online testing [13]. In another study, Hi-ECC provides protection at a coarse granularity by reducing the cost of strong ECC [14]. Virtualized ECC [15] provides adaptable memory protection by mapping ECC to areas that are noticed by software in memory. Memory mapped ECC [16] stores the error protection codes in main memory to decrease area overhead in last level caches.

Software-based techniques provide software-level solutions by putting additional instructions to the original program. Information, data, and time redundancy methods are widely used techniques in this level.

Adaptive redundancy is provided to increase data reliability in shared caches [17]. In addition to full redundancy, partially protected cache is proposed, which ensures a few sections of the cache against soft errors [18]. Applying only data-intensive multimedia applications or considering of single-processor architectures are the fundamental limitations of those studies mentioned above. In various studies, operating system data structures are protected by software-based fault tolerance [19] and checkpoint recovery [20].

Cross-layer approaches that provide solutions at the architecture and application level for the reliability problem are also presented in the literature [21]. The Relax framework provides software level protection for hardware faults [22]. They extract vulnerable code regions as the functions that can be ignored within a program. If a hardware fault occurs while executing these regions, they are either re-executed or the results produced by these regions are ignored. A programming model, EnerJ, is presented by classifying application data as critical or approximate [23]. In this study, approximate data calculations are performed in low-energy, low reliable mode, while critical data calculations are performed in high-energy, high reliable mode.

A programming model, Rely, is presented in which the user can figure out reliability requirements in application level [24]. In this method, probabilistic hardware models are developed to meet the reliability requirements. The authors also proposed a system, Chisel, in which reliable data and operations can be determined automatically [25]. Additionally, a flexible architecture is presented using reliable processors to control algorithmic flow and unreliable processors to run slave tasks [26]. While this work requires rewriting the application source code, the overhead of using reliable processor can be decreased by using many parallel threads. Rehman *et al.* [27] propose a system using heterogeneous error recovery cores to select a task among various task alternatives, where vulnerable tasks are bound to the reliable cores while robust tasks are bound to the unreliable cores. They also provide a compiler-based solution that reduces the vulnerability of critical instructions on unreliable hardware [28]. Yetim *et al.* [29] propose a system that can convert the fatal errors in the inter-thread communication into application-level data errors.

A model-driven method is also proposed for fault-tolerant embedded systems [30]. In this model, design space exploration methods are studied using application-level reliability requirements and hardware platform information.

Researchers have also examined asymmetric multicore architectures in the literature. An asymmetric architecture is proposed in which a high-performance core (called large core) is used to execute critical sections and low-performance cores (small cores) are used to execute other instructions [31]. Thus, the execution of the critical sections, which have to be executed in serial, is accelerated using a high-performance core and the thread waiting times are decreased by this approach.

Asymmetric multicore architectures are also studied in terms of reliability in the literature. A multicore system is proposed where different fault tolerant cores are used for different applications [32]. In this work, critical applications are executed on high-reliability cores, while non-critical applications are executed on low-reliability cores. Application criticality is determined from the scratch and critical applications run on high-reliability cores until the end of the execution. Heterogeneous-reliable memory implementations are also proposed in the literature [33]. In this study, application data are classified in two groups referred to as less vulnerable and more vulnerable data. While more vulnerable data are stored in protected memory, less vulnerable data are stored in unprotected memory. This study has been proposed to reduce the memory cost of the servers.

There are various studies in the literature regarding the determination of critical code regions in applications. Burtscher *et al.* [34] assume that the code regions that have higher execution time percentages are more critical than the other parts. Subotic *et al.* [35] claim that the code regions that could be improved with performance optimizations and those that have maximum speed up in the case of optimizations are selected as the most significant regions. Carbin *et al.* [36] extract critical and non-critical code regions by utilizing flexible input fuzzing methods in which different inputs are given to the program and critical regions are determined depending on the program path.

Duque *et al.* [37] propose a system that can perform an application-specific criticality measurement using static task graphs.

There are significant differences between the studies presented in this section and the system we have proposed. In our proposed system, a high reliability core is not reserved for a particular application or thread. Various threads can use high reliability core(s) at runtime. Therefore, dynamic allocation of application threads and a scheduling technique are required in our work. On the other hand, protection techniques are applied to the individual cache structures of the cores, rather than the external storage as in the studies above. Furthermore, critical code regions are determined in several ways using real applications, rather than using ready task graphs.

2.2. Literature Survey on Scheduling Techniques

Scheduling is a very popular research area in the literature where a large number of policies have been implemented based on various perspectives. Scheduling techniques are studied as part of the reliability-based research in the literature [37–42]. Duque *et al.* [37] propose a reliability-aware task scheduling that uses several metrics such as core reliability, task vulnerability, and task criticality. In this study, time-correlated fault behavior of the applications are modeled and different priorities are given to different tasks. The task vulnerability metric is defined as affecting the task outcome of a possible error. A task criticality metric is defined as the effect of a possible error on the execution time of the application. The task vulnerability and task criticality metrics are calculated based on the task graph information. Using these metrics, the runtime scheduler assigns more vulnerable and critical tasks to the more reliable cores, dynamically. In this study, ready task graphs are used rather than real applications. The authors propose a static reliability-aware scheduling technique in another study [38]. The aim of this work is again to assign more critical and vulnerable tasks to more reliable cores. In the event of a possible error, it is assumed that the faulty task will be re-executed. Then, the level of reliability is defined as the expected increase in execution time in case of a possible error. In this study, List scheduling, which consists of a task prioritization and a core assignment stages are used.

The task prioritization is based on Earliest Possible Execution Time (EPET). The highest priority among the tasks is given by the lowest EPET value.

Chantem *et al.* [39] propose a reliability-aware task assignment and scheduling algorithm that uses prior knowledge of the desirable thermal profile of a heterogeneous system where it adopts the Largest-task first (LTF) algorithm. Tasks are sorted in decreasing order according to their energy consumption and assigned to the core with the lowest total energy consumption. Coskun *et al.* [40] propose a reliability-aware job scheduling technique for chip multiprocessors by considering power management. Thermal cycling - repeatedly heating and cooling the processor - causes errors in the system. Several migration and Dynamic Voltage and Frequency Scaling (DVFS) techniques are utilized in their reliability-aware scheduling algorithm to capture the effects of thermal failures. Within the migration-based techniques, a thermal threshold is set for the cores, and the core exceeding this value sends its task to the core with the lowest heat. As another migration technique, the task with the highest Instruction per Cycle (IPC) value is sent to the core with the lowest heat. Other than that, the task with the highest IPC value is sent to the core with the lowest heat based on the location (assumed that the cores on the corner have less heat, and the cores on the center have higher heat).

There are several scheduling studies for heterogeneous multiprocessor platforms and these have attracted significant research interests [43–50]. Li *et al.* [43, 51] propose a scheduling algorithm for frame-based tasks on heterogeneous multiprocessors, where they use DVFS-enabled processors to decrease energy consumption. Tang *et al.* [44] propose a reliability-aware scheduling algorithm for heterogeneous systems in which they prioritize tasks based on reliability overheads. The overheads are caused by task duplication in order to increase system reliability. In another study, they propose an energy- and reliability-aware scheduling algorithm for parallel applications, where they take the scheduling decisions based on energy consumption, schedule length, and reliability [45].

Additionally, there are several fairness-aware scheduling techniques proposed in the literature [52–54]. Wang *et al.* [52] propose a scheduling algorithm that balances contention and reservation. All the threads belonging to an application are co-scheduled together. They also allow co-scheduling of multiple applications that can co-exist effectively. They give weights to the applications and co-schedule them with the weight values under the degree of contention threshold. Their method is better than the Linux Completely Fair Scheduler (CFS) in terms of fairness and system throughput. Craeynest *et al.* [53] propose a fairness-aware scheduling technique for heterogeneous systems, considering barrier-synchronized multi-threaded workloads. In their methods, they schedule threads such that they have equal time or equal progress on each core type. Their techniques improve fairness and performance more than pinned scheduling.

There are various studies which utilize sophisticated methods in memory scheduling [55–60]. Subramanian *et al.* [59] propose the Blacklisting Memory Scheduler in which they group the applications with high number of successive requests and put them to the blacklist. They prioritize the remaining applications over the blacklisted applications and clear the blacklist at certain time intervals. Usui *et al.* [60] proposes deadline-aware high-performance memory scheduler to meet hardware accelerators deadlines by presenting high performance. They give different ranks to the memory-intensive/non-intensive CPU applications and hardware accelerators with short and long deadline period in order to achieve high performance for heterogeneous systems.

There are several studies based on gang scheduling [61–65], in which the tasks of a parallel job are organized as a group, called as a gang, and executed simultaneously on various processors [61]. Papazachos *et al.* [61] study the performance of gang scheduling policies such as Adapted First Come First Served (AFCFS) and Largest Gang First Served (LGFS). They also implement migration based extensions of the two techniques for multicore clusters. Manickam *et al.* [62] propose a fair and efficient gang scheduling algorithm that is the modified version of AFCFS. A variable, called bypass count, is kept for each gang, which stores the number of gangs that have bypassed it for execution in the queue.

When this value exceeds a certain threshold, the highest priority is given to this gang. They guarantee that each gang can be scheduled in a predictable time period.

In addition to these studies, there are a set of scheduling algorithms that extends the Shortest Job First (SJF) algorithm. Mi *et al.* [66] propose a scheduling algorithm that approximates SJF scheduling without requiring any knowledge of job service times. They use a prediction mechanism to classify jobs as short or long based on the correlation of the successive task service times, and they delay the large jobs by moving them to the end of the queue. Casale *et al.* [67] propose a scheduling approach for workloads with correlated task sizes. They estimate the size of forthcoming tasks based on correlations in previous scheduling policies and prioritize the short-size tasks based on this information.

3. OUR RELIABILITY FRAMEWORK FOR MULTICORE ARCHITECTURES

In this section, we present the scope and limitations of our proposed framework which provides only as much reliability as needed for the application. Our proposed framework allows the application programmer to determine reliability-based critical sections of a given code. These regions in our framework indicate the parts of the code that are critical from reliability perspective and these have to be protected. As we know, all the instructions and memory accesses do not have the same criticality. Some instructions, data structures or functions may need more reliability requirements than the others. These sections are called as Reliability-based Critical Sections (RCS). If a transient fault occurs in these parts, it may cause the failure of a system whereas if it occurs in non-critical parts, the system undergoes without any failure. If the criticality of instructions is not known, the system needs to be conservative to all memory accesses as having the same criticality (which is clearly not the case in reality). Being conservative in turn increases both cost and performance overheads of the provided reliability.

The proposed framework runs on homogeneous ISA, speed and performance but asymmetrically reliable cores which have different fault tolerant hardware. Asymmetrically reliable cores consist of at least one high reliability core and several low reliability cores. Fault tolerance mechanism on the high reliability core is based on Error Correcting Codes (ECC) that is implemented on individual cache structures. In this framework, the software threads that execute critical code fragments are dynamically mapped to the protected core(s), and the software threads that execute non-critical code fragments are mapped to the unprotected cores during the execution. Therefore, sufficient reliability can be guaranteed using minimum additional hardware to reduce reliability costs. This framework provides a protection mechanism only for the reliability-based critical sections in a conservative way to provide the required reliability.

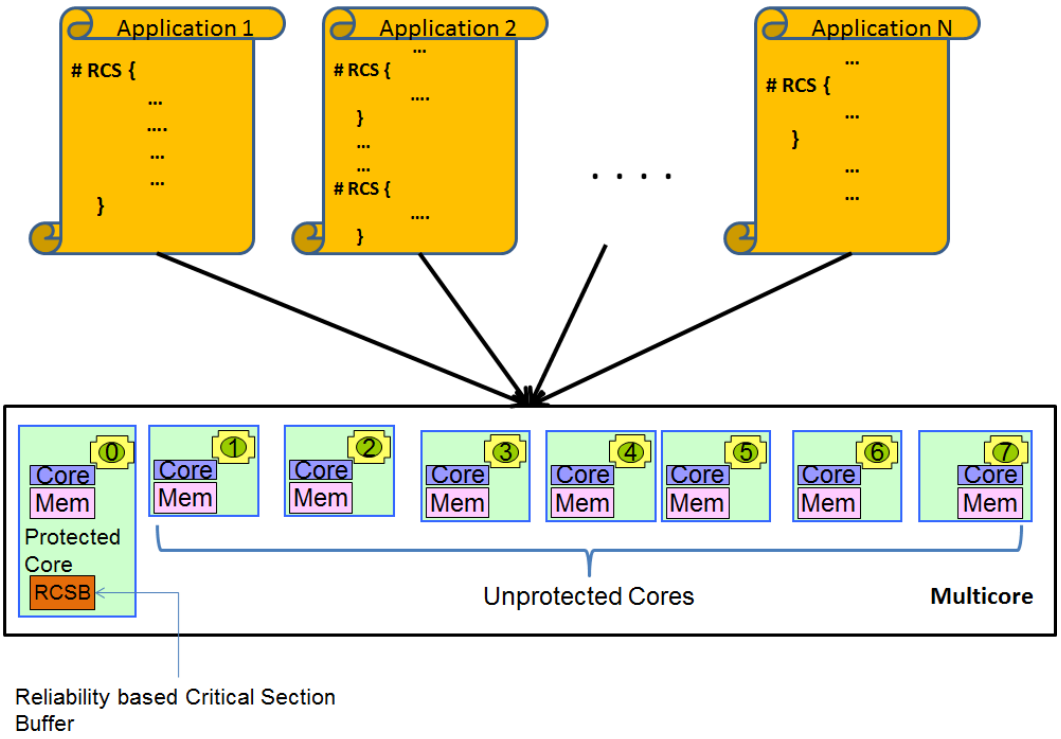


Figure 3.1. Proposed framework.

Figure 3.1 presents the components of our framework. There are multiple applications annotated by the programmer with RCS keywords. An example system consisting of one protected (Core 0) and seven unprotected cores (Cores 1-7) is presented in the figure. The framework executes non-critical code fragments on the unprotected cores (Cores 1-7) and the critical code fragments on the protected core (Core 0). Different applications may have different criticality characteristics. In Figure 3.1, the application 1 has several instructions in its critical code fragments, while the application 2 has more than one critical code fragment in its program code.

The protected core is not reserved for a particular application in the framework. Various threads can use the protected core during the execution. In this way, we have to assign application threads to the protected core dynamically, which requires a scheduling technique in our system. Application threads start the execution on the unprotected cores. When a thread accesses a critical code fragment, it presents a request to run on a protected core.

The usage of the protected core is handled by a scheduler called as Reliability-based Critical Section Scheduler (RCSS), which collects requests issued by the application threads. When the protected core is idle, the requesting thread is mapped to it and the thread executes the critical code fragment. Then, it returns to the initially requested core which is an unprotected one. As a first attempt, First Come First Served (FCFS) based scheduling approach is considered for the application threads. Then, various scheduling techniques with different characteristics are implemented and evaluated.

The reliability-based critical sections are determined based on critical data usage, user annotations, or static analysis, each of which identifies critical code fragments differently throughout this thesis. As the first attempt, an application input is analyzed statically a priori, and a portion of the input is determined as critical data specifically for an application. The code fragments that access critical data are determined as the RCS that need to be protected. Secondly, the user can annotate the application program manually to determine the RCS that need to be protected. In this work, the application is profiled and the function names that cover 90% of total execution time are determined. These functions are treated as the RCS that need to be protected. As the third approach, the RCS are extracted by using static analysis. Each application is profiled using the GNU profiler, gprof, and the function execution time percentages and the call graph of the functions are generated. Then, the high-priority functions to be protected are determined with vulnerability and criticality analysis on the profiling results.

4. ASYMMETRICALLY RELIABLE CACHES FOR CRITICAL DATA

In this chapter, we implement and evaluate the main mechanics of our proposed framework with asymmetrically reliable caches. The input data of the application is examined and reliability-based critical code regions are determined according to the usage of the critical data. The proposed multicore architecture is investigated in two sub-chapters: (i) the system with a single type of protected core and (ii) the system with two types of protected cores where the former one provides only high reliability core and the latter one provides a high and a middle-level reliability cores.

4.1. System with Single Type of Protected Core

In this part, we implement our proposed framework with asymmetrically reliable caches having one high reliability core and a set of low reliability cores. The details about determining reliability-based critical sections based on critical data are given in Section 4.1.1. We present the features of our asymmetrically reliable multicore system providing details of architecture and execution models in Sections 4.1.2 and 4.1.3, respectively. The details of cache protection mechanism with ECC are given in Section 4.1.4. The experimental setup used in our evaluations and results from our experimental analysis are given in Section 4.1.5.

4.1.1. Reliability-based Critical Data

An application's memory accesses may have different reliability requirements. As in the case of a human face in a face recognition program, some regions of an image may be more critical to the user or programmer. A soft error occurrence in one of the pixels on the face may cause the image to be misidentified; however, a fault in the pixels outside the face may not be so critical and it may only cause quality degradation on the image.

Figure 4.1. An output of a fault injection experiment.

As it will be explained in the Section 4.1.5, fault injection experiments are performed on Susan smooting, an image recognition application from the ParMiBench benchmark suite [68]. An example erroneous output of the application is shown in Figure 4.1. The injected fault hits to the pixels on the face and causes this faulty output.

In our proposed framework, the reliability requirements (criticality) of all data should be distinguished in advance. When one of the application threads accesses to a piece of code that uses critical data, then the thread is moved to the high reliability core. Threads running the code fragments that use non-critical data continue the execution on the low reliability cores. Application threads may access these critical regions at different frequencies throughout their lifetime, which shows the frequency of visits to the high reliability core. In our system, the criticality of input data is determined by the user.

If the criticality of the data used by the application is unknown, then all memory accesses should be treated as having the same criticality level. In this sense, protecting all data brings additional overheads in terms of performance, energy consumption and cost. In our framework, the required reliability is ensured by preserving only the code regions that utilizes the critical data.

4.1.2. Architecture Model

Our system consists of identical cores that have similar micro-architecture and speed but they differ by the fault tolerance method used in their private L1 caches. The fault tolerance mechanism utilized in the cache of high reliability core is based on ECC which is a widely used technique in the industry and can detect double bit errors while correcting single bit errors.

The architecture used to test the validity of the system is a 4-core system. While there are individual L1 data and instruction caches for each core, there is a single shared L2 cache in our hardware model. There are three different cache protection configurations in our system: unsafe, safe, and partially safe. ECC protection is available for each L1 data cache of the cores in the safe configuration, while no cache structure is protected in the unsafe configuration. In the case of a partially safe configuration, only the L1 data cache of the high reliability core has ECC protection. Here, we defined criticality in terms of critical data; therefore, we provide protection on data caches.

4.1.3. Execution Model

Firstly, application threads start the execution on the low reliability cores. When a thread accesses a piece of code that uses critical data, it sends a request to use the high reliability core. Since multiple threads may send requests to execute on the high reliability core, the thread management is handled by a scheduler. There is a queue that collects requests sent by the threads running on the low reliability cores. If the high reliability core is idle, the requesting thread is moved to that core and the code fragment containing the critical data is executed. After finishing the code fragment, the thread sends an acknowledgment message to the scheduler. Then, the scheduler moves it back to the initial requesting core. If the high reliability core is not available, then the requesting thread begins to wait in the request queue. Queue structure and thread-to-core mapping are handled by a scheduler, referred to as Reliability-based Critical Data Scheduler (RCDS).

Our system uses a First Come First Served (FCFS) based scheduling algorithm that prioritizes older requests over newer requests in the queue at the first attempt.

Figures 4.2 and 4.3 show the execution scenario of application threads and the RCDS thread, respectively. When an application thread encounters a code region that uses critical data, it appends itself to the requestList for a high reliability core and waits the response from the RCDS thread. In our system, there might be multiple high reliability cores, each with a separate RCDS thread. When a request is added, the RCDS thread receives it and informs the corresponding thread with the core number of high reliability core. Then, the application thread maps itself to the high reliability core. Once the thread finishes the respective code region, it binds itself to the initially requested core and informs the corresponding RCDS thread about completion. After receiving this notification, RCDS removes the corresponding request from the queue. If the high reliability core is not available, the RCDS thread cannot receive a request since it gets stuck at the tenth line of Figure 4.3. These algorithms are executed by more than one thread at the same time, and all operations on shared variables are maintained in a mutually exclusive way.

4.1.4. Cache Protection Scheme

Cache structures are more susceptible to soft errors due to the large area of the logic compared to other parts [69]. ECC is a widely used method to protect cache structures. As an example for ECC protection, Single Error Correction and Double Error Detection (SECDED) can correct single bit errors and detect double bit errors. Since the majority of soft errors occur as single bit errors, the SECDED approach is implemented in our system.

The SECDED approach uses extra bits to encrypt the data, and these control bits are stored with the data in memory. When the data is written to the memory, control bits are also generated and recorded. When the data is read from the memory, the control bits generated are compared with the control bits recorded. If the control bits do not match, they are decoded and the wrong bit position in the data is detected.

```
Require requestList, Core_init,mtx, queueSem, startSem, finishSem
bindThread(Core_init);
if criticalData.isTrue then
    pthread_mutex_lock(&mtx);
    requestList.add(threadID);
    pthread_mutex_unlock(&mtx);
    sem_post(&queueSem); // Notify the RCDS thread
    sem_wait(&startSem[threadID]); // Wait response from the RCDS thread
    Core_protected = whichCore[threadID];
    bindThread(Core_protected);
    compute(criticalData);
    bindThread(Core_init);
    sem_post(&finishSem[threadID]); // Notify the RCDS thread
else
    compute(non-criticalData);
end if
```

Figure 4.2. Application thread.

```
Require requestList, mtx, queueSem, startSem, finishSem
Core_protected = sched_getcpu();
while requestList.isNotEmpty do
    sem_wait(&queueSem) ;
    pthread_mutex_lock(&mtx);
    threadID = requestList.first();
    pthread_mutex_unlock(&mtx);
    whichCore[threadID] = Core_protected;
    sem_post(&startSem[threadID]);
    sem_wait(&finishSem[threadID]);
    requestList.delete(threadID);
end while
```

Figure 4.3. RCDS on a high reliability core.

The wrong bit is inverted and the program continues to execution with the corrected data. The number of control bits may vary depending on the size of the data and the strength of the protection. Each control bit is responsible for several bit positions in the data whereas each bit position is controlled by at least two control bits. We utilize seven control bits for 32-bit data in our implementations. Six of them are responsible for several bit positions within the data, and the last one is the parity of both data and control bits. Figures 4.4 and 4.5 show the algorithms used in the encoding and decoding phases of the SECDED approach, respectively. This process is performed for each 32 bit-data in each row of the cache. Thus, the storage cost of the ECC implementation is 21.9% due to the use of additional control bits.

The protection of all cache structures with ECC introduces significant overheads in terms of performance, energy consumption and area [71]. ECC encoding is applied for each data written to the cache, ECC decoding is applied for every data read from the cache. In addition, the correction phase of the ECC should be applied if there is a bit flip in the data. The processing time of these phases has negative effect on the performance. The protection of all cache structures with SECDED approach is not a practical approach, especially for the performance-, energy-, and area-sensitive systems. Therefore, SECDED approach is implemented only for the cache structures of the high reliability cores in our work.

4.1.5. Experimental Study

4.1.5.1. Simulation Platform. To model our proposed architecture, we use gem5 system simulator [72], which is a sophisticated simulator merging different aspects of M5 and GEMS simulators. The simulator provides different types of ISAs, CPU models and coherence protocols to the instantiation of interconnection networks, devices and multiple systems. It gives support different types of ISAs such as ARM, ALPHA, MIPS, Power, SPARC, and x86. It supports four different CPU models including two simple single CPI models, an out-of-order model, and an in-order pipelined model. We can simulate different caches and interconnects due to the flexibility of memory systems.

procedure setECC (input)

/*Computes the six parity check bits for the "information" bits given in the 32-bit word input.
The check bits are c[5:0].
Bit Checks these bits of **input**
c[0] 0, 1, 3, 5, ..., 31 (0 and the odd positions).
c[1] 0, 2-3, 6-7, ..., 30-31 (0 and positions xxx1x).
c[2] 0, 4-7, 12-15, 20-23, 28-31 (0 and posns xx1xx).
c[3] 0, 8-15, 24-31 (0 and positions x1xxx).
c[4] 0, 16-31 (0 and positions 1xxxx).
c[5] 1-31 */

$\mathbf{c0} \Leftarrow \mathbf{input} \oplus (\mathbf{input} >> \mathbf{2})$;
$\mathbf{c0} \Leftarrow \mathbf{c0} \oplus (\mathbf{c0} >> \mathbf{4})$;
$\mathbf{c0} \Leftarrow \mathbf{c0} \oplus (\mathbf{c0} >> \mathbf{8})$;
$\mathbf{c0} \Leftarrow \mathbf{c0} \oplus (\mathbf{c0} >> \mathbf{16})$; // c0 is in posn 1.

$\mathbf{t1} \Leftarrow \mathbf{input} \oplus (\mathbf{input} >> \mathbf{1})$;
$\mathbf{c1} \Leftarrow \mathbf{t1} \oplus (\mathbf{t1} >> \mathbf{4})$;
$\mathbf{c1} \Leftarrow \mathbf{c1} \oplus (\mathbf{c1} >> \mathbf{8})$;
$\mathbf{c1} \Leftarrow \mathbf{c1} \oplus (\mathbf{c1} >> \mathbf{16})$; // c1 is in posn 2.

$\mathbf{t2} \Leftarrow \mathbf{t1} \oplus (\mathbf{t1} >> \mathbf{2})$;
$\mathbf{c2} \Leftarrow \mathbf{t2} \oplus (\mathbf{t2} >> \mathbf{8})$;
$\mathbf{c2} \Leftarrow \mathbf{c2} \oplus (\mathbf{c2} >> \mathbf{16})$; // c2 is in posn 4.

$\mathbf{t3} \Leftarrow \mathbf{t2} \oplus (\mathbf{t2} >> \mathbf{4})$;
$\mathbf{c3} \Leftarrow \mathbf{t3} \oplus (\mathbf{t3} >> \mathbf{16})$; // c3 is in posn 8.

$\mathbf{c4} \Leftarrow \mathbf{t3} \oplus (\mathbf{t3} >> \mathbf{8})$; // c4 is in posn 16.

$\mathbf{c5} \Leftarrow \mathbf{c4} \oplus (\mathbf{c4} >> \mathbf{16})$; // c5 is in posn 0.
$\mathbf{c} \Leftarrow ((\mathbf{p0} >> \mathbf{1}) \wedge \mathbf{1}) \vee ((\mathbf{p1} >> \mathbf{1}) \wedge \mathbf{2}) \vee ((\mathbf{p2} >> \mathbf{2}) \wedge \mathbf{4}) \vee ((\mathbf{p3} >> \mathbf{5}) \wedge \mathbf{8}) \vee$
$((\mathbf{p4} >> \mathbf{12}) \wedge \mathbf{16}) \vee ((\mathbf{p5} \wedge \mathbf{1}) << \mathbf{5})$;

$\mathbf{ca} \Leftarrow getParity(\mathbf{c} \oplus \mathbf{u})$;
$\mathbf{ca} \Leftarrow \mathbf{ca} << \mathbf{6}$;

$\mathbf{c} \Leftarrow \mathbf{c} \vee \mathbf{ca}$;
$\mathbf{c} \Leftarrow \mathbf{c} \oplus (-(\mathbf{input} \wedge \mathbf{1}) \wedge \mathbf{0x3F})$; // Now account for input[0].

Figure 4.4. Coding phase of the SECDED [70].

```
procedure decodeECC (cr, input)
// This function looks at the received seven check bits and 32 information bits (cr and input),
// and it returns with 0, 1, or 2, meaning that no errors, one error, or two errors occurred. It
// corrects the information word received (input) if there was one error in it.

cr0 ⇐ getParity(cr ⊕ input) ; // Compute overall parity of the received data.
c ⇐ setECC(input) ; // Calculate check bits for the received info.

syn ⇐ c ⊕ (cr ∧ 0x3F) ;
syn ⇐ syn ∧ 0x3F ;

if  cr0 == 0 then
   if  syn == 0 then
      return 0 ; // If no errors, return 0.
   else
      return 2 ; // Two errors, return 2.
   end if
end if

if  ((syn − 1) ∧ syn) == 0 then
   // If syn has zero or one bits set, then the error is in the check bits or the overall parity bit
   // (no correction required).
   return 1 ;
end if

// One error, and syn bits [5:0] tell where it is in input.
b ⇐ syn − 31 − (syn >> 5) ;
// Correct input.
input ⇐ input ⊕ (1 << b) ;
```

Figure 4.5. Decoding phase of the SECDED [70].

It is primarily written in C++ and python and all the components are based on object-oriented design.

We run the gem5 simulator in ALPHA full system mode with timing cpu model. Our target architecture is a 4-core system, with individual L1 instruction and data caches for each core and a shared L2 cache. The main parameters of the simulated multicore system are given in Table 4.1. The gem5 simulator produces a configuration file which shows the visualization of the simulated architecture. The details of the simulated architecture can be found in Figure 4.6. To implement ECC protection in different levels of cache structures, modifications are made to the classical memory model of the gem5. The most difficult part of this section is not to implement ECC protection, but to track each data packet between the CPU and cache structures to perform ECC encoding/decoding phases. Otherwise, a consistent protection could not be enabled for cache structures. We define a variable, called protectionType, in the cache configuration file of the simulator. According to the value of this variable ECC protection is activated for each data packet in the individual cache structures.

Table 4.1. Gem5 simulator parameters.

Processor	
Number of cores	4
Processor type	ALPHA
Processor frequency	2 GHz
Simulation mode	Full System
Cache and Memory Hierarchy	
L1 instruction cache	32 KB, 2-way, 64 byte blocks, 2 cycle latency
L1 data cache	8 KB, 2-way, 64 byte blocks, 2 cycle latency
L2 cache	2 MB unified, 8-way, 64 byte blocks, 20 cycle latency
Memory	512 MB, 30 ns latency

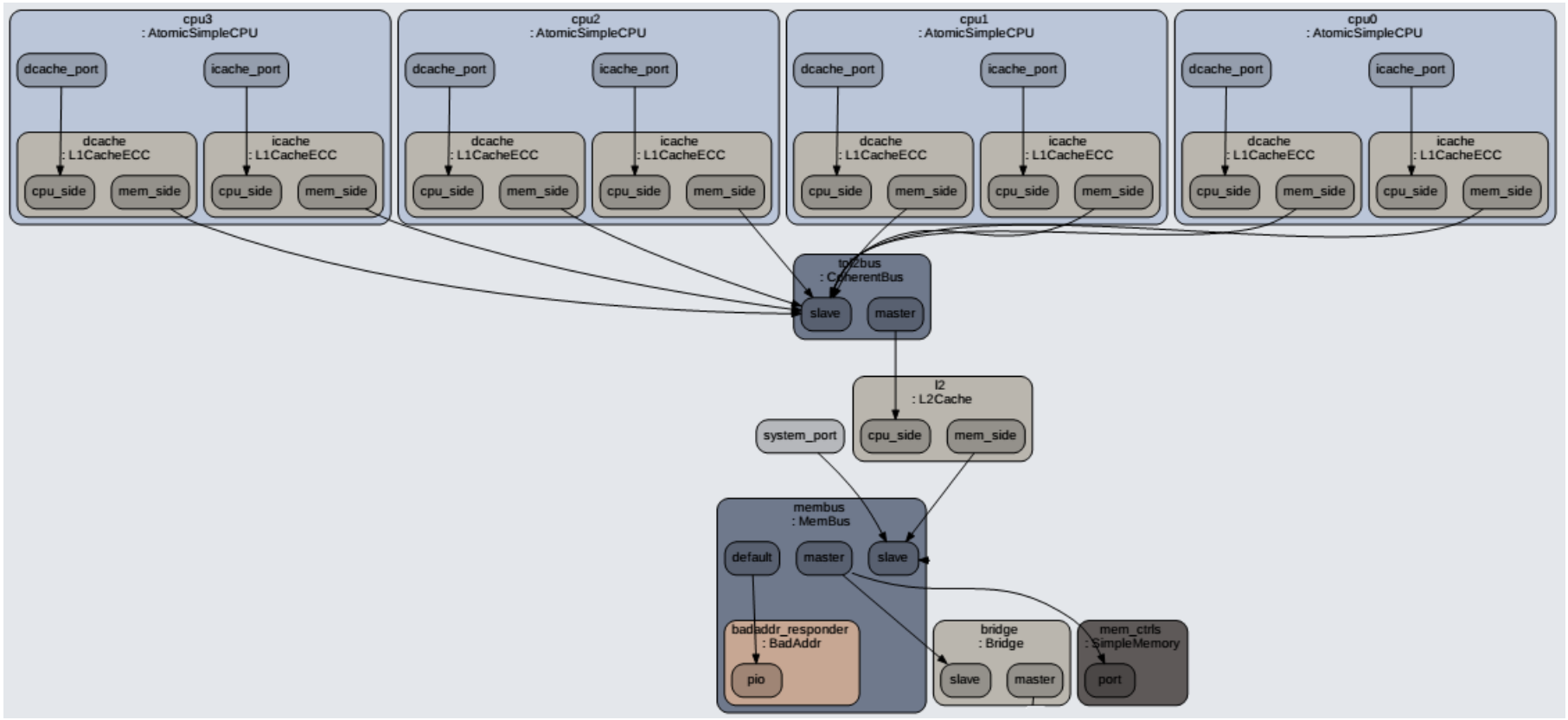


Figure 4.6. The visualization of the simulated system [72].

In addition to the gem5, we use the CACTI [73] which takes the cache parameters and models delay/power/area of cache components. It is used for calculating energy consumption in the cache structures. Energy consumption is calculated with the energy values of per read and per write accesses taken from the CACTI. The details of the energy consumption calculation will be explained in Section 4.1.5.4.

4.1.5.2. Applications. Applications selected to validate our system should contain code fragments that use critical and non-critical data proportionally. If the majority of code fragments use critical data, all of the application threads run on the high reliability core which reduces the utilization of the low reliability cores. On the other hand, if a few code fragments use a relatively small amount of critical data, few threads need to run on the high reliability core, which decreases the utilization of that core. Thus, the applications are selected by considering the utilization of all cores in our experimental study.

Our first application, LU Decomposition, is selected from SPLASH-2 [74] which is a benchmark suite for parallel computational applications. This application decomposes a dense matrix into a lower triangular matrix and a higher triangular matrix. In this decomposition process, the matrix is divided into blocks in order to utilize the temporal locality for the individual sub-matrix elements [75]. To reduce communication between the threads, the blocks are assigned to the threads using 2-D scatter decomposition. The pseudocode of Blocked Dense LU Factorization, expressed in blocks, is shown in Figure 4.7. Firstly, the diagonal block is decomposed in the second line, then all perimeter blocks are updated using the diagonal block in the third line. In the sixth line, the matrix multiplication of two blocks is performed by the thread having the target block.

In this application, the diagonal block is assumed to be more critical when compared to the other blocks because the diagonal block is used for updating the perimeter and inner blocks. A soft error occurrence on this data might easily spread to other regions of the matrix by affecting the result of the computation.

For this reason, the operations on the diagonal block are performed in a more protected manner using a high reliability core.

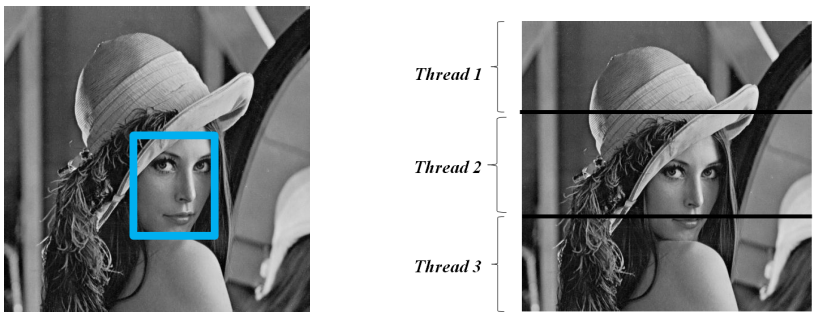
```
for k=0 to N-1 do
    Factor diagonal block A_kk
    Compute values for all perimeter blocks in column k and row k using A_kk
    for  j=k+1 to N-1 do
        for i=k+1 to N-1 do
            A_ij = A_ij - A_ik * A_kj
            /* Update interior blocks using corresponding perimeter blocks */
        end for
    end for
end for
```

Figure 4.7. Blocked Dense LU Factorization [75].

Our second application is the Susan application from ParMiBench [68], consisting of a set of parallel applications for embedded systems. It is an image recognition application that can detect the edge and corner points in an MRI of the brain. It takes a black-white image as input and performs regulations for brightness, smoothness and spatial control on the image [68]. Application threads share the input image by a row-wise block partitioning method and perform calculations on their own chunks.

For Susan, an image recognition application, the critical parts of the input image can be easily detected by the user. A black-white Lena image is used as input and it is assumed that the pixels on the face are more critical than the pixels not on the face. The critical parts of the input data and the sharing between the threads are given in Figure 4.8. In this example, sharing of critical data is not homogeneous for each thread. Thread1 do not have critical data, Thread2 has a large fraction of the critical data, and Thread3 has less critical data than Thread2.

In Susan application, each pixel needs neighboring pixels on the same line for calculation, so the threads cannot run the entire critical part at once on the high reliability core. On the other hand, migration of a corresponding thread for each pixel on the face significantly increases the migration cost and the waiting time of the threads at the request queue. Therefore, it is considered to migrate the corresponding thread for each row in the critical data (see rectangular region in Figure 4.8(a)).



(a) Critical data. (b) Data sharing among the threads.

Figure 4.8. Susan application with Lena image.

4.1.5.3. Fault Injection Model. A random fault injection model is performed using single bit flip model in the cache structures. In order to determine a fault point, five random numbers are selected: a clock cycle number (among the clock cycles of running application), a core number (out of 4 cores), a cache line (among the 128 cache lines), a cache index within a line (among 64 bytes, the size of the cache line), and a bit position (among 8 bits).

To enable fault injection experiments on the gem5 simulator, the event scheduling mechanism of the gem5 is used. Firstly, uniformly distributed random fault injection points are determined. After determining the fault injection point, the target bit of the target cache is corrupted by inverting the bit at a predefined clock cycle. For each experiment, a fault is definitely injected and the fault position and the incorrect output are recorded on the host machine. A schematic view of our fault injection model is shown in Figure 4.9.

There is a controller mechanism that monitors the status of the running program and evaluates the result of the fault injection experiments. There are three possible outcomes for a fault injection experiment: correct execution, Silent Data Corruption (SDR), and program error. There is no error in the output of the program in case of correct execution. In this case, the fault might hit to the data not used by the application, or it might not affect the application output although it is used by the application. In the SDR case, the program terminates with an incorrect output.

This means that the erroneous data is used by the application and the generated output is affected by this fault. In the last case, the program cannot continue to execute because of a fault such as a segmentation fault, or the program cannot be terminated normally due to an unexpected situation such as an infinite loop. In order to evaluate the result of a fault injection experiment, the controller checks the output of the application with the golden output that is generated by the correct execution (with the safe configuration). If there is no difference between the files, then the experiment is assumed as correct execution. Otherwise, the execution is assumed as Silent Data Corruption and the faulty data is logged in the host machine.

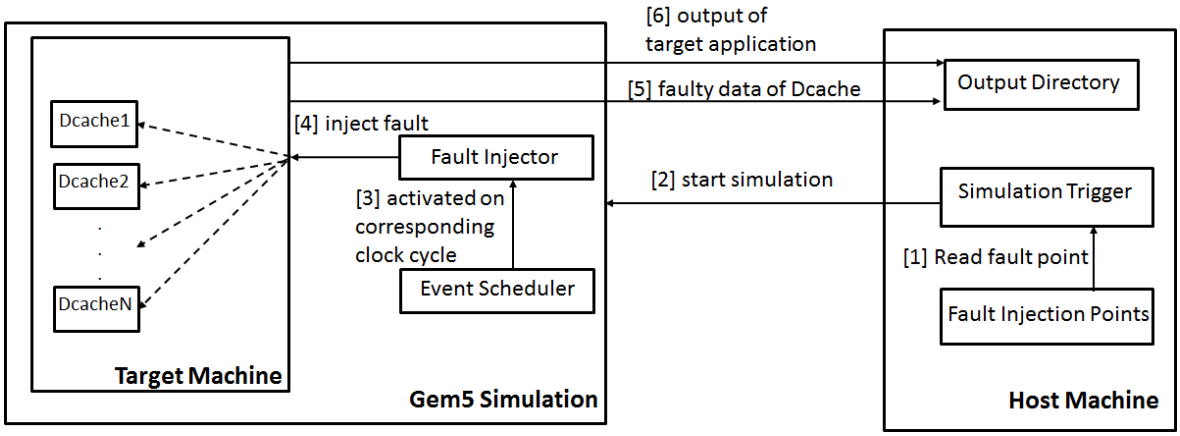


Figure 4.9. Fault injection model in our framework where numbers on the arrows represent order of operations for an experiment.

4.1.5.4. Performance and Energy Consumption. Protecting cache structures with the SECDED causes considerable performance, energy and area overheads [71]. However, the gem5 simulator does not consider the time spent in SECDED encoding and decoding phases. For this reason, the cache access latency values are increased for the caches that use SECDED approach. The cache access latency value is set as two cycles for an unprotected cache whereas this value is increased to three for a protected cache since there is an extra one cycle penalty due to SECDED approach [71].

Another overhead that should be considered when using ECC protection is energy consumption. In order to calculate energy consumption on cache structures, the energy per access values are estimated for protected and unprotected caches using CACTI [73].

Energy consumption is calculated based on two different methods, cache energy and system energy. In the first case, only the energy consumption of the cache structure is modeled by considering only dynamic energy consumption [76]. Cache energy is modeled as follows:

$$E = \sum_{i=0}^{\#ofCPUs-1} [Eread * Nread_i + Ewrite * Nwrite_i] \tag{4.1}$$

where $Nread_i$ and $Nwrite_i$ are the number of read and write accesses to the data cache of core i, respectively. $Eread$ and $Ewrite$ are the energy values of read and write accesses estimated by the CACTI, respectively. $Nwrite_i$ value consists of both write accesses and the number of cache line replacements due to the cache misses.

In the second method, not only the cache energy but also the energy consumption of the entire system is modeled by considering the processor, memory, and off-chip buses [71]. The similar energy consumption values are used for the processor, memory, off-chip bus and ECC encoder/decoder [71]. The system energy is modeled as follows:

$$\begin{aligned} E \quad = \quad & \sum_{i=1}^{n-1}(N_{unprotected_i} * E_{readECCoff}) \\ & + N_{protected} * (E_{readECCon} + E_{dec}) + (W_{protected} * E_{cod}) \\ & + \sum_{i=0}^{n-1} M_i * (E_{bus} + E_{mem}) + (M_{protected} * E_{cod}) \\ & + (R_{protected} * E_{dec}) + E_{proc} * \sum_{i=0}^{n-1} N_i \end{aligned} \tag{4.2}$$

where N is the number of cache access, W is the number of write access, M is the number of misses, R is the number of replacements, and E is the energy value. $N_{unprotected}$ and $N_{protected}$ are the number of cache accesses for the unprotected and protected caches, respectively. $W_{protected}$ is the number of write access, $M_{protected}$ is the number of misses and $R_{protected}$ is the number of replacements for the protected caches, respectively. $E_{readECCoff}$ and $E_{readECCon}$ are the energy values of the read accesses for the unprotected and protected caches, respectively. E_{dec} and E_{cod} are the ECC decoder and encoder energies, respectively. E_{bus} is the energy value for off-chip bus.

E_{mem} is the memory access energy and E_{proc} is the processor energy. The values used for the experiments are; $E_{dec} = 0.39\ nJ$ (per decoding), $E_{cod} = 0.2\ nJ$ (per coding), $E_{bus} = 10\ nJ$ (per access), $E_{mem} = 32\ nJ$ (per access) and E_{proc} =0.67 nJ [71]. We consider estimated energy values for $E_{readECCoff}$ and $E_{readECCon}$ taken from CACTI.

4.1.5.5. Experimental Results. We conduct a set of experiments to compare our partially safe configuration with the unsafe and safe configurations in terms of reliability, performance and energy consumption by using selected applications. Thread-to-core mapping is handled by our system in the partially safe configuration, while it is left to the Linux scheduler for the unsafe and safe configurations. The hardware characteristics of the proposed system is given in Table 4.1. The size of the L1 data cache is set smaller than the instruction cache to increase the probability of hitting the fault to the application's data. There are four protected cores in the safe configuration, four unprotected cores in the unsafe configuration, and a total of four cores in a partially safe configuration, one of which is a high reliability core. The applications run with a total of four threads where one of them is the main thread and the remaining ones are the worker threads. Therefore, we utilize one-to-one mapping for our partially safe configuration. A total of 800 fault injection experiments, 200 for each core, are performed. Thread migration and queue waiting overheads are implicitly included in the execution time.

For the LU application, a 512 x 512 dense matrix with 16 x 16 blocks is used as an input. The result of the LU application based on the number of instructions executed on each core, the number of read/write accesses, and the number of miss/replacement counts for each data cache are listed in Table 4.2 for the safe, unsafe and partially safe configurations. The total number of instructions is almost similar for each configuration. This indicates that the additional overheads of queue management and thread migration are not dominant for our partially safe configuration.

Table 4.2. Instruction counts and cache accesses of each data cache for the LU application.

Cache Configuration	**Core ID**	**# of Insts.**	**Nread**	**Nwrite**	**Nmiss**	**Nreplacements**
Safe	0	440,888,188	198,684,451	39,387,672	7,023,559	6,828,389
	1	5,701,266	1,223,544	663,705	112,147	106,057
	2	2,145,693,556	454,405,771	181,625,409	12,482,373	12,142,590
	3	475,839,465	214,654,934	42,163,220	7,577,934	7,200,661
	Total	3,068,122,475	868,968,700	263,840,006	27,196,013	26,277,697
Unsafe	0	6,937,039	1,536,886	843,600	235,003	233,332
	1	435,764,290	197,185,600	38,765,818	7,053,832	6,436,703
	2	2,140,400,031	453,665,106	181,212,126	12,673,549	11,997,851
	3	477,278,607	215,301,379	42,257,407	7,602,027	7,062,127
	Total	3,060,379,967	867,688,971	263,078,951	27,564,411	25,730,013
Partially Safe (Core0 - protected)	0	72,507,528	30,222,272	6,796,384	1,429,322	1,400,186
	1	431,567,408	193,562,815	38,512,843	6,727,238	6,607,317
	2	1,049,202,213	309,997,356	96,191,350	10,075,521	9,975,493
	3	1,529,104,708	338,178,883	123,966,324	11,670,652	11,620,602
	Total	3,082,381,857	871,961,326	265,466,901	29,902,733	29,603,598

Table 4.3. Average reliability, performance and energy consumption values for different cache configurations of 800 fault injection tests for the LU application.

	Unsafe Configuration	**Safe Configuration**	**Partially Safe Configuration**
Failure Rate	0.157	0	0.137
Execution Time (sec.)	3.138	3.492	3.166
Cache Energy (mJ)	45.799	51.426	46.223
System Energy (mJ)	1,784.508	2,233.001	1,901.266

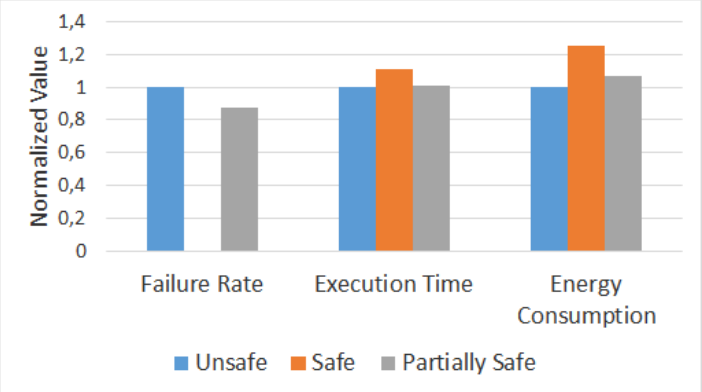


Figure 4.10. Normalized values of failure rate, execution time and energy consumption for the LU application.

Table 4.3 shows the results of execution time, energy consumption, and fault injection experiments for the safe, unsafe and partially safe configurations. Figure 4.10 shows the normalized failure rates, execution time, and energy consumption values (based on the system energy model) of the three cache configurations relative to the unsafe configuration. The partially safe configuration results in better performance and energy consumption, but worse in terms of reliability, compared to the safe configuration. However, it gives close performance and energy consumption results compared to the unsafe configuration with less failure rates.

The safe configuration consumes 25% more energy compared to the unsafe configuration, while our proposed partially safe configuration consumes 6% more energy than the unsafe configuration. The partially safe configuration provides roughly 13% better reliability with only 6% energy and 0.9% performance overheads relative to the unsafe configuration based on the normalized results. On the other hand, the safe configuration leads to better reliability behavior than the partially safe configuration, with 17% more energy consumption and 10% performance loss relative to the partially safe configuration.

Table 4.4. Average reliability, performance and energy values for different cache configurations of 800 fault injection tests for the Susan smoothing application.

	Unsafe Configuration	**Safe Configuration**	**Partially Safe Configuration**
Failure Rate	0.304	0	0.240
Execution Time (sec.)	0.391	0.395	0.369
Cache Energy (mJ)	9.479	10.639	9.640
System Energy (mJ)	161.573	235.265	164.750

For our second application, two different algorithms of the Susan application [68], Susan smoothing and Susan corners, are considered. In the first of these two different algorithms, image smoothing is performed while in the second algorithm corners are detected on an input image.
A 512 x 512 black-white Lena image is used for both applications.

Table 4.4 shows the results of execution time, energy consumption and failure rates for the safe, unsafe and partially safe cache configurations for the Susan smoothing application. Figure 4.11 shows the normalized results of failure rates, execution time and energy consumption of the three cache configurations for the Susan smoothing application, relative to the unsafe configuration. Our partially safe configuration provides 21% better reliability behavior with only 2% energy consumption overhead compared to the unsafe configuration based on the normalized results. On the other hand, there is no performance overhead compared to the unsafe configuration, although the partially safe configuration has more instructions and high cache access latency value for the high reliability core. Such a result may be due to the thread-to-core mapping strategy of the proposed configuration. We perform one-to-one mapping in the partially safe configuration, while the mapping is left to the Linux scheduler for the traditional safe and unsafe configurations.

The results of execution time, energy consumption and failure rates for the Susan corners application are shown in Table 4.5 and Figure 4.12.

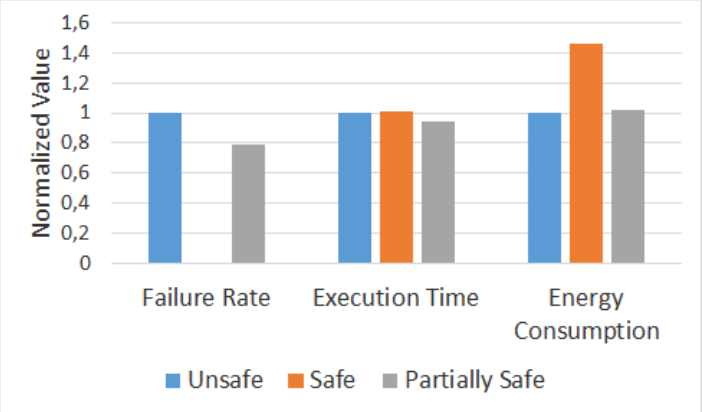


Figure 4.11. Normalized values of failure rate, execution time and energy consumption for the Susan smoothing application.

Table 4.5. Average reliability, performance and energy values for different cache configurations of 800 fault injection tests for the Susan corners application.

	Unsafe Configuration	**Safe Configuration**	**Partially Safe Configuration**
Failure Rate	0.049	0	0.034
Execution Time (sec.)	0.070	0.078	0.066
Cache Energy (mJ)	1.067	1.197	1.086
System Energy (mJ)	57.475	69.750	62.774

In the case of Susan corners, the partially safe configuration has better performance and lower energy consumption values compared to the safe configuration with higher fault rates based on the normalized results. The proposed partially safe configuration provides 30% better reliability behavior with an overhead of 9% energy consumption compared to the unsafe configuration without performance loss.

When the average results of these three applications are considered, partially safe configuration provides 21% better reliability behavior with an average of 6% energy consumption overhead compared to the unsafe configuration.

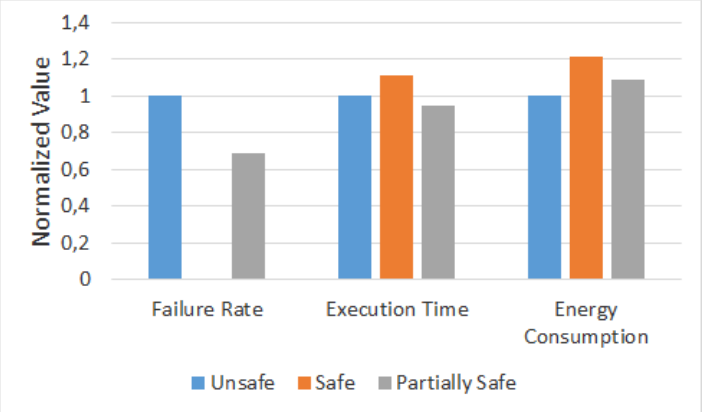


Figure 4.12. Normalized values of failure rate, execution time and energy consumption for the Susan corners application.

On the other hand, performance overhead is not dominant for the partially safe configuration when the average values are considered. According to these results, the proposed partially safe configuration is highly recommended for the applications where the reliability is an important concern under the performance and energy constraints and where the user can be able to define critical data for an input. It should be noted that, the performance of our approach can be fairly influenced by the number of critical data accesses. The partially safe configuration is appropriate for the applications where the number of critical and non-critical data accesses are balanced.

4.2. System with Two Types of Protected Cores

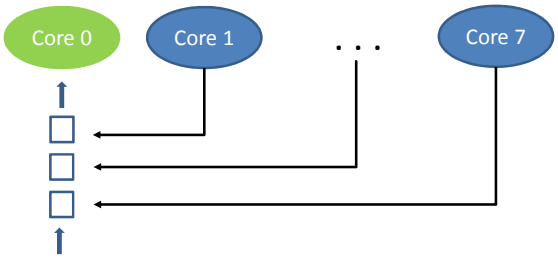
In this part of our study, we utilize a heterogeneous chip multiprocessor framework which consists of three types of cores: (i) a high reliability core, (ii) a middle-level reliability core, and (iii) a set of low reliability cores. While the ECC protection is provided for the L1 data cache of high reliability core, parity check is used for the L1 data cache of middle-level reliability core. Input data is classified as critical, semi-critical and non-critical where application threads are mapped to the different cores in terms of reliability based on their data usage. We utilize the similar applications with the previous section in our experimental study part.

The simulation-based fault injection framework is used to make an extensive experimental study on both proposed asymmetrically reliable caches and the traditional unsafe and safe caches.

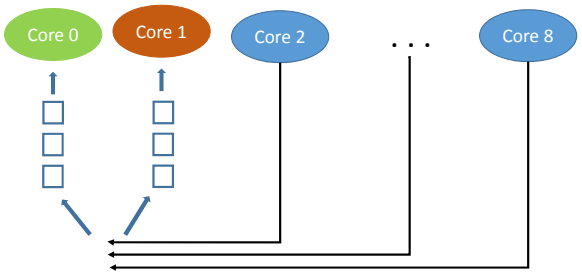
4.2.1. Architecture and Execution Model

Error Detection Codes (EDC) and Error Correction Codes (ECC) are two common methods to protect cache structures. While parity protection, a type of EDC, can detect single bit errors, SECDED can both correct single-bit errors and detect two bit errors. These two methods provide different degrees of reliability-level and also they differ in performance and power consumptions of the application. For ECC protection, seven control bits are used for each 32 bit data in the cache and the additional storage cost of using this protection is 21.9%. For the parity check, a single parity bit is stored for each 32 bit data in the cache and the additional storage cost of this method is 3.125%. The cost of parity check is significantly less than the ECC protection; however, the strength of the protection provided by the ECC is higher than the parity check. Therefore, the ECC protection is used for the high reliability cores and the parity check is used for the middle-level reliability cores.

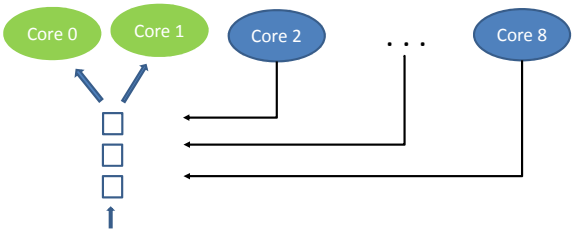
To test the effectiveness of our methodology, we consider two different architecture models in experimental study. In the first one, we consider an 8-core system with one protected core for the partially safe configuration (see Figure 4.13(a)). In the second model, we use a 9-core system with two different partially safe configurations. In the first configuration, we use a 9-core system including one high reliability core and one middle-level reliability core. In this case, there are two types of protected cores to be used for critical and semi-critical data accesses. Therefore, there are two separate queues for two different types of data accesses (see Figure 4.13(b)). The second configuration provides a 9-core system with two high reliability cores, which has only one queue to collect critical data accesses for both high reliability cores (see Figure 4.13(c)).



(a) Partially safe configuration with 1 high reliability core



(b) Partially safe configuration with 1 high and 1 middle-level reliability cores.



(c) Partially safe configuration with 2 high reliability cores.

Figure 4.13. Different types of partially safe configuration.

The execution model used in the system with two types of protected cores is similar to the system with single type of protected core, except that there are a separate queue for the ECC protected core and a separate queue for the parity check (see Figure 4.13(b)). The queue in front of the core that uses the parity check (Figure 4.13(b) - Core 1) consists of threads that send requests for the semi-critical data, while the queue in front of the ECC protected core (Figure 4.13(b) - Core 0) consists of threads that send requests for the critical data.

4.2.2. Applications

In this section, the similar applications presented in Section 4.1.5.2 are used. The classification of input data is divided into three types: critical, semi-critical and non-critical data. For the LU application, the diagonal blocks of the matrix are classified as critical data since they are used in the update step of the other perimeter and internal blocks in our first configuration of 8-core system. In our second configuration of 9-core system (with one high, one middle-level and several low reliability cores), we classify diagonal blocks as critical data and perimeter blocks as semi-critical data since the perimeter blocks are used for updating the internal blocks. In the configuration of 9-core system (with two high reliability cores) both diagonal and perimeter blocks are assumed as critical data. For all cases, the remaining parts of the matrix are left as non-critical data.

For the Susan application, the image pixels on the face are selected as critical data in our first configuration of 8-core system (Figure 4.14(a)). They are assumed more critical than the ones not on the face which corresponds to 6.4% of input data. In our second configuration of 9-core system (with one high, one middle-level and several low reliability cores), we reduce the size of critical data. Some parts of the face including eyes, nose, and mouth which corresponds to 3.6% of input data are selected as critical data (Figure 4.14(b)) and the remaining parts of the face which correspond to 2.8% of input data are selected as semi-critical. In the configuration of 9-core system with two high reliability cores, the image pixels on the face are again selected as critical data. For all cases, the pixels which are not part of the face are left as non-critical data.

4.2.3. Experimental Study

We conduct a set of experiments to compare reliability, energy and performance results of our partially safe configuration with the unsafe and safe configurations by using the set of applications given. The similar simulator environment (gem5) and the similar performance and energy consumption methods are used in the system with two types of protected cores.

(a) Critical data for Case 1. (b) Critical data for Case 2.

Figure 4.14. Critical data on the Lena image.

Thread-to-core mapping is left to Linux scheduler for the safe and unsafe configurations, but it is controlled by our system for partially safe configurations. While the similar fault injection model is used in this part, the number of fault injection tests is calculated differently. Leveugle *et al.* [77] propose a statistical model to determine sample size in fault injection experiments with different margin errors and confidence levels. The number of fault injection tests is equal to 1064 for each application in our framework with 95% confidence level and 3% error margin.

4.2.3.1. Experimental Results Using Single Type of Protected Cores. In the first set of experiments, there are 8 cores, where one of them is selected as high reliability core for the partially safe configuration. The LU and Susan applications are executed with eight threads. Table 4.6 presents the number of instructions committed by each core, the number of read/write accesses and the number of miss/replacement counts at each data cache for execution of the LU application on the safe, unsafe and partially safe cache configurations. Total number of committed instructions, read, write, miss and replacement counts are almost similar for each configuration type. It demonstrates that the impacts of migration and queue management overheads are not prevailing for the partially safe configuration.

Table 4.6. Instruction counts and cache accesses of each data cache for the LU application in an 8-core system.

Cache Configuration	**Core ID**	**# of Insts.**	**Nread**	**Nwrite**	**Nmiss**	**Nreplacements**
Safe	0	7,135,987	1,597,582	787,419	224,147	221,472
	1	86,236,431	16,818,411	7,972,973	3,437,525	3,237,080
	2	1,780,372,920	270,150,562	149,748,106	11,081,458	10,889,136
	3	73,677,023	14,306,509	6,805,841	2,937,301	2,691,191
	4	91,753,850	17,690,193	8,450,817	3,665,942	3,504,014
	5	63,824,223	12,471,423	5,924,179	2,557,943	2,320,178
	6	69,444,854	13,564,742	6,452,695	2,795,960	2,529,415
	7	79,624,290	15,531,771	7,369,731	3,137,745	2,958,814
	Total	2,252,069,578	362,131,193	193,511,761	29,838,021	28,351,300
Unsafe	0	6,626,228	1,417,569	733,336	194,494	192,264
	1	76,419,160	14,903,690	7,061,768	3,196,799	2,841,788
	2	1,774,775,414	270,406,529	149,665,763	11,647,478	11,277,490
	3	80,226,204	15,640,574	7,423,588	3,300,431	3,075,060
	4	74,788,110	14,327,206	6,802,979	3,055,794	2,622,561
	5	72,735,329	14,208,913	6,766,453	2,995,107	2,628,901
	6	69,146,913	13,494,665	6,414,730	2,959,574	2,495,431
	7	74,697,341	14,571,615	6,912,537	3,128,540	2,648,017
	Total	2,229,414,699	358,970,761	191,781,154	30,478,217	27,781,512
Partially Safe (Core0 - ECC protected)	0	39,174,044	7,692,981	3,850,275	1,391,878	1,324,989
	1	74,361,957	14,199,107	6,889,662	2,857,762	2,797,319
	2	80,138,130	15,240,721	7,540,872	3,094,469	3,030,371
	3	81,125,964	15,487,988	7,515,767	3,130,614	3,064,780
	4	84,020,844	16,035,825	7,769,701	3,242,751	3,183,447
	5	65,092,603	12,456,704	6,037,397	2,495,487	2,443,942
	6	68,269,967	13,061,189	6,341,244	2,607,961	2,548,636
	7	1,776,789,667	269,216,693	149,437,085	7,195,478	7,148,747
	Total	2,268,973,176	363,391,208	195,382,003	26,016,400	25,542,231

Figure 4.15 shows normalized values of the failure rate, execution time and energy consumption (based on both cache energy and system energy) for three cache configurations with respect to the unsafe cache configuration. The safe configuration has the largest execution time and energy consumption values with the best reliability behavior.

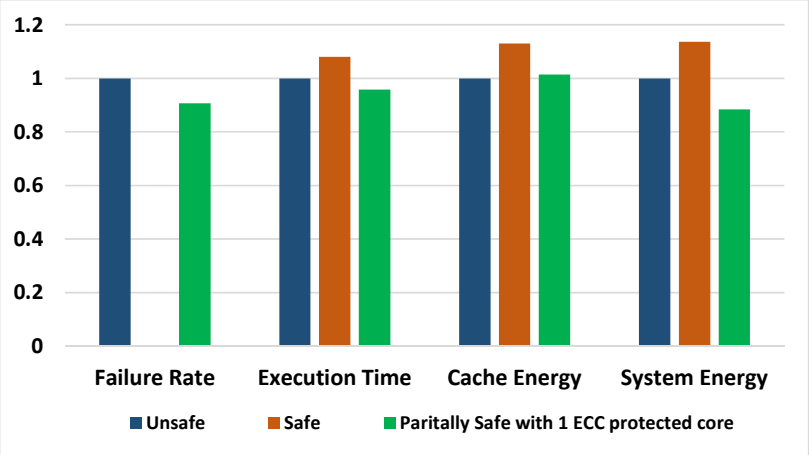


Figure 4.15. Normalized values of failure rate, execution time and energy consumptions for the LU application in an 8-core system.

On the other hand, the performance of partially safe configuration is better than both safe and unsafe configurations in spite of migration and queuing overheads since it maps threads to cores based on one-to-one mapping for unprotected cores. Moreover, the cache energy results of the the partially safe configuration is very close to the unsafe configuration while it is better than both safe and unsafe configurations in terms of system energy. Since the partially safe configuration has less miss counts compared to the safe and unsafe configurations, this has direct influence on the energy consumption of memory and off-chip bus systems. Our proposed partially safe configuration consumes 1% more cache and 12% less system energy than the unsafe configuration, whereas the safe configuration consumes 13% more cache energy and 13.7% more system energy than the unsafe configuration. In consequence, our partially safe configuration presents 10% less failure rate with 4% better performance and 12% less system energy compared to the unsafe configuration based on normalized results.

Figure 4.16 presents normalized values of failure rate, execution time and energy consumption for three cache configurations with respect to the unsafe cache configuration for the Susan smoothing application.

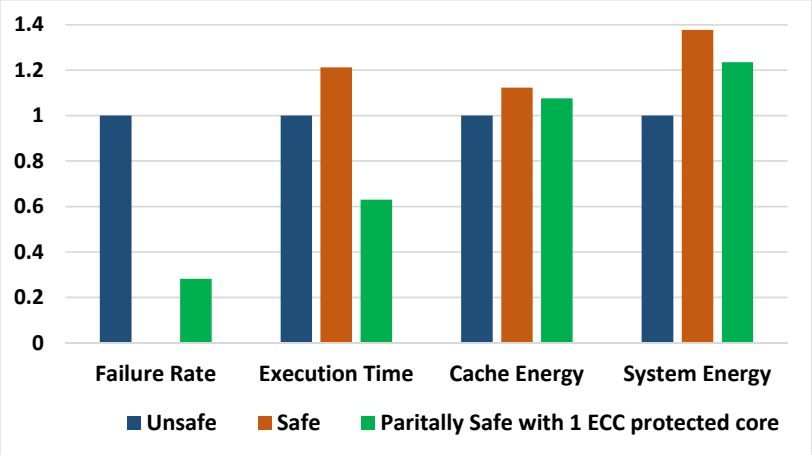


Figure 4.16. Normalized values of failure rate, execution time and energy consumption for the Susan smoothing application in an 8-core system.

The partially safe configuration has 72% less failure rate than the unsafe configuration with better performance as a result of thread-to-core mapping strategy. When we analyze the thread-to-core mapping of Linux scheduler for this application, it uses only two out of eight cores actively in the unsafe and safe configurations. Therefore, the partially safe configuration shows better performance since it uses all cores in the execution. Our partially safe configuration consumes 7.6% more cache energy and 23.5% more system energy than the unsafe configuration, whereas the safe configuration consumes 12.3% more cache energy and 37.8% more system energy compared to the unsafe configuration. Therefore, our proposed partially safe configuration presents 72% less failure rate and 47% better performance with 23.5% energy consumption overhead (based on system energy).

Figure 4.17 displays normalized values, where the partially safe configuration shows 11% better performance with 15% less failure rate and 9.9% more system energy consumption compared to the unsafe configuration for the Susan corners application. Although our partially safe configuration has lower energy per access values than the safe configuration, the cache energy consumption of it is close to the safe configuration. The high number of cache read/write accesses and miss/replacement counts of the partially safe configuration is the main reason behind this result.

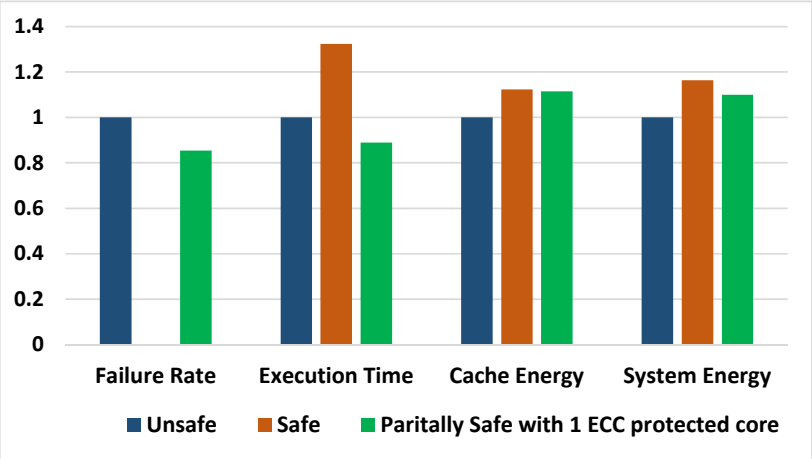


Figure 4.17. Normalized values of failure rate, execution time and energy consumption for the Susan corners application in an 8-core system.

On the other hand, the partially safe configuration consumes 9.9% more system energy and the safe configuration consumes 16.3% more system energy than the unsafe configuration. In consequence, significant reliability gain is observed with marginal increase in energy consumption and performance improvement using a single high reliability core in this set of experiments.

4.2.3.2. Experimental Results Using Different Types of Protected Cores. In this set of experiments, we use the following comparative configurations: (i) a 9-core system with two high reliability and seven low reliability cores, (ii) a 9-core system with one high reliability, one middle-level reliability and seven low reliability cores, (iii) a 9-core system consisting of nine unprotected cores, and (iv) a 9-core system with fully protected cores.

Figure 4.18 presents the normalized values of the failure rate, execution time, cache and system energy consumption in a 9-core system with different configurations relative to the unsafe configuration for the LU. The partially safe configuration with two high reliability cores has 3% less execution time and 4% less system energy consumption, while reducing the failure rate by 15% compared to the unsafe configuration.

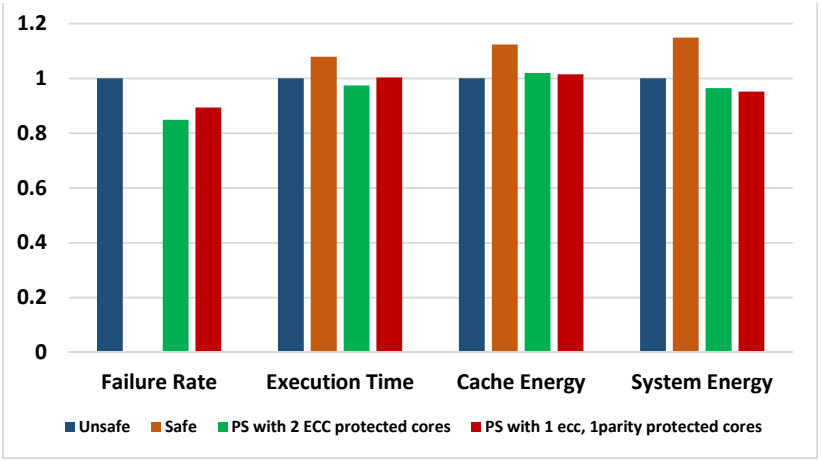


Figure 4.18. Normalized values of failure rate, execution time and energy consumptions for the LU application in a 9-core system.

On the other hand, the partially safe configuration with one high and one middle-level reliability cores reduces the failure rate by 10% with close performance and 5% less system energy consumption compared to the unsafe configuration. While the partially safe configuration with two high reliability cores has fewer failure rates than the partially safe configuration with one high and one middle-level reliability cores, both of them have fairly close performance and energy consumption results. One of the reasons is that there are two separate queue structures in the system with one high and one middle-level reliability cores, resulting in a higher performance overhead relative to the system with two high reliability cores. On the other hand, the safe configuration has 7.9% more execution time and 14.8% more system energy consumption than the unsafe configuration. As a result, both of the proposed partially safe configurations have better reliability results than the unsafe configuration, while they show better performance and energy consumption results than the safe configuration.

In Figure 4.19, the normalized values of failure rate, execution time, energy consumption of the four configurations are presented relative to the unsafe configuration for the Susan smoothing. The partially safe configuration with two high reliability cores has 38% lower execution time and 29% higher system energy consumption, while reducing the failure rate by 78% compared to the unsafe configuration.

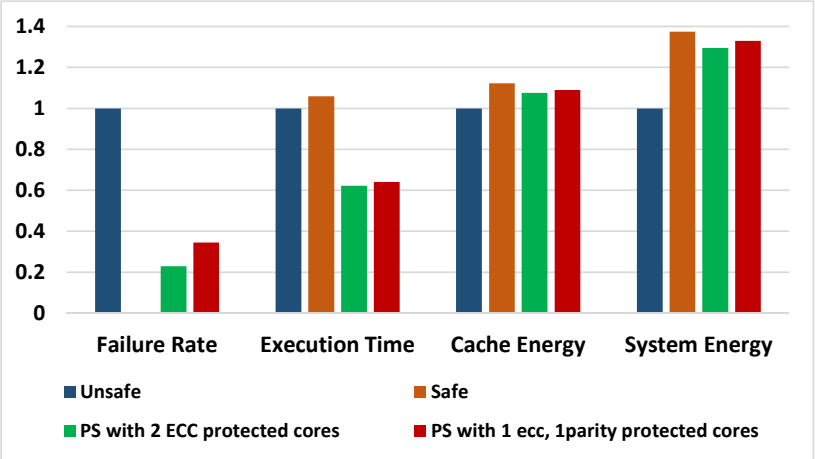


Figure 4.19. Normalized values of failure rate, execution time and energy consumptions for the Susan smoothing application in a 9-core system.

On the other hand, the partially safe configuration with one high and one middle-level reliability cores has 36.5% less execution time and 32.9% more system energy consumption, while reducing the failure rate by 65.5% compared to the unsafe configuration. When the failure rates of the both partially safe configuration are considered, the Susan smoothing application is a vulnerable application to the faults, and a significant reduction in the fault rates are observed by providing protection for only the critical data. On the other hand, the performance results of the both partially safe configurations are significantly better than the unsafe and safe configurations due to thread-to-core mapping.

As another observation, the partially safe configuration with one high and one middle-level reliability cores has more energy consumption (especially for the system energy) than the partially safe configuration with two high reliability cores. The L1 cache access counts of the four different configurations for the Susan smoothing application are shown in Table 4.7. While both of the partially safe configurations increase L1 cache access counts obviously, the configuration with one high and one middle-level reliability cores has the highest number of L1 cache accesses. This increase results from the additional codes to separate critical, semi-critical, and non-critical data on the input image and has a direct influence on the calculation of energy consumption.

Table 4.7. Total number of instruction counts and cache accesses for the Susan smoothing application in a 9-core system.

Cache Configuration	**# of Insts.**	**Nread**	**Nwrite**	**Nmiss**	**Nreplacements**
Safe	838,101,177	181,380,156	2,983,097	1,610,764	1,606,448
Unsafe	837,874,889	181,319,897	2,961,502	1,607,403	1,602,572
P.S. with 2 high reliability cores	898,786,343	193,441,798	11,833,609	2,661,297	2,242,799
P.S. with 1 high and 1 middle-level reliability cores	911,248,932	196,456,133	12,696,926	2,831,397	2,298,901

As a result, the partially safe configurations with one high and one middle-level reliability cores has slightly higher energy consumption results than the partially safe configuration with two high reliability cores.

Figure 4.20 shows the normalized results of the four configurations for the Susan corners application. According to these results, the partially safe configuration with two high reliability cores has a 16% reduction in failure rate compared to the unsafe configuration with 7% less execution time and 11% more system energy consumption. The partially safe configuration with one high and one middle-level reliability cores has 1% more execution time and 18.9% more system energy consumption while reducing the failure rate by 12.5% compared to the unsafe configuration. On the other hand, the safe configuration has 4.6% more execution time and 11% more energy consumption than the unsafe configuration. According to these results, the partially safe configuration with one high and one middle-level reliability cores consumes more energy than the safe configuration. When the L1 cache access counts of the Susan corners application shown in Table 4.8 are considered, this is an expected result. As in the Susan smoothing application, the partially safe configuration with one high and one middle-level reliability cores has the highest L1 cache access counts. However, in Susan corners application, a more obvious increase (12.4%) in the access counts hides the advantage of the this configuration in terms of energy consumption.

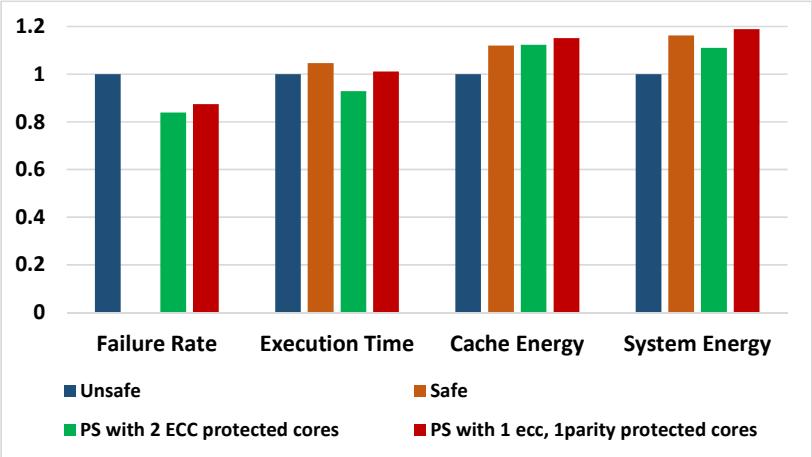


Figure 4.20. Normalized values of failure rate, execution time and energy consumptions for the Susan corners application in a 9-core system.

Apart from these metrics, the cost is another important metric shown in reliability studies. The additional memory cost of preserving L1 caches with ECC method is 21.9%, while the cost of parity checking is 3.125%. According to the cost metric, given in terms of the additional memory cost, both of the partially safe configurations have less cost than the safe configuration. The user can prefer one of the proposed partially safe configurations with a cost-sensitive approach regarding the reliability of the system.

According to the test results taken in this part, different applications might show different behavior on the results. A user can decide to use a configuration specifically for an application based on the reliability needs of the system under the performance and energy consumption constraints. Both of the proposed partially safe configurations can be utilized for the applications where the criticality of the data can be identified.

4.2.3.3. Analysis of Fault Injection Experiments. In this section, the results of fault injection experiments for a selected application, Susan smoothing, are analyzed in more detail. Figure 4.21 displays the distributions of three cases in fault injection experiments for four different configurations.

Table 4.8. Total number of instruction counts and cache accesses for the Susan corners application in a 9-core system.

Cache Configuration	**# of Insts.**	**Nread**	**Nwrite**	**Nmiss**	**Nreplacements**
Safe	87,370,933	22,864,512	13,294,260	1,886,250	1,887,905
Unsafe	87,330,324	22,858,048	13,284,244	1,882,738	1,884,171
P.S. with 2 high reliability cores	92,283,705	25,588,770	14,071,122	2,071,833	2,049,350
P.S. with 1 high and 1 middle-level reliability cores	99,786,303	26,255,604	14,973,985	2,248,456	2,207,518

In the first case of correct execution, the fault does not hit to the application data or the program does not produce an erroneous output even if it uses the faulty data. In SDC (Silent Data Corruption) case, the injected fault is not recognized by the system and the program may produce erroneous outputs. In the last case of program error, the program cannot terminate in a certain time due to a fault on the system, or the program cannot continue to the execution because of a fault such as a segmentation fault.

According to the results shown in Figure 4.21, the unsafe configuration has 81.5% correct execution rate, 6.6% SDC rate and 11.9% program error rate. On the other hand, no erroneous output is observed for the safe configuration since the ECC protection is applied to the caches of all cores in that configuration. While the partially safe configuration with two high reliability cores has 95.6% correct execution, 2.1% SDC and 2.1% program error rates, the partially safe configuration with one high and one middle-level reliability cores has 93.6% correct execution, 2.5% SDC and 3.9% program error rates. According to these results, both of the partially safe configurations increase the correct execution rate by reducing the SDC and program error rates relative to the unsafe configuration. Furthermore, the partially safe configurations can reduce the SDC rate, which is considered as the most dangerous fault type, by a factor of three compared to the unsafe configuration.

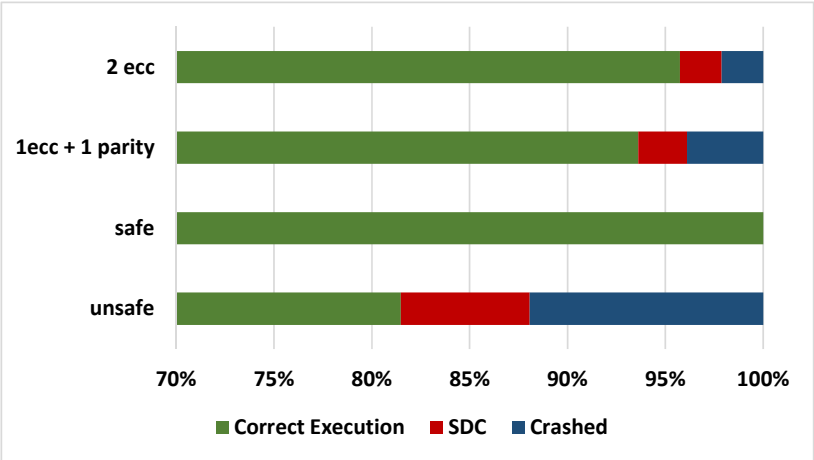


Figure 4.21. Results of fault injection experiments for the Susan smoothing application.

Figure 4.22 shows the behavior of Susan smoothing application when injecting faults to different cores for the unsafe configuration. Here, equal number of fault injection experiments are performed for each core and the fault injection points are selected based normal distribution. According to these results, injected faults to the $Core0$ and $Core5$ do not cause any SDC rate; however, they are resulted in roughly 10% of program error rates. So our understanding is that the system instructions are mostly executed on these cores. On the other hand, the highest SDC rates are observed in the $Core6$ and $Core3$ with 27.5% and 23.3% rates, respectively. It is clear that these cores are extensively used by the program and the injected faults corrupt the program output. The faults injected to the $Core1$ and $Core7$ are resulted in higher SDC rate than the $Core2$, $Core4$ and $Core8$. It is clear that the program uses $Core1$ and $Core7$ at medium density and $Core0$ and $Core5$ rarely. These rates may show different results for different runs of the unsafe configuration. Instead of such an unsafe configuration, we can utilize one of the proposed partially safe configurations by determining the criticality of input data and we can guarantee that the faults hit to the critical data will be eliminated with this configuration.

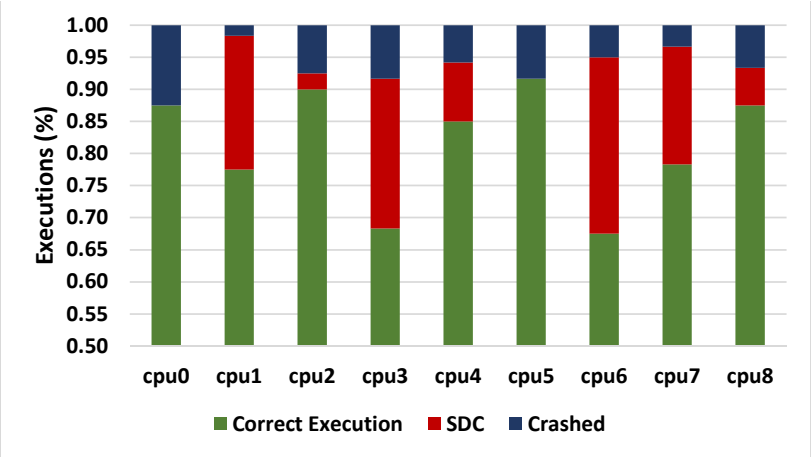


Figure 4.22. Application behavior when injecting faults to different cores for the Susan smoothing application in the unsafe configuration.

Figure 4.23 displays the correlation of the fault injection timing with the application behavior. The horizontal axis indicates the timing of the fault injection and the vertical axis indicates the percentage of the tests resulted in correct execution, SDC and program error. When the faults are injected toward the end of the execution, they do not significantly affect the program output. However, the injected faults in the middle of the execution increase the SDC rate. These results indicate that the Susan smoothing application is more susceptible to errors at the beginning of the execution, but the results converge to correct execution towards the end.

4.3. Summary

In the first subsection, the main mechanics of our proposed approach are implemented and evaluated with selected applications. Reliability-based critical sections are determined based on critical data usage. Experimental studies validates that the partially safe cache configuration significantly outperforms the unsafe configuration with respect to reliability (given in failure rate), and the safe configuration with respect to performance (given in execution time) and energy (given in both cache energy and overall system energy). Our system provides only as much reliability as needed for an application, thereby reducing potential overheads of the reliability enhancement.

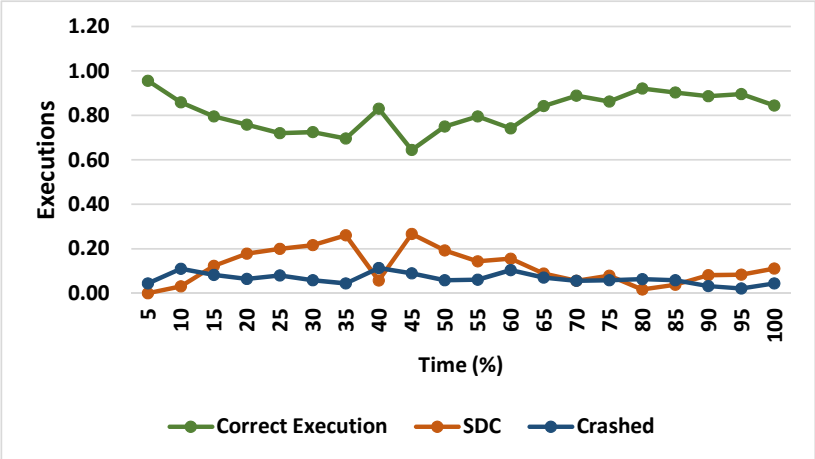


Figure 4.23. Correlation of fault injection timing with the Susan smoothing application execution.

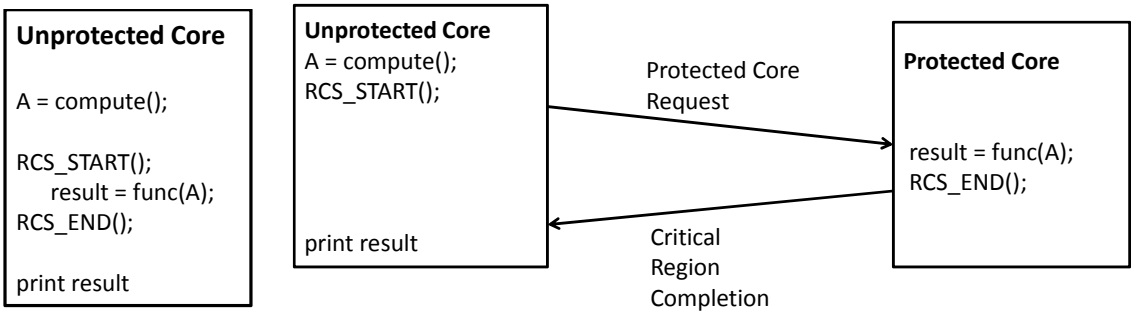
In the second subsection, our proposed system with asymmetrically caches supports two types of protected cores (high reliability core and medium-level reliability core). Application threads can use high, medium-level, or low reliability cores based on the criticality of the application's data (critical, semi-critical, and non-critical data). Our detailed experimental work shows that both of the partially safe configurations reduce the failure rate significantly compared to the unsafe configuration with much better performance and energy consumption results than the safe configuration.

5. ASYMMETRICALLY RELIABLE CACHES FOR CRITICAL REGIONS

In the previous chapter, we identified critical code sections based on the critical data usage. In this chapter, reliability-based critical sections are extracted depending on either user annotations or static analysis. The first part of the chapter presents how user can annotate the application program manually to determine the code fragments that need to be protected. In the second part, the critical code fragments are extracted by using static analysis based on function call graphs and execution time percentages according to the profiling results. After determining these regions, the applications are executed on a hardware platform that contains different number of protected and unprotected cores.

5.1. Code Criticality Based on Annotations

In this part of our study, a programmer can annotate the reliability-based critical code regions of the application and threads are mapped to the asymmetrically reliable cores based on the annotated regions. Figure 5.1 shows the code annotations and execution of the critical section. In the figure, funcA() is a critical function for the user and it is annotated with RCS_START and RCS_END keywords. When an application thread encounters with an RCS_START keyword, it should send a request for executing on the protected core. If it is idle, the scheduler migrates this thread to the protected core and the thread continues to its execution on the protected core. When an application thread encounters with an RCS_END keyword, it should send an acknowledgment message to the scheduler about the completion of its critical section, and migrate back to the initial unprotected core. The application should include the required files provided by our framework to call RCS_START and RCS_END functions.



(a) (b)

Figure 5.1. Source code and its execution.

5.1.1. Architecture Model

The hardware model used to test the validity of our approach is a 16-core architecture where the individual L1 instruction and data caches of the high reliability core(s) are protected. There are again three different cache protection methods in our system: unsafe, safe, and partially safe In the unsafe configuration, 16 unprotected cores are used, while the safe configuration uses 16 ECC-protected cores. In the partially safe configuration, there are eight cores with no protection and at least one core (one to eight cores) as the protected one(s). We assume that the cores are reconfigurable in terms of ECC protection. We can use them as either unsafe, safe or partially safe configuration according to our application needs. Our goal for a partially safe configuration is to provide performance and power values close to the unsafe configuration, and fault rates (SDC rates) close to the safe configuration.

5.1.2. Execution Model

Figure 5.2 and Figure 5.3 simply show the working principle of the application and the queue threads, respectively. When an application thread encounters a critical section of code in terms of reliability (funcA()), it has to add itself to the *requestList* and wait for a response from the queue thread running on a protected core. In our system, each protected core has one queue thread, which is responsible for getting the first request from the global queue.

```
Require requestList, unprotectedCoreID
bindThread(unprotectedCoreID);
 A = compute() ;
requestList.add(threadID);
protectedCoreID = waitResponseFromQueueThread();
bindThread(protectedCoreID);
 func(A); //Critical function
bindThread(initCoreID);
notifyQueueThread(threadID);
```

Figure 5.2. Application thread.

```
Require requestList
protectedCoreID = sched_getcpu(); //learn cpu id
while requestList.isNotEmpty do
   threadID = requestList.first();
   notifyThread(threadID, protectedCoreID);
   waitResponseFromThread(threadID);
   requestList.delete(threadID);
end while
```

Figure 5.3. Queue thread on a protected core.

In this study, we continue with the First Come First Served (FCFS) approach to map application threads to the protected cores. When a queue thread on a protected core receives a request from the list, it sends a response containing the protected core ID that the corresponding application thread will work with. The application thread that receives this response maps itself to the protected core and runs the corresponding critical code region. After the thread finishes this part, it returns back to the initial requesting core and notifies the corresponding queue thread. Then, the queue thread deletes the request from the global queue. If there are not any available protected cores, the queue threads cannot be able to receive new requests from the list and it will get stuck on the sixth line (waitResponseFromThread(threadID)) of Figure 5.3.

5.1.3. Experimental Study

5.1.3.1. Experimental Setup. The gem5 simulator [72] mentioned in previous chapters is used to implement the proposed system. The parameters of the simulated system are listed in Table 5.1. In addition to gem5, McPat [78] which models power, area and timing of desired architecture is used to estimate power consumption in the whole system. In McPat, the timing and area models derived from CACTI [73], and the dynamic power model is similar to Wattch [79]. The gem5 produces statistics and configuration files of the simulated architecture. These files are integrated and served as inputs to McPat to estimate power consumption. A separate script is used to parse the gem5 output and generate XML file for the McPat. Finally, the total power (including dynamic and leakage) and area values of the simulated architecture are estimated. The execution flow of the simulation is presented in Figure 5.4. In order to differentiate power values of ECC-protected and unprotected cores for different cache configurations, we update the McPat source code. It models each architectural component as a separate CACTI block and a parameter is defined which identifies whether a cache structure has an ECC protection or not.

Table 5.1. Gem5 simulator parameters.

Processor	
Number of cores	16
Processor type	ALPHA
Processor frequency	2 GHz
Simulation mode	Full System
Cache and Memory Hierarchy	
L1 instruction cache	32 KB, 2-way, 64 byte blocks, 2 cycle latency
L1 data cache	8 KB, 2-way, 64 byte blocks, 2 cycle latency
L2 cache	2 MB unified, 8-way, 64 byte blocks, 20 cycle latency
Memory	512 MB, 30 ns latency

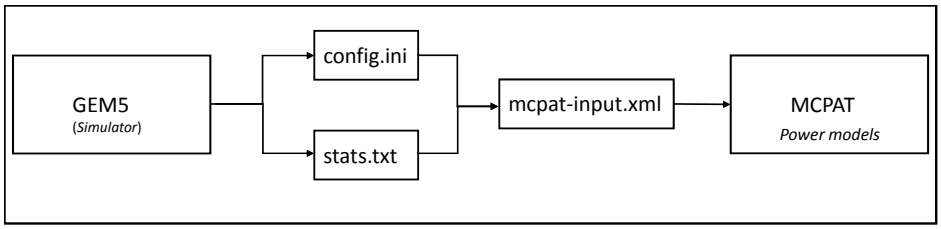


Figure 5.4. Execution flow of the simulation.

5.1.3.2. Applications. Applications are selected from the PARSEC benchmark suite [80] and profiled with the gprof, the GNU profiler. According to the profiling results, the functions that cover 90% of the execution time are specified as the reliability-based critical code regions for each application. Then, each of these functions (three or four functions) are executed on a system with asymmetrically reliable cores. For our experimental study, the criticality is defined in terms of application functions; however, it can be a portion of a function depending on the annotations. It does not matter for our system which parts of the code are annotated by the programmer. Any function or code fragment can be specified as critical code regions.

There are several constraints and restrictions in order to select applications. First of all, applications should contain multiple functions and the execution time percentages should be balanced (i.e. the most of the execution time should not be passed inside a single function). If the execution time of a critical function is too long, it occupies the high reliability core for a long time which diminishes the usage of low reliability cores. On the other hand, if the execution time of a critical function is too small, it uses the high reliability core(s) for a small time which decreases the utilization of the high reliability core(s). In this manner, applications in our experimental study are chosen by considering the utilization of all cores.

Secondly, the application should produce a valid output to interpret the results of fault injection tests and to compare the output with the golden one. If an application produces only a single output, such as an integer value, it may reduce the probability of having erroneous output.

Table 5.2. Applications that could not be selected.

Application Name	**Elimation Reason**
Blackscholes	100% of execution time passes in one function!
LU	100% of execution time passes in one function!
Radix	100% of execution time passes in one function!
Streamcluster	100% of execution time passes in one function!
fmm	Output may change!
Barnes	No output produced!
Water	No output produced!
Canneal	No output produced!

Additionally, applications that generate different results based on approximations or their stochastic nature may not be considered in our framework. In this case, we cannot observe SDC in the output. Based on the restrictions stated, it is quite tedious work to find appropriate applications for our framework. These constraints eliminate several applications from known benchmarks showed in Table 5.2.

In consequence, experiments are performed on the Bodytrack and Fluidanimate applications from the PARSEC benchmark suite. Bodytrack is a computer vision application that monitors the human body with multiple cameras [80]. Fluidanimate is a computer animation program that simulates real-time fluid motion [80]. Table 5.3 shows the profiling results of two applications. The first column in the table shows the names of the applications, and the second column shows the functions of these applications. The third column contains the percentage of execution time that indicates how much the protected core will be busy with this function. The fourth column shows the number of calls for each function. These numbers have direct influence on the number of protected core requests and the queue overhead. The fifth column shows that whether these functions run in parallel or in serial. The values in this column indicate how many threads will compete for the protected cores.

Table 5.3. Application functions, percentage of execution time, and total number of calls for each function.

Application Name	**Function Name**	**Ex. Time Percentage**	**# of Calls**	**Serial or Parallel**	**% of faults hit to this function**
Bodytrack	Exec	50%	96	Parallel	18.80%
	OutputBMP	21.43%	1	Serial	5.46%
	InsideError	7.14%	13045	Parallel	1.03%
	EdgeError	7.14%	13011	Parallel	1.11%
	ImageProjections	7.14%	13040	Parallel	0.59%
	LoadSet	7.14%	1	Serial	1.19%
Fluidanimate	ComputeForcesMT	45%	24	Parallel	14.74%
	ComputeDensitiesMT	40%	24	Parallel	12.46%

The percentage of faults that hit to the functions as a result of fault injection experiments performed in the unsafe configuration are shown in the last column. The fault percentages shown in the last column will be completely removed from the system when they are executed on the protected cores.

5.1.3.3. Experimental Results. Several experiments are conducted to compare reliability, performance and power consumption results of our partially safe configuration with the unsafe and safe configurations. The Bodytrack and Fluidanimate applications are executed with eight threads using the simdev input set of the PARSEC benchmark suite. The number of fault injection tests is set to 2400 with 95% confidence level and 2% error margin according to the study proposed by Leveugle *et al.* [77]. The SDC rate is used primarily to compare the results of fault injection experiments. Since SDC cases are not recognized by the application, such faults are the most critical type of faults for the applications. On the other hand, the user can be aware of the program error case since the program terminates abnormally.

The performance and power consumption results of the Bodytrack application are given in Figure 5.5. While running this set of experiments, different numbers of protected cores are used for each function in the application. Any "x-8" configuration given in the figures are composed of x-protected and 8-unprotected cores.

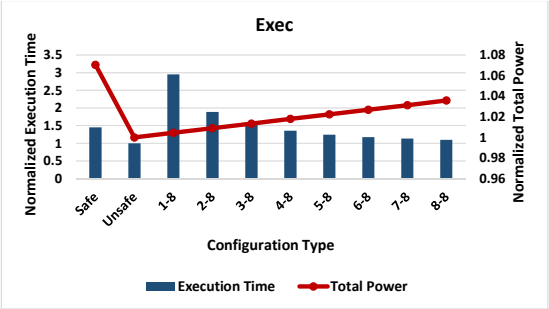
Figure 5.6 compares the results of fault injection experiments for the unsafe and partially safe configurations. Only a single function is preserved in each case of the partially safe configuration. The unsafe configuration has the lowest execution time and power consumption results, while it has the highest fault rates (in terms of both SDC and program error). For this reason, other columns in the figure show the normalized values relative to the unsafe configuration. The number of correct executions becomes so high for the case of 2400 fault injection experiments. Therefore, the bars related with the correct execution do not increase dramatically for each case in Figure 5.6. For example, the numbers of correct execution, program error and SDC cases are 2120, 26, and 254, respectively for the unsafe configuration. On the other hand, these values change as 2153, 17 and 230 for protecting the LoadSet function. While the change of 33 number of correct execution yields 1.55% increase in correct execution rate, the change of nine faulty execution yields 35% change in SDC rate. In this study, our goal is to decrease SDC rates as much as possible.

Figure 5.5(a) shows the results of the Exec function, which is determined as the critical code region, on different numbers of protected cores. Exec is a thread function that cannot be executed serially by a single thread at a time. When running the Bodytrack application in 1-8 configuration, all threads running the critical regions (i.e. Exec function) use only one protected core. The resource contention among the threads leads to the worst performance result for this configuration. When we add a new protected core in the system with every configuration, threads can run critical code fragments in parallel, and the waiting times of the threads decrease in the queue; hence, performance improvement is observed. Conversely, there is an increase in power consumption as more ECC-protected cores consume more power. Providing five protected cores for the Exec function seems appropriate in terms of power-performance trade-off.

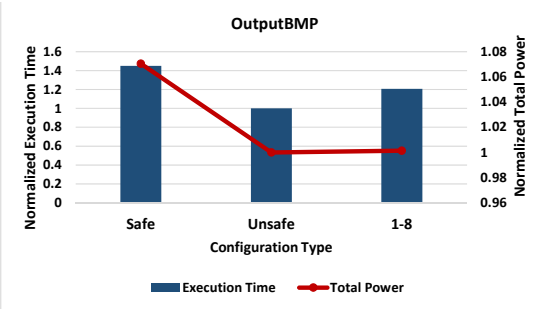
In this configuration (i.e. 5-8 configuration), execution time and power consumption values fall between the values of safe and unsafe configurations. When we perform fault injection experiments preserving the Exec function, a significant reduction in the SDC rate is observed relative to the unsafe configuration. The reliability of the system can be improved by 42% (in terms of SDC rate) with 24% performance loss and 2.2% more power consumption relative to the unsafe configuration. On the other hand, the safe configuration reduces the SDC rate to zero by correcting all single bit errors, with 45% performance loss and 7% more power consumption over the unsafe configuration.

Figures 5.5(b) and 5.5(f) display the results of execution time and power consumption by preserving OutputBMP and LoadSet functions, respectively. These functions are executed serially by the main thread. Therefore, only one protected core is sufficient for the partially safe configuration. The OutputBMP function increases reliability (in terms of SDC) by 46% with 20% performance loss and 0.14% additional power consumption compared to the unsafe configuration. Similarly, the LoadSet function improves the SDC rate by 34% with 8% performance loss and 0.44% additional power consumption compared to the unsafe configuration.

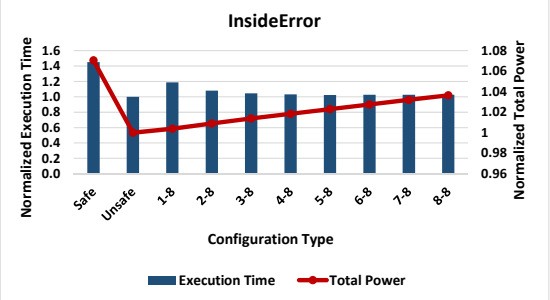
On the other hand, InsideError, EdgeError and ImageProjections are thread-parallel functions that have very similar behavior in terms of reliability, performance and power consumption results. Using only one protected core results in a significant additional queue overhead (see Figures 5.5(c), 5.5(d) and 5.5(e)). This effect decreases when new protected cores are added to the system. Considering power-performance trade-off, two protected cores for the InsideError and EdgeError functions and three protected cores for the ImageProjections function seem to be sufficient. By protecting the InsideError, EdgeError and ImageProjections functions, it is possible to provide SDC improvement of 65%, 57%, and 61% with 8%, 6%, and 10% performance loss and 0.9%, 0.9%, and 1.5% additional power consumption relative to the unsafe configuration, respectively.



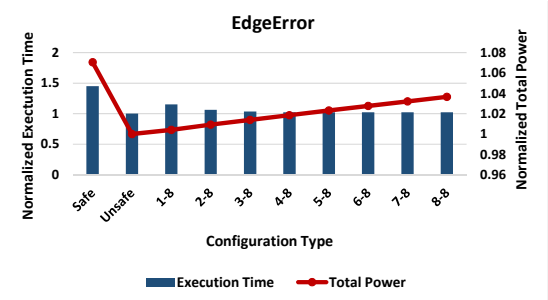
(a) Execution time and power consumption results with protecting Exec function.



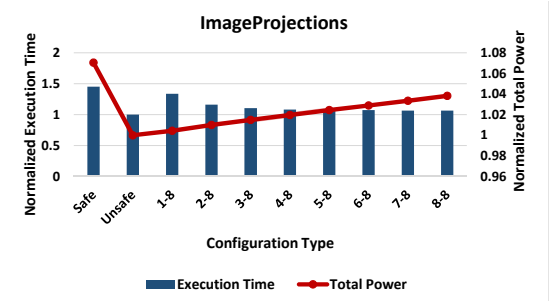
(b) Execution time and power consumption results with protecting OutputBMP function.



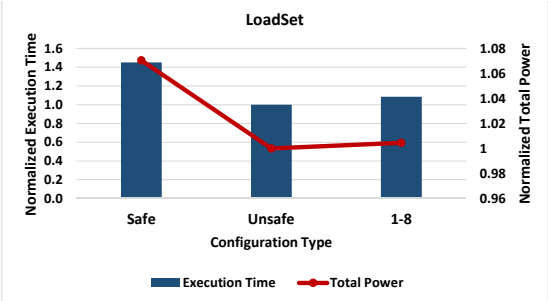
(c) Execution time and power consumption results with protecting InsideError function.



(d) Execution time and power consumption results with protecting EdgeError function.



(e) Execution time and power consumption results with protecting ImageProjections function.



(f) Execution time and power consumption results with protecting LoadSet function.

Figure 5.5. Normalized values of execution time and power consumption for the Bodytrack application. (x-8) configuration: x protected and 8 unprotected cores.

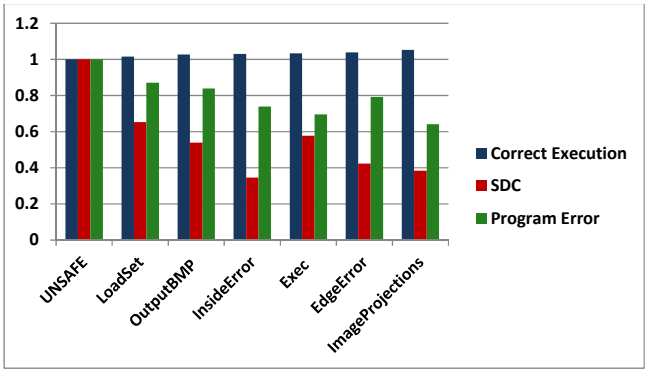
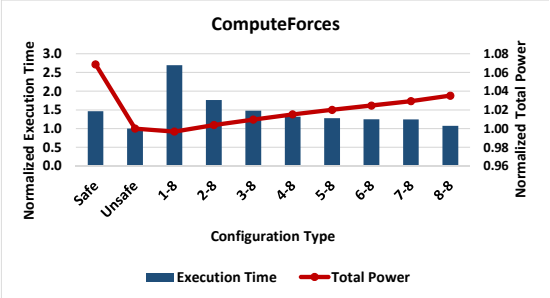


Figure 5.6. Fault injection experiment results of the Bodytrack application.

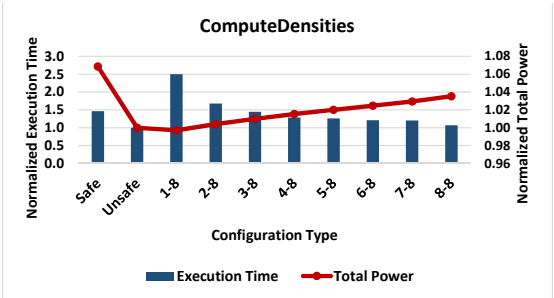
The results of performance, power consumption and the fault injection experiments of Fluidanimate application are shown in Figure 5.7. ComputeForces and ComputeDensities functions are similar in terms of working principle; therefore, they show similar behavior in the results. Both of them are thread-parallel functions, and they have additional queuing overhead. The best performance among the partially safe configurations belongs to the 8-8 configuration in which the numbers of protected and unprotected cores are equal. In other words, the presence of a protected core for each thread significantly reduces the waiting time of the threads in the queue.

Fault injection experiments are carried out in a partially safe configuration with four protected cores. In this configuration (i.e. 4-8 configuration), the execution time result falls between the values of the unsafe and safe configurations, and there is not as much power consumption as the 8-8 configuration. Compared to the unsafe configuration, 75% improvement in SDC rate is achieved with 32% performance loss and 1.5% additional power consumption when we protect the ComputeForces function. Similarly, 83% improvement in SDC rate is observed with 26% performance loss and 1.5% additional power consumption in the ComputeDensities function when compared to the unsafe configuration.

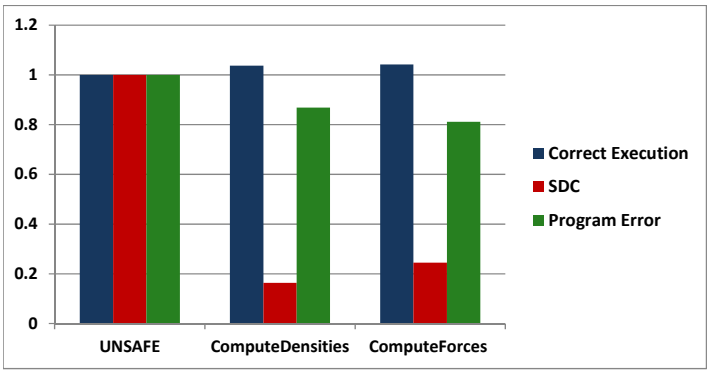
In order to measure the queuing overhead, an additional experiment is performed using the ComputeForces function of the Fluidanimate application used in our previous experiments.



(a) Execution time and power consumption results with protecting ComputeForces function.



(b) Execution time and power consumption results with protecting ComputeDensities function.



(c) Fault injection experiment results of unsafe, protecting ComputeDensities and ComputeForces functions.

Figure 5.7. Normalized values of execution time, power and reliability results for the Fluidanimate application.

In this experiment, four protected cores are used in the partially safe configuration. However, instead of waiting in the queue when protected cores are busy, the threads can run the critical code fragments on the unprotected cores. The normalized values of SDC rate, execution time and power consumption for the new configuration compared to the previous configuration are shown in Figure 5.8. The performance is improved by 21% in the new configuration since there is no waiting overhead in the queue. The SDC rate is worsened by 75% in the new configuration, as only the half of the threads are executing on the protected cores. Since equal numbers of protected cores are used in both configurations, the resulting power consumption is very close to each other.

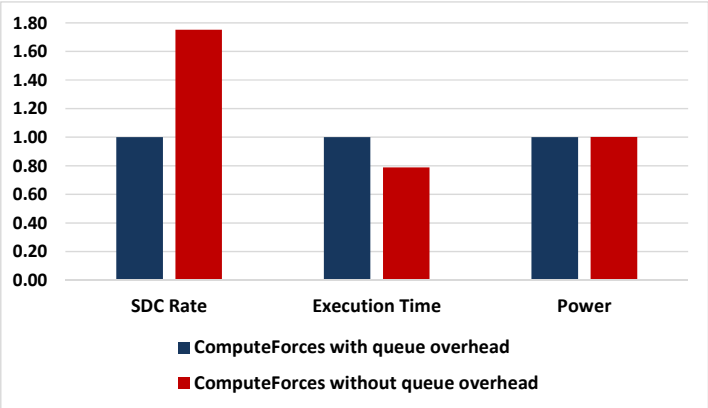


Figure 5.8. Effect of queue waiting with ComputeForces function

The user will be able to choose the appropriate version from these configurations based on their priorities in terms of reliability, performance and power consumption.

In this set of experiments, we use different numbers of protected cores for each function of the applications. According to the experiment results, it is confirmed that the reliability of the applications can be improved when only the critical code fragments are protected compared to the unsafe configuration. Similarly, the performance and power consumption values of the partially safe configuration are better than the safe configuration. A different number of protected cores can be used for each function in the partially safe configuration. While the level of reliability of the system does not change with each added protected core in the partially safe configurations, this leads to changes in performance and power consumption. As another observation, there is not an ideal number of protected cores for each application, but an ideal number of protected cores for each function. In this sense, in a system with a limited number of protected cores, the user can determine the number of protected cores used by considering the reliability-based critical regions.

5.2. Code Criticality Based on Static Analysis

In previous sections, reliability-based critical code sections are determined either by using the critical data or by providing the user annotations.

In this section, we determine the critical code sections of an application automatically without user annotations. One way to determine these parts for an application is to perform fault injection experiments and determine which parts of the program is affected by errors, depending on the analysis of the program output. However, performing fault injection experiments might be inefficient since it requires a lot tests which take long execution time.

Apart from this, the application code can be examined statically (compile time) to determine the critical code regions. In this work, each application is profiled using the GNU profiler, gprof, and the function execution time percentages and the call graph of the functions are generated. Then, high-priority functions to be protected are determined with vulnerability and criticality analysis on the profiling results.

5.2.1. Determining Function Priorities

Duque *et al.* [37] propose a method to determine task priority based on task criticality and task vulnerability metrics statically. A task vulnerability metric is defined as the effect of a possible error on the task output. A task criticality metric is defined as the effect of a possible error on the execution time of the application. The task vulnerability and task criticality metrics are calculated based on the task graph information of the application in advance.

In this part of the work, we adopt the task-based definitions proposed in the literature [37] as function-based definitions. The proposed task criticality and task vulnerability metrics are used to determine the high-priority functions for an application. Each application is profiled with the gprof and the flat profile containing the execution times of the functions and the call graph file showing the execution order of the functions are generated. By examining these files, function vulnerability and function criticality metrics are calculated and the function priorities are determined accordingly.

The function vulnerability is defined similar to the task vulnerability, and the functions with longer execution times assumed as more vulnerable. The vulnerability of each function in an application is determined as follows:

$$vulnerability_i = \frac{ET_i}{Max_{j \in Functions}(ET_j)}, \forall i \in Functions \tag{5.1}$$

where the ET_i is the execution time of function i, and $Max(ET_j)$ is the execution time of a function having the longest execution time. While the vulnerability values of the functions vary between [0,1], the vulnerability value of the function with the longest execution time is set to one.

The function criticality is defined as the effect of a possible error on the application performance because of a resource unavailability. This metric is defined based on two basic factors. In the first one, the slack times of functions are calculated and the functions with small slack times have more effect in application performance. In the second one, the number of functions invoked by the function is counted since they will increase the delay effect.

Slack is defined as the measure of flexibility for a function call to be delayed without lengthening application execution time. The slack time of a function is computed as the gap between the Earliest Start Time (EST) and Latest Start Time (LST) of the function by examining the function call graph. At that point, this value is normalized to the maximum slack value and the fixity value is obtained in the range of [0,1]. The fixity metric can be calculated by examining the call graph file as follows:

$$\begin{gathered} fixity_i = 1 - \frac{slack_i}{Max_{j \in Functions}(slack_j)}, \\ \forall i \in Functions \text{ and } Max_{j \in Functions}(slack_j) \neq 0 \end{gathered} \tag{5.2}$$

where $fixity_i$ shows the fixity value of $function_i$ and $slack_i$ presents the slack value of $function_i$ [37]. $Max(slack_j)$ figures out the largest slack value among all the functions in the call graph.

A high value of the $fixity_i$ indicates that the delay in the function i has a large effect on the application performance. A fixity value of one belongs to the functions on the critical path. However, the value of fixity may only change for the functions running in parallel. Since the slack time of a function running in serial will be zero, the fixity value will always be one based on Equation 5.2.

On the other hand, if there is a delay in a function, the functions invoked by it will also be delayed. In this manner, a function with more dependents (i.e. the function invokes a high number of other functions) are assumed to be more critical than the others. The function criticality is calculated with the function's own fixity value as well as the fixity values of the functions invoked as follows:

$$criticality_i = \left(fixity_i + \frac{\sum_{k \in children_i} fixity}{Max_{j \in Functions}(OutDegree_j)} \right) /2 \tag{5.3}$$

where $OutDegree_j$ is the number of invoked functions by $function_i$, and $Max(OutDegree_j)$ is the largest number of out-degree (i.e. the maximum number of invoked functions) in the function call graph. In this calculation, the first part shows the effect of the delay of the function itself and the second part shows the effect of the invoked functions. The function criticality is a value ranging between [0,1] by using a normalization term of two. A low value of function criticality indicates that this function has not a dominant effect on the application performance. On the other hand, the criticality value of a serial function, which has a fixity value of one, changes proportionally with the total number of functions called by this function.

In order to determine the high-priority functions for an application, the function vulnerability and function criticality metrics are represented in a single metric in the function priority as follows:

$$priority_i = \alpha * vulnerability_i + (1 - \alpha) * criticality_i \tag{5.4}$$

Here, α is the relative importance of the vulnerability metric on the criticality metric. Since both metrics are equally important in our system, the value of α is taken as 0.5 in our calculations. As a result, the high-priority functions are assumed to be reliability-based critical sections that must be protected in our system.

All of the metrics presented in this subsection are adopted from the work proposed by Duque *et al.* [37]. On the other hand, we assume that the tasks are functions in our case and we use real applications rather than ready task graphs.

5.2.2. Experimental Study

In this section, the architectural, execution and cache protection models used in Section 5.1 are utilized in a similar way. Since the fault injection experiments take too long, the percentage of L1 cache accesses are provided in the protected cores for the reliability assessment. Although taking fault injection experiments will give more accurate results, the percentage of protected L1 cache accesses will also give a point of view for the provided reliability. The following subsections presents the applications that are used for our experiments and the results of performance, power consumption and reliability of the configurations with different numbers of protected cores.

5.2.2.1. Applications. To show the performance, power consumption and reliability results of our system, a total of six applications, five from the SPLASH-2 benchmark suite [74] and one from the ParMiBench benchmark suite [68] are used. Since most of the execution time passes inside a single function in the remaining applications, these are not included the in our set. The main characteristics of the selected applications are summarized as follows.

- Cholesky: An application decomposes a matrix into the product of a lower triangular matrix and its transpose. Since the program works on sparse matrices, the cost of communication is much more than the calculation. The tk29.O input is used for this application.

- Raytace: An application renders a three-dimensional scene onto a two-dimensional image plane by using ray tracing method. In this application, data access patterns are irregular and not structured. For this application teapot.env input is used.
- Barnes: An application simulates the interaction of a system of bodies by utilizing the Barnes-Hut N-body method. Thread communication patterns highly depend on particle distribution. The default input is used for this application.
- Water-nsquared: An application that computes the forces and potentials occurring in a system of water molecules. The application consists of cell grids trying to reach each other's data, and the threads communicate intensively. The default input is used for this application.
- fmm: An application solves the N-body problem with Fast Multipole Method. In this application, the communication patterns are not structured. The largest input for this application (input.16384) is used.
- Dijkstra: An application computes the single-source shortest path in a graph represented by an adjacency matrix. The inputsmall.dat is used as an input for this application.

We implement a program in python to identify the high-priority functions for the applications by taking the flat profile and call graph files generated by the GNU profiler. This program computes the function vulnerability, fixity, criticality and priority metrics and returns the most prior functions in the application by examining these metrics. The profiling results of the selected applications are shown in Table 5.4. In this table, the first column shows the name of the applications, the second column shows the function name according to the priority order, the third column shows the execution time percentages of the functions alone, the fourth column contains the cumulative execution time percentages including their descendants and the fifth column shows the number of calls for these functions. While the fifth column shows the number of high reliability core visits, the fourth column shows that how long the high reliability core will be busy with this function.

Table 5.4. Profiling results of six applications.

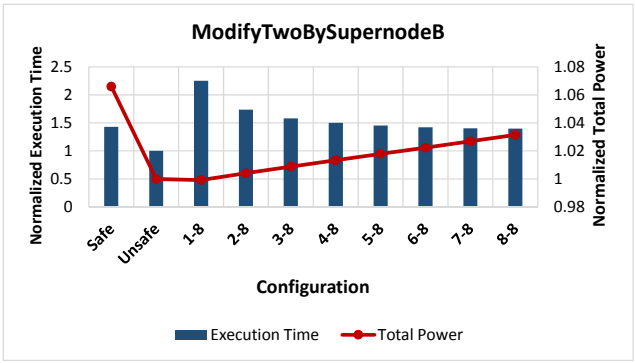
Application Name	**Function Name**	**Ex. Time Percentage (alone)**	**Ex. Time Percentage (cumulative)**	**# of Calls**
Cholesky	1. ModifyTwoBySuperNodeB	61.93%	61.93%	167,354
	2. FactorLLDomain	0.09%	68.00%	28
	3. CompleteSupernodeB	0.47%	18.70%	1,187
Raytrace	1. TriPeIntersect	23.20%	23.20%	736,266
	2. IntersectHuniformPrimList	6.19%	25.20%	182,607
	3. HuniformShadowIntersect	2.32%	17.40%	9,061
	4. TraverseHierarchyUniform	0.99%	44.60%	26,100
Dijkstra	1. startWorking	3.92%	4.40%	1
	2. dijkstra	4.40%	4.40%	1
Barnes	1. gravsub	40.81%	40.81%	15,459,166
	2. walksub	25.49%	95.50%	65,536
	3. subdivp	29.17%	29.17%	16,737,571
	4. stepsystem	0.35%	99.20%	4
	5. hackgrav	0.39%	95.90%	65,536
Water - nsquared	1. CSHIFT	55.10%	55.10%	1,962,240
	2. INTERF	26.06%	73.80%	4
	3. MDMAIN	0.95%	91.30%	1
fmm	1. VListInteraction	61.19%	61.19%	122,174
	2. ParallelExecute	0.01%	96.00%	1
	3. ListIterate	0.20%	82.50%	15,561

In this table, some functions are omitted from the experiment set because the total number of calls for these functions directly affects the total number of migrations for the high reliability core. In this context, as the TriPeIntersect function of the Raytrace application is called 736,266 times, there will be so many thread migrations among the cores and this amount of migration cannot be realized due to the limitations of the simulator environment. Similarly, since the gravsub and subdivp functions of the Barnes application are called 15,459,166 and 16,737,571 times, respectively, these functions are also removed from the experiment set.

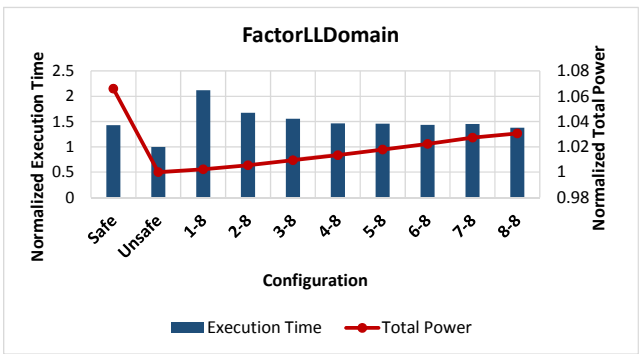
On the other hand, since the RCS_START and RCS_END keywords indicating where the critical region starts and ends are added to include the loop that calls the CSHIFT function, the cost of migration is not dominant for this function, even though it is called 1,962,240 times in the Water-nsquared application. The data dependencies among the functions are not examined and are assumed to be independent of each other.

5.2.2.2. Experimental Results. A set of experiments is performed to compare the performance, power consumption and reliability results of the partially safe configuration with the safe and unsafe configurations. The applications are executed with eight threads in each case. There are 16 unprotected cores for the unsafe configuration and there are 16 protected cores for the safe configuration in our experiments. As in the previous section, each experiment is performed in a system with eight unprotected and at least one (one to eight) protected cores for the partially safe configurations. To make a fair comparison for power consumption, eight unprotected cores and a different number of protected cores are used actively in a 16-core system for the partially safe configurations.

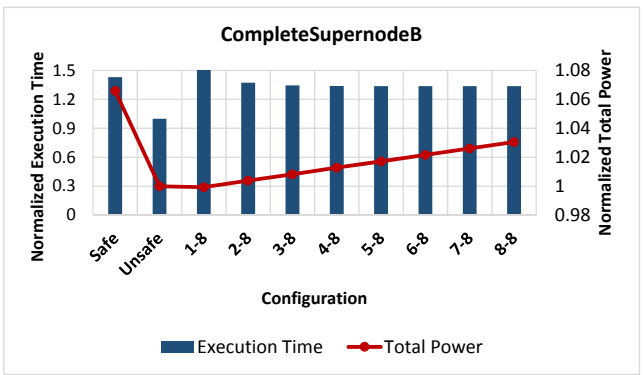
Figure 5.9 shows the performance and power consumption results of the Cholesky application. By examining all functions of the Cholesky application, a considerable performance overhead is observed in the 1-8 configuration where only one protected core is provided. As more protected cores are added to the system, performance improvements are observed since the protected cores share the load in the queue; however, more protected cores consume more power. To protect the ModifyTwoBySuperNodeB function, at least five protected cores should be used when performance and power constraints are considered. For FactorLLDomain function, at least four protected cores must be added to the system. On the other hand, to protect the CompleteSupernodeB function, use of only two protected cores seems to be sufficient by considering the power constraint (since the performance values converge after two protected cores).



(a) Execution time and power consumption results with protecting ModifyTwoBySuperNodeB function.



(b) Execution time and power consumption results with protecting FactorLLDomain function.



(c) Execution time and power consumption results with protecting CompleteSupernodeB function.

Figure 5.9. Normalized values of execution time and power consumption for the Cholesky application. (x-8) configuration: x protected and 8 unprotected cores.

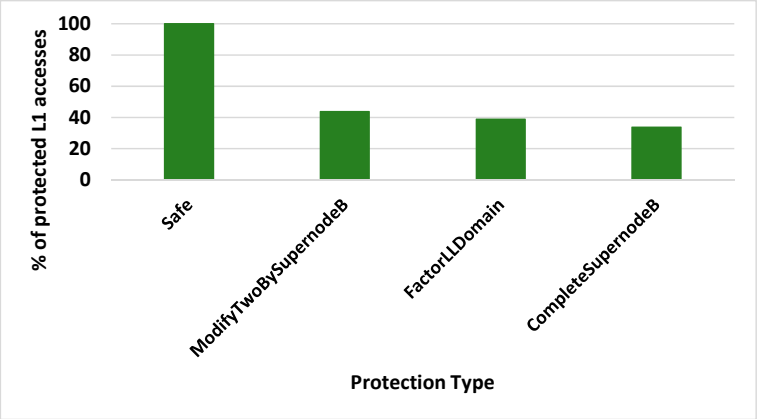
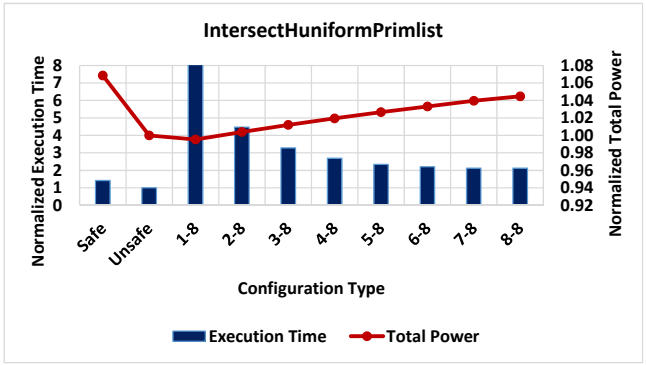


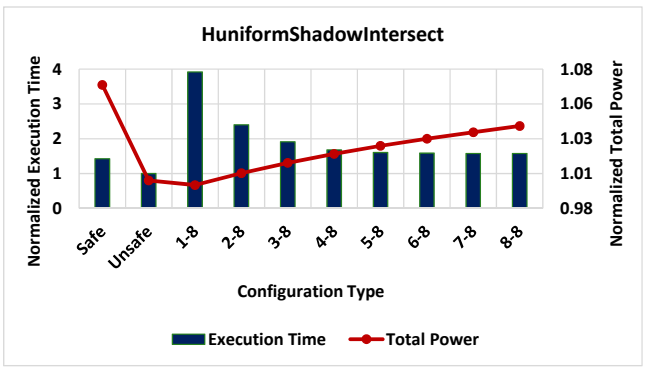
Figure 5.10. Percentage of protected L1 accesses for the Cholesky application.

In Figure 5.10, the percentage of protected L1 cache accesses is shown when the functions specified for the Cholesky application are individually protected by comparing with the safe configuration. For this experiment, different partially safe configurations (such as 1-8, 2-8, ..., 8-8) are not utilized. Since the same level of protection is provided for every partially safe configuration, this does not change the reliability results substantially. When the ModifyTwoBySuperNodeB, FactorLLDomain, and CompleteSupernodeB functions are specified as the critical code fragments, 44%, 39%, and 34% of L1 cache accesses can be preserved, respectively. These percentages can vary depending on the resource usage of the function whether it is a CPU-intensive or memory-intensive function. For this reason, it is observed that there is no direct correlation between the function duration and the L1 cache access counts while executing that function. According to these test results, to protect the ModifyTwoBySuperNodeB, FactorLLDomain, and CompleteSupernodeB functions separately, using four, five and two protected cores seems satisfactory considering the power and performance constraints in the Cholesky application, respectively.

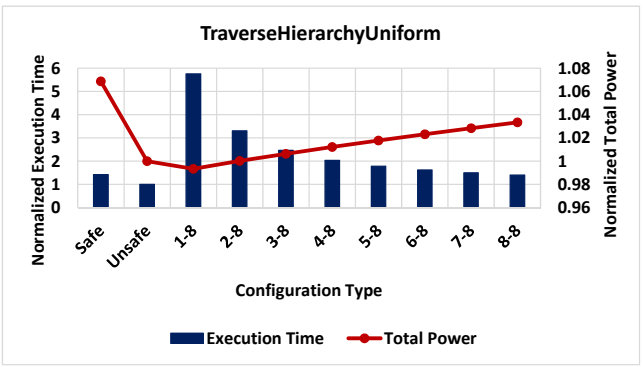
Figures 5.11 and 5.12 show the performance, power consumption and reliability results of the Raytrace application. When the IntersectHuniformPrimlist function is preserved, any of the partially safe configurations could not show better performance results than the safe configuration.



(a) Execution time and power consumption results with protecting IntersectHuniformPrimlist function.



(b) Execution time and power consumption results with protecting HuniformShadowIntersect function.



(c) Execution time and power consumption results with protecting TraverseHierarchyUniform function.

Figure 5.11. Normalized values of execution time and power consumption for the Raytrace application.

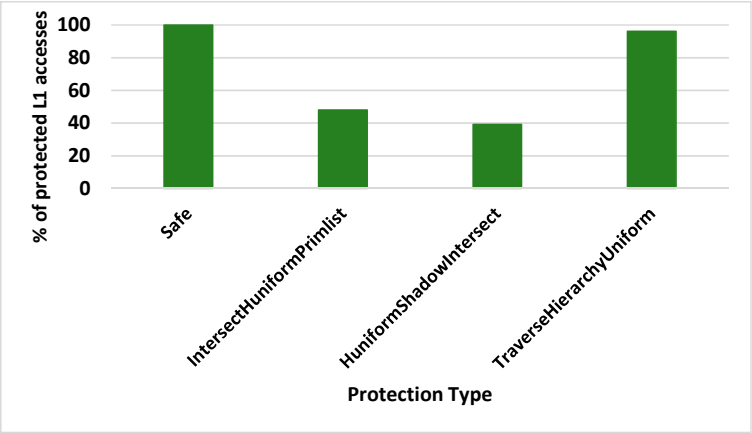
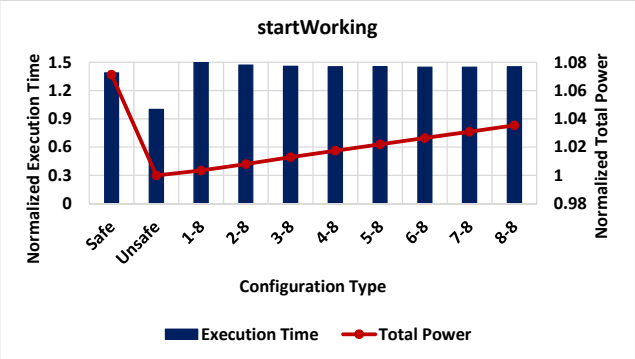


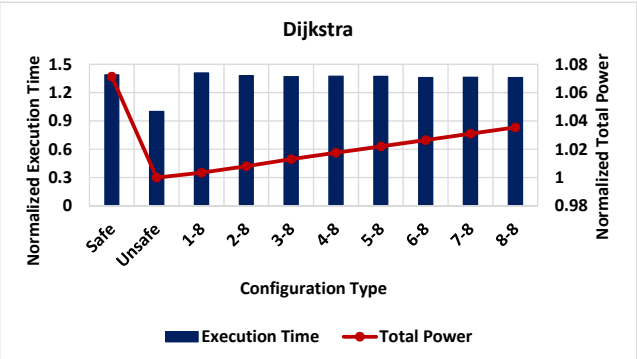
Figure 5.12. Percentage of protected L1 accesses for the Raytrace application.

Even in the 8-8 configuration, which has a separate protected core for each thread, the performance results do not seem close to the safe configuration. It is observed that the cost of thread waiting in the queue is considerably high and the amount of migration (i.e. 182,607) is adversely affecting the performance for this function. On the other hand, each partially safe configuration offers better power consumption result than the safe configuration. As another observation, the 1-8 configuration may consume less power than the unsafe configuration for the functions of Raytrace application. This case is observed for some of the other applications. To make a fair comparison, there are one protected and 15 unprotected cores for the 1-8 configuration; although, we used one protected core and eight unprotected cores actively in the system. So the total number of cores is same for each experiment and there are several idle cores for the partially safe configurations except the case of 8-8 configuration.

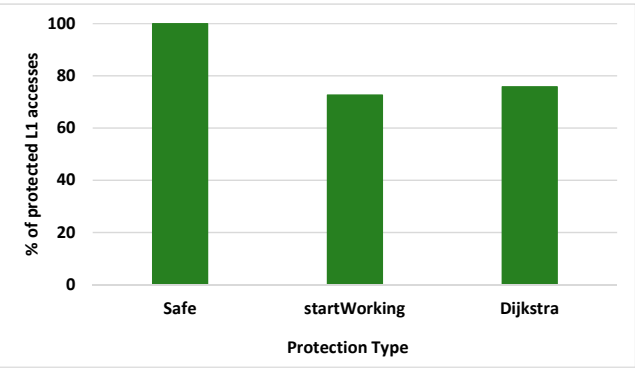
When the HuniformShadowIntersect function is protected, the performance values converge after five protected cores are added. This result shows that the 5-8 configuration has the same performance as the safe configuration, with having lower power consumption. When the TraverseHierarchyUniform function is preserved, the 8-8 partially safe configuration results in better performance and less power consumption than the safe configuration. When IntersectHuniformPrimlist, HuniformShadowIntersect, and TraverseHierarchyUniform functions are preserved, 48%, 40% and 92% of L1 cache accesses can be protected, respectively.



(a) Execution time and power consumption results with protecting startWorking function.

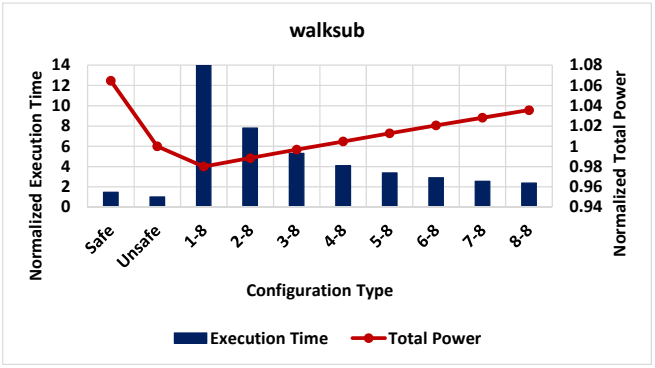


(b) Execution time and power consumption results with protecting Dijkstra function.

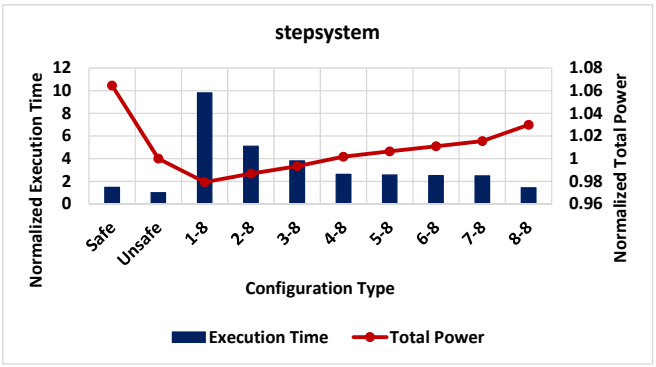


(c) Percentage of protected L1 accesses.

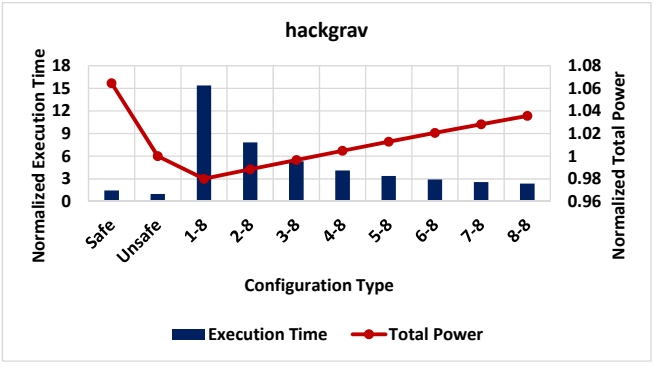
Figure 5.13. Normalized values of execution time and power consumption for the Dijkstra application.



(a) Execution time and power consumption results with protecting walksub function.



(b) Execution time and power consumption results with protecting stepsytem function.



(c) Execution time and power consumption results with protecting hackgrav function.

Figure 5.14. Normalized values of execution time and power consumption for the Barnes application.

Therefore, we can provide reliability results close to the safe configuration with better performance and lower power consumption values when we protect the TraverseHierarchyUniform function.

Figure 5.13 shows the performance, power consumption and reliability results of the Dijkstra application. For this application, both functions are invoked once by each thread; therefore, they show similar trends on the results. Since the startWorking function spends only a small portion of the execution time on the high reliability core(s), the performance values of the partially safe configurations are close to each other, and have near values with the safe configuration. When the Dijkstra function is preserved, it is observed that the performance of the partially safe configuration is slightly better than the safe configuration after the six protected cores are added. The thread migration and queue waiting overheads are not dominant for both functions of the Dijkstra application. On the other hand, it is observed that 72% and 76% of the L1 cache accesses can be preserved when startWorking and Dijkstra functions are protected, respectively.

Figures 5.14 and 5.15 show the performance, power consumption and reliability results of the Barnes application. Since the walksub and hackgrav functions require a high number of migrations, the performance of any partially safe configuration does not seem better than the safe configuration. If there is a high number of function calls, which implies a high number of thread migrations, our proposed partially safe configuration may not have satisfactory performance. On the other hand, when the stepsytem function is protected, the 8-8 configuration, which has a protected core for each thread, has better performance than the safe configuration with lower power consumption. Furthermore, 95% of the total L1 cache accesses can be preserved when the stepsytem function is protected, and 65% of the total L1 cache accesses can be protected when the walksub and hackgrav functions are preserved. Most of the execution time passes inside the stepsytem function (including its descendants) and the partially safe configuration with 8-8 configuration presents a similar reliability behavior with the safe configuration, showing considerable performance and power consumption gains.

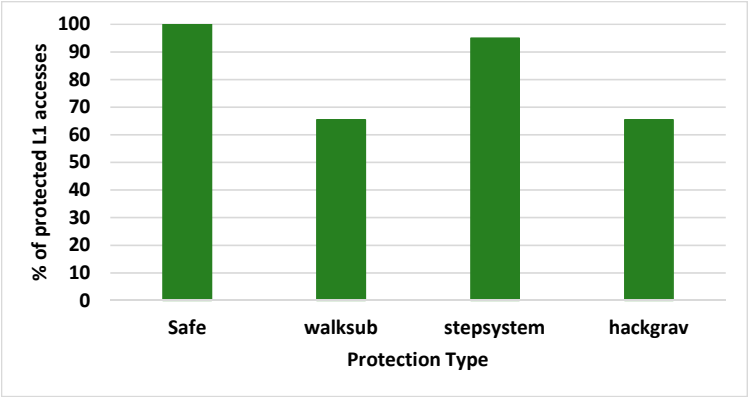
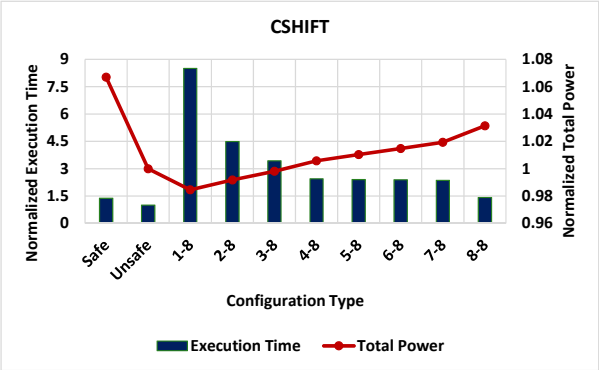


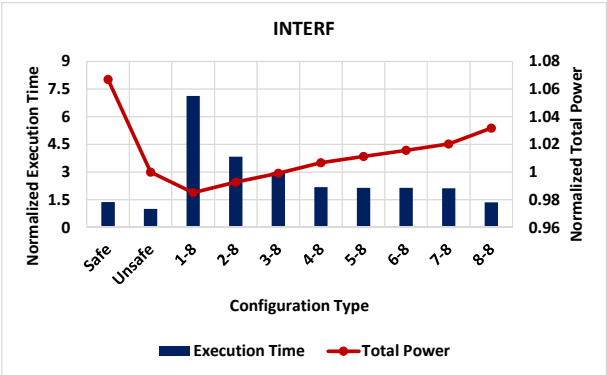
Figure 5.15. Percentage of protected L1 accesses for the Barnes application.

Figures 5.16 and 5.17 show the results of the Water-nsquared application. In all three functions of this application, the partially safe configuration with 8-8 configuration shows close performance results to the safe configuration. Specifically, when the CSHIFT function is protected, 96% of the total L1 cache accesses are preserved, showing close performance and less power consumption results than the safe configuration. The MDMAIN function is called once by each thread, and 97% of L1 cache accesses can be preserved when this function is protected. When the INTERF function is protected, 79% of the L1 cache accesses can be preserved. From these results it can be seen that the partially safe configuration with 8-8 configuration has close performance and reliability results showing less power consumption than the safe configuration.

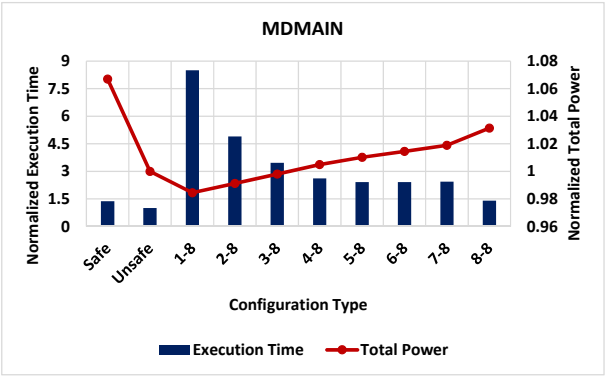
Figures 5.18 and 5.19 show the performance, power consumption and reliability results of the fmm application. The ParallelExecute function has the 96% of the total execution time. Hence, the performance of the partially safe configuration with the 1-8 configuration seems considerably poor relative to the safe configuration for this function. The number of function call and the total number of migrations are not dominant for this function. However, the waiting cost of threads in the queue seems to be high. When we provide an equal number of high reliability and low reliability cores (i.e. the 8-8 configuration), the performance results are improved and they become close to the safe configuration.



(a) Execution time and power consumption results with protecting CSHIFT function.



(b) Execution time and power consumption results with protecting INTERF function.



(c) Execution time and power consumption results with protecting MDMAIN function.

Figure 5.16. Normalized values of execution time and power consumption for the Water-nsquared application.

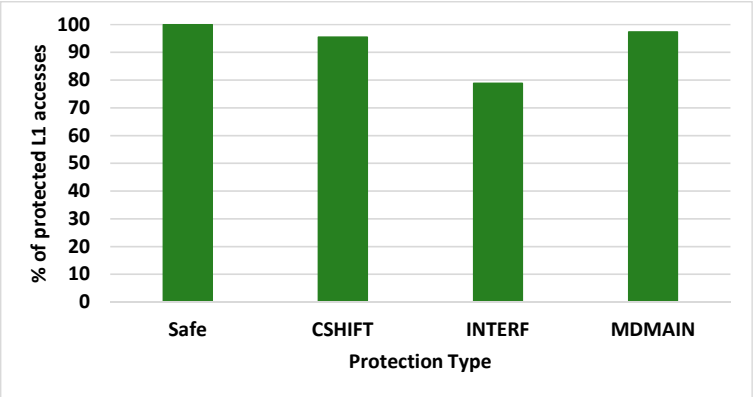
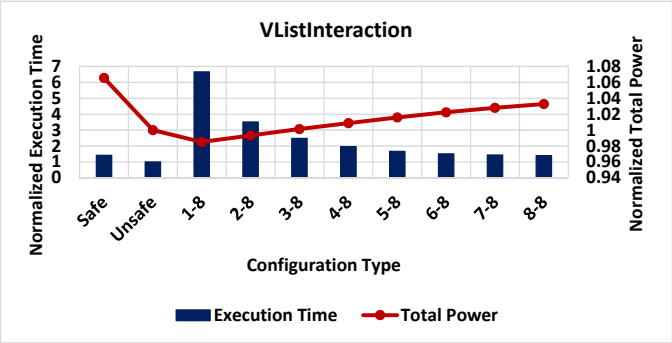


Figure 5.17. Percentage of protected L1 accesses for the Water-nsquared application.

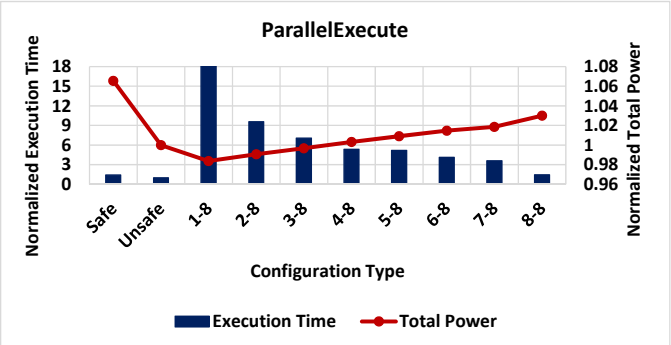
The partially safe configurations with 8-8 configuration protecting the VListInteraction and ListIterate functions show similar behavior in the results. When VListInteraction, ParallelExecute and ListIterate functions are individually protected, 61%, 98% and 81% of total L1 cache accesses can be preserved, respectively. According to these results, the partially safe configuration with 8-8 configuration might be preferred to the safe configuration especially for the protection of ParallelExecute function under the performance and power constraints.

In this section, the experimental set includes the results of the partially safe configuration with different numbers of protected cores for each of the three high-priority functions of each application. While similar protection is provided in every partially safe configuration, different configurations vary in terms of performance and power consumption results. The 8-8 partially safe configuration, where we provide an equal number of protected and unprotected cores, presents close reliability with lower power consumption results than the safe configuration. On the other hand, there are several factors that affect the performance of our partially safe configuration negatively, such as thread migration and queue waiting overheads, while there are no such factors in the safe and unsafe configurations.

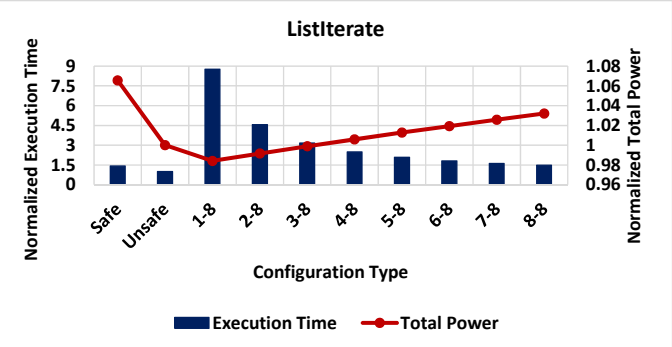
Apart from these, a false serialization effect is observed in the partially safe configuration since a protected core can be used by only one thread at the same time.



(a) Execution time and power consumption results with protecting VListInteraction function.



(b) Execution time and power consumption results with protecting ParallelExecute function.



(c) Execution time and power consumption results with protecting ListIterate function.

Figure 5.18. Normalized values of execution time and power consumption for the fmm application.

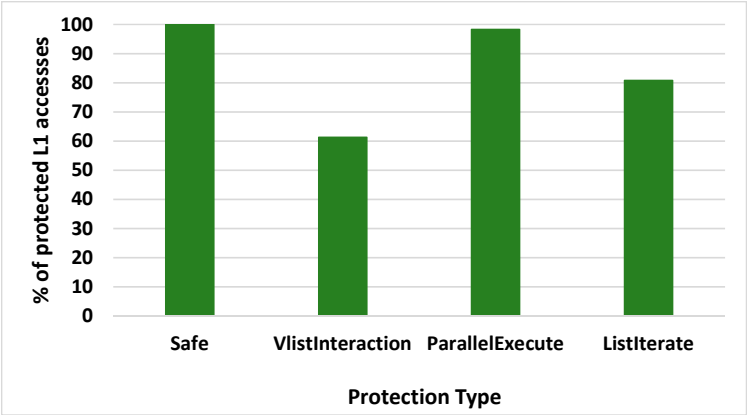


Figure 5.19. Percentage of protected L1 accesses for the fmm application.

When the power consumption results are examined, the proposed partially safe configuration offers lower results than the safe configuration. As another observation, the cost metric should also be evaluated, even if it is not mentioned in the results. Providing ECC protection in L1 caches of all cores has a significant additional memory (21.9% for each core) and hardware costs. The proposed partially safe configuration is superior to the safe configuration in terms of power consumption and cost, and offers similar performance and close reliability results. Our proposed adaptive approach seems to be suitable for the systems where reliability is the main factor under the performance, power and cost constraints.

5.3. Summary

In the first part of this section, we determine reliability-based critical code regions based on user annotations. We profiled the selected applications and the functions that cover 90% of total execution time are used as the critical code regions. The reliability-based critical code regions of the applications are executed using different numbers of protected cores. Experimental studies with different function characteristics validate that our partially safe configuration takes the advantage of protecting only reliability-based critical code regions of the applications and offers significant performance and power savings compared to the safe configuration and lower failure rates compared to the unsafe configuration.

In the second subsection, we provide an enhanced protection mechanism for the reliability-based critical code regions determined as high-priority functions of each application. The high-priority functions are identified by calculating the function vulnerability and criticality metrics based on the function execution times and call graphs, statically. The high-priority functions are protected more conservatively than the other functions by utilizing different partially safe configurations. The experimental results with a diverse set of applications confirm that our partially safe configuration offers close performance and reliability results to the safe configuration with lower power consumption and cost.

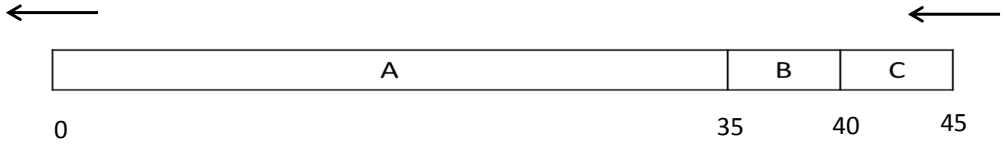
6. SCHEDULING OPPORTUNITIES FOR OUR FRAMEWORK

Our framework with asymmetrically reliable caches achieves a certain level of success in protecting critical code fragments in terms of reliability with marginal increase in execution time and power consumption compared with a fully unprotected system. On the other hand, when compared with a fully protected system, it has comparable reliability results at a lower cost. The queue structure and the assignment of threads to high reliability cores are provided by an FCFS-based scheduling method, in our framework. In this section, different scheduling methods are used besides FCFS and these methods are evaluated in terms of system performance and fairness.

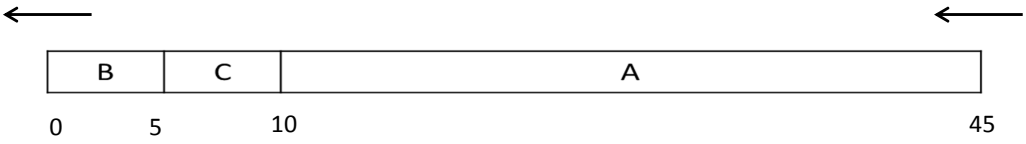
Scheduling in general is a crucial research field for high performance computing especially for the heterogeneous systems. Wrong scheduling decisions may lead to suboptimal performance as well as low fairness among the applications. Our framework utilizes a heterogeneous system in terms of reliability, where the FCFS-based scheduling technique is used for assigning application threads to the protected core(s). When multi-application workloads are considered in our framework, FCFS-based policy may lead to suboptimal performance and low fairness for some workloads. In this study, to improve performance and fairness of our proposed system, we implement and evaluate different types of schedulers with various characteristics for the protected core(s).

6.1. Shortcomings of FCFS Scheduling

The FCFS-based approach prioritizes older requests over newer requests in the queue. However, this approach may unfairly prioritizes an application with a high number of requests over an application with a low number of requests. On the other hand, a request with short burst time may have long waiting time in the queue if there is an older request with long burst time. Suppose that we have a workload with three applications, A, B, C.



(a) Case I - FCFS approach.



(b) Case II - SJF approach.

Figure 6.1. Order of applications at the queue with the FCFS and SJF based scheduling techniques.

Assume that the first request is sent by application A with a burst time of 35 seconds. Then, application B and application C send their requests with burst times of five seconds, respectively at the beginning of the execution. When we apply the FCFS-based scheduling policy, the applications are served according to their arrival order as seen in Figure 6.1(a). In this case, the waiting times of the applications B and C are 35 and 40 seconds, respectively. This may increase the execution time of these applications. However, if we reorder the requests in the queue by prioritizing the requests with shorter burst time over the requests with longer burst time (which is known as Shortest Job First (SJF) scheduling), in this case application A can be served with a 10-second delay, application C can be served with only a 5-second delay and application B can be served without any delay (Figure 6.1(b)). Therefore, we can decrease the waiting times of the applications by reordering the requests in the queue.

6.2. Determining Function Priorities

The reliability-based critical regions of applications are extracted by using static analysis as in Section 5.2.1. The vulnerability metric is used in similar way such that the functions with longer execution time are assumed to be more vulnerable.

The definition of function criticality metric is simplified in this section. When a function is delayed due to a fault or resource unavailability, it affects the application performance. A function that calls a large number of other functions may increase this delay effect. Therefore, we analyze the call graph file generated by gprof and calculate the number of callee functions of each function (i.e. out-degree value). The function criticality metric is measured by accumulating the number of out-degree value by considering the propagation effect.

$$criticality_i = \frac{OutDegree_i}{Max_{j \in Functions}(OutDegree_j)} \tag{6.1}$$

Here, $OutDegree_i$ is the number of callee functions in $function_i$, and $Max(OutDegree_j)$ is the largest number of out-degree in the function call graph. In order to calculate the out-degree value of a function, we count the number of callee functions in one-level depth. A higher value of $criticality_i$ represents that the function delay has a dominant effect on overall application execution time.

To select the order of functions to be protected, we calculate the vulnerability and criticality metrics for each function. We select the functions that maximize the criticality value under the fixed value of vulnerability criteria, which are presented in more detail in Section 6.4.1.

6.3. Different Scheduling Techniques

There are two main goals in modern scheduling techniques: high system performance and high fairness. Our goal, in this work, is to achieve high system performance and high fairness by applying different scheduling techniques rather than the FCFS. In this section, we present the details of different scheduling methods that are utilized in our computational experiments. There are a total of nine methods considered, where six of them are priority-based ones. Additionally, equal-time scheduling, equal-progress scheduling, and threshold-based priority scheduling are the other alternatives presented.

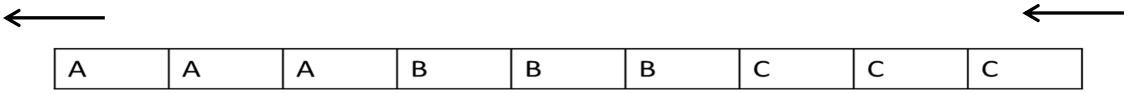
6.3.1. Priority-based Scheduling

Priority-based scheduling technique is one of the frequently used scheduling methods in operating systems. In this technique, each process is given a priority value in advance, and the process with the highest priority is assigned first. Prioritization can be based on memory or any other resource usage in the system. We generate multiple multi-threaded workloads to test the effectiveness of different scheduling methods; therefore, the priority order of the applications can be determined in advance.

Suppose that there are three applications, A, B, C, each of which has three threads. Figure 6.2(a) shows the request order of all threads at a time. Here, a thread of application A is at the front of the queue and followed by a thread of application B. It should be noted that the length of the queue may change dynamically during the execution based on the requests sent by the applications to the protected core(s). Here, we suppose that there are nine threads in the queue at that time. The details such as thread-id of each thread or the burst time of each request are not shown in the figures; however, we assume that the requests of the same application are sent by different threads of it and each request may have different burst time. Threads can be assigned to the protected core(s) according to their arrival order in the FCFS scheduling method (see Figure 6.2(a)). On the other hand, suppose that applications are given a priority order statically, with the highest priority belonging to application A, medium-level priority belonging to application B, and the lowest priority belonging to application C. In this case, the order of the application threads is changed, and the updated queue is seen in Figure 6.2(b). Threads of each application are put together in the queue and take their place according to the priority level of the application.

We propose a set of priority-based scheduling techniques in which the applications are prioritized based on their execution order, cache miss rates, number of requests sent to the protected core(s), or total burst time spent on the protected core(s). These static priority techniques require executing the applications in advance to determine the priority-level of the applications.

(a) Case I - FCFS approach.



(b) Case II - Priority-based approach.

Figure 6.2. Snapshot of the queue with the FCFS and priority-based scheduling techniques.

6.3.1.1. Priority-ldf. Prioritization of the applications is similar to the latest deadline first scheduling technique. Total execution time of a workload is highly dependent on the application that finishes last. Since our aim is to improve performance of our framework, we give the highest priority to the application that finishes last in a workload. The application that has the longest execution time, has a larger impact on the overall schedule length if it is delayed. The lowest priority is given to the application that finishes first. Therefore, when we delay the requests of the first finished application, it may not affect the overall execution time of the workload significantly.

6.3.1.2. Priority-missRate. The priority order is determined according to the cache usage of the applications. Priority of applications with high cache hit ratio is higher than the applications with high miss ratio.

6.3.1.3. Priority-min-reqs. The priority order is determined based on the number of requests sent by the applications to the protected core(s). The priority of the applications with a small number of requests is higher than those with a large number of requests. Since the applications with a large number of requests are frequently assigned to the protected core(s), the delay caused by executing them on the protected core(s) is already high.

On the other hand, the applications with a small number of requests are able to return to their original unprotected cores immediately and continue with the remaining tasks, if they are assigned to the protected core(s) in the first place.

6.3.1.4. Priority-max-reqs. In contrast to the previous method, the applications with a large number of requests are given higher priority over those with a small number of requests.

6.3.1.5. Priority-min-burst. The priority order is based on the total burst time of the applications on the protected core(s), not on the number of requests sent by the applications as in the priority-min-reqs method. The application with the highest number of requests may not be the one with the longest burst time on the protected core(s). Therefore, the priority of applications with shorter total burst time of requests is higher than those with longer burst time of requests.

6.3.1.6. Priority-max-burst. Unlike the previous method, the applications with longer total burst time of requests are set as high-priority applications compared to those with shorter total burst time of requests.

Priority-based methods presented in this section require preliminary information about the applications, which need to be executed once in advance. On the other hand, priority-based methods assign application threads by grouping them, which provides a high cache hit ratio on the protected core(s); in contrast, they may lead to unfairness for low priority applications.

6.3.2. Equal-time Scheduling

The main goal of the priority-based methods is to improve system performance by grouping the threads belonging to the same application according to a predefined criteria. In our system, the applications should use the protected core as much as possible while the protected core should be fair to all applications. In the priority-based methods, the applications with low priority may unfairly slow down when they are located at the end of the queue. A scheduling method should take into account the fairness as well as the system performance.

For equal-time scheduling, all of the applications start the execution with the highest priority value. Priority level of an application is reduced as directly proportional to the total burst time of the application spent on the protected core(s), dynamically during the execution. If there is an application that sends a request with a burst time of one second, the priority level of this application is decreased by one unit. On the other hand, if there is another application that sends a request with a burst time of 10 seconds, in this case the priority level is decreased by 10 units. Thus, fairness among the applications can be satisfied, while ensuring each application has equal time on the protected core.

To give an illustrative example, assume that three separate applications, A, B, C, work together with three threads for each one in a workload. All of the applications are started the execution with the highest priority level. The request order of the applications for a particular time is shown in Figure 6.2(a). Assume that a request for each application has an equal burst time, which is one second. In this case, the priority level of the application A is reduced by one unit after being assigned to the protected core(s) firstly. In the same way, application B is assigned to the protected core(s) secondly and the priority level is reduced by one unit. As shown in Figure 6.2(a), the next request in the queue belongs to application A according to the arrival order; however, there is a request of application C with the priority level of one. Therefore, the thread of application C can be assigned next and the priority level of this application is decreased by one unit.

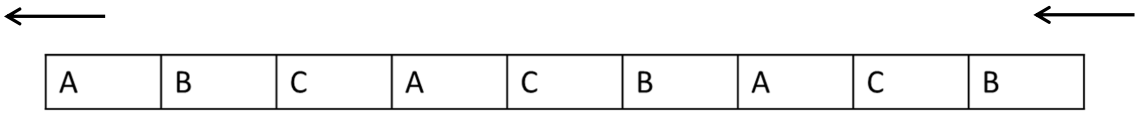


Figure 6.3. Snapshot of the queue with the equal-time scheduling.

In this way, all of the application threads can get their new positions according to the new priority order dynamically. The updated queue, according to the equal-time scheduling, is shown in Figure 6.3. A different application may have a chance to be scheduled at each turn if the burst time of each application request is equal to each other.

The equal-time scheduling updates the queue by taking into account the fairness. The method tries to equalize the total burst time of each application on the protected core(s) at each turn. The major advantage of this method is the lack of prior information about applications. Priority levels of the applications are updated dynamically at runtime after the assignment of each request to the protected core(s). Therefore, we have to keep the information of the total burst time spent on the protected core for each application at each scheduling point. The main disadvantage of this method might be the case that a thread from a different application is assigned to the protected core at each scheduling point. This may cause performance degradation by increasing cache miss rates of the applications on the protected core(s).

6.3.3. Equal-progress Scheduling

Each application may send a different number of requests with different burst times. When we try to equalize the amount of time spent on the protected core(s) for each application as in the equal-time scheduling, some of the applications can complete half of the total requests while some of them can complete only a small part of them during the same time. Therefore, we used equal-progress scheduling in this section, which prioritizes applications based on the percentage of requests executed on the protected core(s).

In this scheduling technique, each application starts the execution with the highest priority-level. When an application is mapped to the protected core(s), its priority level is decreased by the progress of its requests. The progress of an application is the ratio of time spent on the protected core(s) until now, over the total burst time of all requests for the same application. The total burst time of the applications spent on the protected core(s) should be known in advance. This technique monitors the progress of each application's requests and dynamically updates the priority levels of them. The application with the minimum progress can be scheduled to the protected core(s) for the next turn.

6.3.4. Threshold-based Priority Scheduling

In priority-based techniques, applications with low priority-levels may unfairly slow down since they are located at the end of the queue. In this scheduling algorithm, we adapt priority-based scheduling by taking into account the last scheduled application. All applications are given priority values based on a predefined criteria as described in Section 6.3.1. The priority levels of the applications are not changed dynamically at runtime. Additionally, this algorithm needs to record the Application ID of the last scheduled request and the number of waiting requests belonging to each application in the queue. At each scheduling point, the scheduler compares the Application ID of the request at the head of the queue with the the Application ID of last scheduled request.

- If the Application IDs of both requests are not the same, then it searches a waiting request belonging to the last scheduled application in the queue.
- If the scheduler finds such a request in the queue, then it compares the difference of the priority levels of the last scheduled application and the application located at the head of the queue.
- If the difference between their priority levels is less than the Locality Threshold value, then the scheduler takes the waiting request of the last scheduled application to the head of the queue, and schedules it for the next turn.

The Locality Threshold value determines whether the last scheduled application with low priority level has a chance to be scheduled for the next turn. If the difference between the priority levels of the applications is too large, then the application with low priority level is not taken, even if it is scheduled last. In this case, the scheduler selects the request with the highest priority level.

Here, we consider the previous illustrative example in order to clarify this algorithm (see Figure 6.2(a)). Applications are given the following priorities, A(1), B(2), C(3), where lower numbers correspond to higher priorities. Therefore, the highest priority belongs to application A. We assume that the last scheduled request belongs to B at the beginning of that scheduling point, and the Locality Threshold value is set as two a priori. In this case, although the application A has the highest priority level, application B is scheduled next since it is the last scheduled application and the difference between the priority levels of applications A and B is less than the Locality Threshold. When we perform this scheduling technique to the queue presented in Figure 6.2(a), the resultant queue is shown in Figure 6.4. This technique simply prioritizes the last scheduled application even if it has a lower priority level than the application located at the head of the queue. It should be noted that if application C was the last scheduled application, the requests of this application would not be placed at the head of the queue, since the difference between the priority levels of applications A and C is not less than two. Therefore, we prioritize the last scheduled application up to a point determined by the Locality Threshold.

When the proposed nine techniques presented in this part are compared with the FCFS, it has the smallest computation overhead since it does not require any types of additional arrangements in the queue. On the other hand, there are static priorities in priority-based scheduling algorithms and the queue is reordered based on the priority levels of applications. The priorities are dynamically changing during the execution of equal-time and equal-progress scheduling techniques while the latter one requires prior knowledge. The queue is reordered based on static priorities by considering the last scheduled application in the threshold-based priority technique.

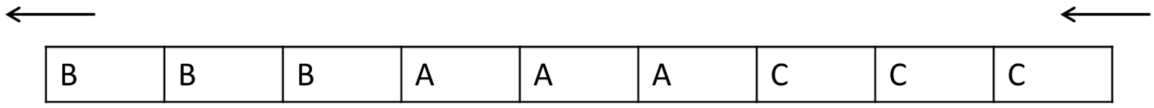


Figure 6.4. Snapshot of the queue with the threshold-based priority scheduling.

6.4. Methodology

6.4.1. Applications

A set of parallel applications are chosen to test the performance of different scheduling algorithms. A total of seven applications are considered, where one application is selected from the ParMiBench suite [68] and five applications are selected from the SPLASH-2 benchmark suite [74]. These applications (Cholesky, Raytrace, Barnes, Water-nsquared, fmm, Dijkstra) are similar with the ones presented in Section 5.2.2.1. Additionally, we use parallel implementation of FFT application with transpose algorithm [81]. Fast Fourier Transform (FFT) is an algorithm to compute Discrete Fourier Transform (DFT) which is a mathematical transform requiring complex number calculations and used in various applications including time series, partial differential equations, and digital signal processing [82]. The transpose algorithm, which involves a matrix transposition operation, requires smaller amount of communication among cores. The remaining applications from the given benchmark suites cannot be selected for the comparison study, since the most of the execution time is spent inside a single function.

We profile the applications with GNU profiler and implement a python code that examines flat profile and call graph files to compute function vulnerability and criticality values. Table 6.1 presents the profiling results of the applications. Application names and functions according to their execution time order are listed in the first two columns, respectively. Execution time percentage spent in the functions as alone and cumulative (including their children) are shown in the fourth and fifth columns, respectively.

The fifth column shows the percentage of time that the protected core is occupied by these functions. The total number of function calls is shown in the sixth column, which displays the quantity of thread migrations for the protected core(s) as well as the queue load. Finally, the last two columns show the vulnerability and criticality values of the functions.

For limiting selected functions for protection, a vulnerability and criticality based criteria is need to be considered. The vulnerability metric is related to the execution time of a function, which shows that the protected core is busy with the function during this time period. Since we assume that only a single thread can be executed on the protected core at one time, we come across performance degradation while protecting the function. Consequently, we limit performance degradation up to 30% in order to execute the functions on the protected core(s). Then, we select the functions that maximize the criticality under the fixed value of vulnerability criteria for each application. The functions that maximize the criticality value with vulnerability values less than 0.3 are selected for each application with the following formulation:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} (criticality_i * x_i) \\ \text{subject to} \quad & \sum_{i=1}^{n} (vulnerability_i * x_i) < 0.3, \\ & x_i \in \{0,1\},\ \forall i \in Functions \end{aligned} \tag{6.2}$$

Therefore, we select the functions with the asterisk sign (*) in the corresponding rows of Table 6.1. It should be noted that only a single function or several ones might be selected for an application according to the constraints mentioned above and we ignore the functions with vulnerability values less than 0.01.

Table 6.1. Profiling results of seven applications.

Application Name	**Function Name**	**Ex. Time Percentage (alone)**	**Ex. Time Percentage (cumulative)**	**# of Calls**	**vulnerability**	**criticality**	
Cholesky	1. ModifyTwoBySupernodeB	61.90%	61.90%	167,354	0.6368	0.0000	
	2. TriBSolve	4.31%	4.31%	1	0.0448	0.0000	
	3. FillIn	4.14%	4.31%	13,992	0.0404	0.0080	*
	4. ScatterSuperUpdate	3.24%	3.24%	3,530	0.0314	0.0000	
	5. ComputeNZ	2.26%	2.26%	1	0.0224	0.0000	
	6. FindDomStructure	2.18%	2.20%	1,187	0.0224	0.0000	
Raytrace	1. TripeIntersect	23.20%	23.20%	736,266	0.2530	0.0000	
	2. lookup_hashtable	6.30%	6.30%	349,572	0.0723	0.0000	
	3. IntersectHuniformPrimlist	6.19%	25.20%	182,607	0.0602	0.0148	*
	4. push_down_grid	6.08%	6.08%	105,807	0.0602	0.0074	*
	5. PolypeIntersect	4.42%	4.42%	125,438	0.0482	0.0000	
	6. step_grid	3.31%	3.31%	292,127	0.0361	0.0000	
	7. Shade	2.65%	21.00%	9,801	0.0241	0.0370	*
	8. next_nonempty_leaf	2.32%	22.60%	268,496	0.0241	0.0296	*
	9. HuniformShadowIntersect	2.32%	17.40%	9,601	0.0241	0.0370	*
Barnes	1. gravsub	40.81%	40.81%	15,459,166	0.4089	0.0000	
	2. subdivp	29.17%	29.17%	16,737,571	0.2911	0.0000	
	3. walksub	25.49%	95.50%	65,536	0.2556	0.0682	*
	4. loadtree	0.91%	1.80%	65,536	0.0089	0.0909	*
Water - nsquared	1. CSHIFT	55.10%	55.10%	1,962,240	0.5591	0.0000	
	2. INTERF	26.06%	73.80%	4	0.2688	0.1333	*
	3. POTENG	5.25%	16.03%	1	0.0538	0.0667	*
	4. UPDATE_FORCES	3.71%	3.71%	40,231	0.0430	0.0000	
fmm	1. VListInteraction	61.19%	61.19%	122,174	0.6265	0.0000	
	2. UListInteraction	13.19%	13.19%	35,938	0.1355	0.0000	
	3. XListInteraction	4.58%	4.58%	8,184	0.0482	0.0000	
	4. WListInteraction	3.31%	3.31%	8,184	0.0331	0.0000	
	5. ComputeSelfIteration	2.10%	2.10%	4,697	0.0211	0.0000	
	6. ComputeMPExp	1.90%	1.90%	4,697	0.0181	0.0000	
	7. ShiftLocalExp	1.87%	1.87%	6,262	0.0181	0.0000	
	8. EvaluateLocalExp	1.81%	1.90%	4,697	0.0181	0.0120	*
Dijkstra	1. dijkstra	3.92%	4.40%	1	0.0391	0.0909	*
	2. main	1.10%	1.80%	1	0.0175	0.1818	*
	3. enqueue	0.50%	0.50%	2,456	0.0050	0.0000	
FFT	1. binary2decimal	13.09%	13.09%	319,410	0.1298	0.0000	
	2. iterative_FFT	7.57%	30.00%	1	0.0769	0.6250	*
	3. copyArray	3.65%	3.65%	212,940	0.0385	0.0000	
	4. initializeArray	2.85%	2.85%	212,940	0.0288	0.0000	
	5. decimal2binary	2.85%	4.30%	106,470	0.0288	0.1250	*

6.4.2. Multi-threaded Workloads

In our experiments, we create several workloads containing multiple multi-threaded parallel applications. While selecting applications for the workloads, we consider the number of requests sent by each application to the protected core(s) presented in Table 6.2. These numbers show the quantity of thread migrations for each application. We cluster the applications based on their total number of requests as small-, medium-, and large-size applications. The Cholesky, Water-nsquared, Dijkstra, and FFT applications are determined as small-size applications. The fmm and Raytrace applications are assigned as medium-size applications, and the Barnes application is identified as a large-size application. In the experimental study, the letters 'C', 'R', 'B', 'W', 'Fm', 'D', and 'F' stand for the applications Cholesky, Raytrace, Barnes, Water-nsquared, fmm, Dijkstra, and FFT, respectively.

We construct four different workloads, which are CWF, RFFm, RBFm, and 7app, for the experiments. A small-size workload 'CWF' is constructed that does not stress the protected core(s). A medium-size workload 'RFFm' is created in which two applications are selected from the medium-size applications and one application is selected from the small-size applications. Our third workload 'RBFm', a large-size workload, sends a relatively larger number of requests to the protected core(s) than the medium-size and small-size workloads. Our last workload, '7app', is a 7-application workload that contains all of the applications presented here. This workload is a representative workload that contains applications with different characteristics.

6.4.3. Evaluation Metrics

Our comparison study validates the effectiveness of scheduling algorithms with respect to both system performance and fairness perspectives. For the system performance perspective, we utilize average speedup and harmonic speedup metrics.

Table 6.2. Number of requests of each application for the protected core(s).

Application Name	**# of requests**
Cholesky	1
Raytrace	35,901
Barnes	131,072
Water-nsquared	80
fmm	4,697
Dijkstra	8
FFT	120

The average speedup of a workload using a scheduling algorithm is defined as the average of per application speedups (in terms of execution time) with respect to the baseline scheduler, the FCFS. The average speedup is calculated as follows [83]:

$$AS = \frac{1}{n} \sum_{i=1}^{n} \frac{T_i(baseline)}{T_i(approach)} \tag{6.3}$$

where n is the number of applications in a workload, $T_i(baseline)$ is the execution time of the application i with the FCFS technique, and $T_i(approach)$ is the execution time of the application i with the proposed scheduling technique.

The average speedup is a commonly used metric to measure the system performance. However, it can be misleading by unfairly favoring a single application with disproportionate improvement. Therefore, we use also a harmonic speedup metric [52] that is derived from the fair speedup metric [84]. The relative speedup of a workload using a scheduling algorithm is defined as the harmonic mean of per application speedups (in terms of execution time) with respect to the FCFS. The harmonic speedup is calculated as follows [52]:

$$HS = \frac{n}{\sum_{i=1}^{n} \frac{T_i(approach)}{T_i(baseline)}} \tag{6.4}$$

This metric takes into consideration fairness as well as system performance, since the harmonic mean of a vector is maximized when the vector elements are equal.

On the other hand, the fairness perspective is an important design goal in scheduler implementations for concurrently running applications [52]. Each application should be provided with proportional resource usage according to its demand under a fair operation. The fairness among n concurrently running applications is calculated as the Jain's fairness index [85] applied to the relative slowdown. The fairness metric is calculated by considering the relative slowdown of each application as follows [52]:

$$Fairness = \frac{\left(\sum_{i=1}^{n} \frac{T_i(approach)}{T_i(alone)}\right)^2}{n \times \sum_{i=1}^{n} \left(\frac{T_i(approach)}{T_i(alone)}\right)^2} \tag{6.5}$$

where n is the number of applications in the workload, $T_i(approach)$ is the execution time of the application i in the workload with the proposed scheduling technique, and $T_i(alone)$ is the execution time of the application i in the workload in which there is no scheduler and the workload is executed as is. In a fair operation, each application should be equally slowed down with the proposed scheduling technique. The value of the fairness varies between zero and one, and the higher value is the better.

Maximum slowdown, which is the maximum value of slowdowns among the applications in the workload, is considered as a fairness metric in several studies [55–60]. This metric implies the slowdown of applications in the workload due to resource sharing. According to this metric, the scheduling method that has the lowest maximum slowdown provides the maximum fairness; hence lower values are better.

6.5. Evaluation

6.5.1. Experimental Setup

In this study, we use three 3-application workloads and a single 7-application workload in which each application is executed with eight threads for both cases. A 16-core system is simulated where 12 of the cores are left unprotected and four of them are protected for the 3-application workloads. In this case, there are four unprotected cores per application and two threads per core for the 3-application workloads. On the other hand, 14 unprotected and two protected cores are utilized for the 7-application workload. In this case, there are two unprotected cores per application and four threads per core. Firstly, we perform the experiments by providing only one protected core to compare different scheduling techniques relative to the FCFS in terms of performance and fairness. Then, we increase the number of protected cores to two for all workloads and increase it to four for only the 3-application workloads due to hardware limitations. Based on the results of a set of pre-experiments, the value of Locality Threshold is set to two for the threshold-based priority technique.

6.5.2. Experimental Results

6.5.2.1. Comparison of Average Speedup Metric. Figure 6.5 plots the normalized average speedup metric of all scheduling algorithms for various workloads with respect to the FCFS technique. We observe that, in all cases, there is a scheduler approach performing better than the FCFS technique for all workloads. The best scheduling approach may vary for each workload according to the number and the timing of the requests sent by the applications. The techniques, which prioritize the applications according to the minimum number of requests or minimum burst time on the protected core(s), perform well for the CWF, RBFm, and the 7-application workloads. On the other hand, the equal-time scheduling technique has the best performance for the RFFm workload.

We obtain 42% better performance results with priority-min-burst technique relative to the FCFS for the CWF workload. The priority order of the applications is the same for the priority-min-burst and priority-min-reqs techniques; hence, both of them show similar results. For the same workload, equal-time scheduling and threshold-based priority techniques show 32% and 24.5% better performance results than the FCFS technique, respectively. For the RFFm workload, we observe 94% performance improvement with the equal-time scheduling relative to the FCFS approach. In the same way, 70% performance improvement is obtained with priority-min-burst and priority-max-reqs techniques relative to the FCFS, where the priority order of the applications is similar in both cases. Under the threshold-based priority method, the same workload achieves 60.5% better performance than the FCFS. On the other hand, we achieve 60%, 46%, 38.7%, and 25.8% better performance results with the priority-min-burst, threshold-based priority, equal-time, and priority-min-reqs scheduling techniques for the 7-application workload, respectively.

In the priority-min-burst method, which performs better than the FCFS for all workloads, applications with low total burst time on the protected core(s) are able to finish their work as soon as possible and continue the remaining work by running on the unprotected cores. This positively affects the overall performance results for all workloads. Similarly, priority-min-reqs method, which has better performance than the FCFS for the majority of the workloads, may be preferred in cases where the total burst times of the applications spent on the protected core(s) are unknown, but the total number of requests to send is known. In the same way, the threshold-based priority method starts with the priority order of the priority-min-burst method, and it also considers the last scheduled application. Therefore, this method shows close performance results with the priority-min-burst method. The equal-time scheduling technique, which tries to equalize the burst time of applications on the protected core(s), shows good performance results relative to the FCFS.

On the other hand, the techniques that prioritize the applications based on the maximum number of requests or maximum burst time on the protected core(s) show poor performance relative to the FCFS, with the exception of the RFFm workload.

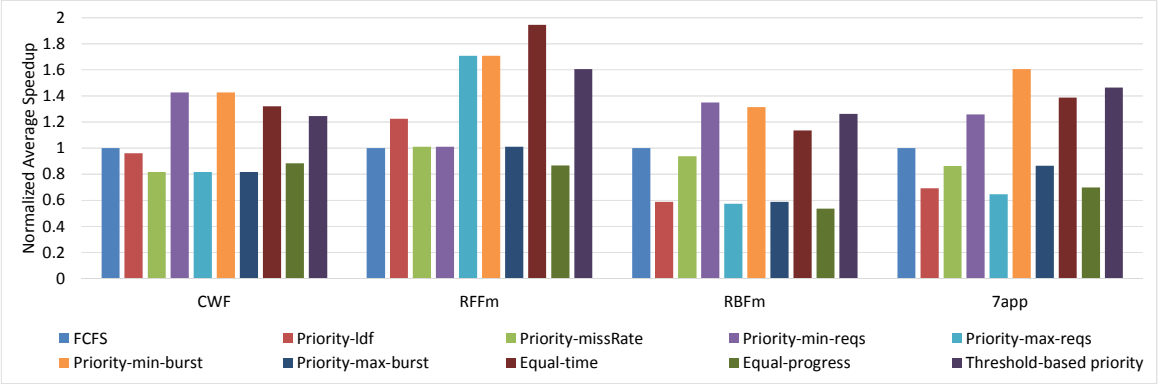


Figure 6.5. Average speedup results.

In the RFFm workload, with only three applications, the priority order of the applications might be similar for various priority-based scheduling techniques and they are the same for priority-min-burst and priority-max-reqs methods. This clearly shows that the applications with a large number of requests might be the applications with minimum total burst time on the protected core(s). In the priority-max-reqs or priority-max-burst methods, the applications that have a small number of requests or low total burst time on the protected core(s) are unfairly placed at the end of the queue and they are affected negatively by this delay. This situation causes performance degradation for the majority of the workloads.

Similarly, the techniques that prioritize applications based on the latest deadline in the workload or L1 cache miss rates do not present better performance results than the FCFS for the majority of the workloads, except for the RFFm. While the priority-missRates method shows similar performance results as the FCFS, the priority-ldf achieves 22.5% better performance results than the FCFS for the RFFm workload. Prioritization of the last finished application over the remaining applications might unfairly delay the applications with low execution time in the priority-ldf method. However, for the RFFm workload, prioritizing the applications with high execution time does not delay the applications with low execution time; therefore, performance improvement can be observed for only this workload. For the priority-missRate technique, the hit rates of the applications are so close to each other and they are generally around 99%.

Therefore, the prioritization technique based on cache miss rates does not perform well for these applications.

6.5.2.2. Comparison of Harmonic Speedup Metric. Figure 6.6 presents the normalized harmonic speedup metric for various workloads with respect to the FCFS technique. We observe similar improvement trends for each scheduling technique with the average speedup metric, but with different magnitudes of improvement. The harmonic speedup metric takes into consideration the fairness metric as well as the performance metric in multi-application workloads. The magnitudes of improvement are fewer in the harmonic speedup metric than the average speedup metric, since the aggressive performance improvement in a single application may dominate the overall speedup results in the average speedup metric.

We obtain 24% better performance results with the priority-min-burst and the priority-min-reqs methods than the FCFS for the CWF workload. Under the equal-time and threshold-based priority methods, the same workload achieves 22.5% and 19% better performance results than the FCFS. For the RFFm workload, 28.9% performance improvement is obtained with the equal-time scheduling relative to the FCFS as in average speedup metric with smaller magnitude of improvement. For the same workload, 14.6% better performance results for the priority-min-burst and the priority-max-reqs methods and 13.6% performance improvement for the priority-ldf method are obtained relative to the FCFS.

We observe 23.7% and 11.9% performance improvements with the priority-min-reqs and equal-time methods relative to the FCFS for the RBFm workload. An important observation about this workload is that while the priority-min-burst method performs better than the FCFS with respect to the average speedup metric, the same method shows worse performance in terms of harmonic speedup metric. This clearly indicates that an aggressive performance improvement on a single application dominates the overall performance in average speedup calculation for this workload.

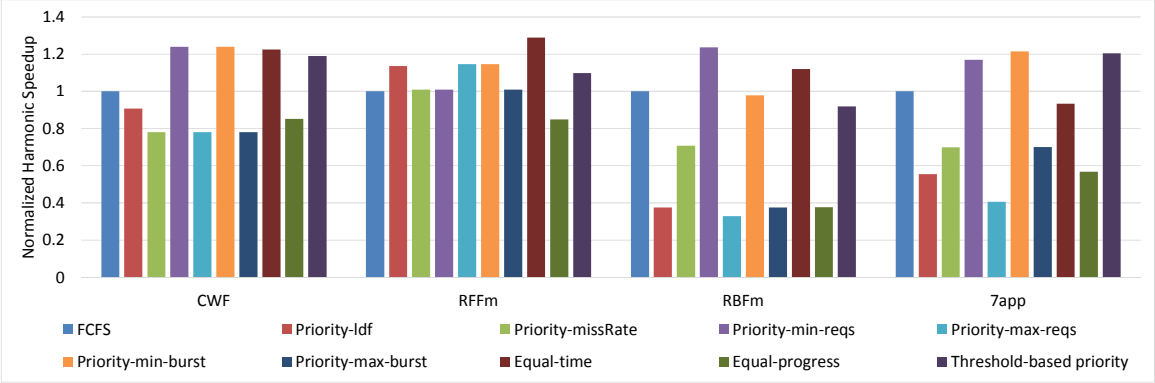


Figure 6.6. Harmonic speedup results.

On the other hand, for the 7-application workload, we obtain 21.4%, 20.5%, and 17% performance improvements with the priority-min-burst, threshold-based priority, and the priority-min-reqs methods, respectively.

An important observation about the performance results presented here is that the equal-progress scheduling, which tries to equalize the progress of the requests for each application, shows poorer performance than the FCFS for all workloads. Better performance results with different scheduling techniques can be obtained by reordering the requests in the queue. However, if the amount of the reordering is too large, then there is a performance degradation due to the high cache miss rates on the protected core. This is the case of the poor performance results for the equal-progress scheduling.

6.5.2.3. Comparison of Fairness Results. We compare different scheduling techniques for various workloads in terms of fairness. While the fairness results based on Jain's fairness index (a higher-is-better-metric) are presented in Figure 6.7, the fairness results based on maximum slowdown (a lower-is-better-metric) are shown in Figure 6.8. Our first observation is that the priority-min-burst, equal-time, and threshold-based priority methods achieve better fairness results than the FCFS for all workloads in terms of both fairness metrics.

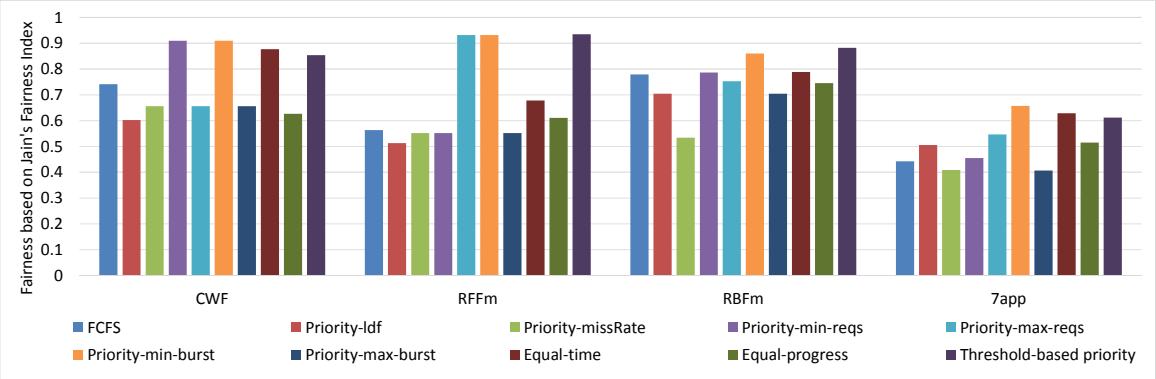


Figure 6.7. Fairness results based on Jain's fairness index.

Our second observation is that while the performance results of the equal-progress scheduling technique are poor for the RFFm and 7-application workloads, the fairness results based on Jain's fairness index of the same scheduling are 8.4% and 16.3% better for these workloads, respectively.

Based on Jain's fairness index, we obtain 22.5%, 65.4%, 10.4%, and 48.4% better fairness results than the FCFS with the priority-min-burst method for the CWF, RFFm, RBFm, and 7-application workloads, respectively. The equal-time method achieves 18.2%, 20.4%, and 41.9% better fairness results for the CWF, RFFm, and 7-application workloads, respectively. Lastly, the threshold-based priority method has 15%, 66%, 13.2%, and 38.1% better fairness results relative to the FCFS for the CWF, RFFm, RBFm, and 7-application workloads, respectively.

When we analyze the maximum slowdown among the individual slowdown rates of the applications, priority-min-burst, equal-time, and threshold-based priority techniques show better results than the FCFS for all workloads. Since the maximum slowdown metric can be quite affected by an aggressive performance degradation of a single application, the Jain's fairness index is mainly used as the fairness metric, and the maximum slowdown metric is presented for comparative purposes.

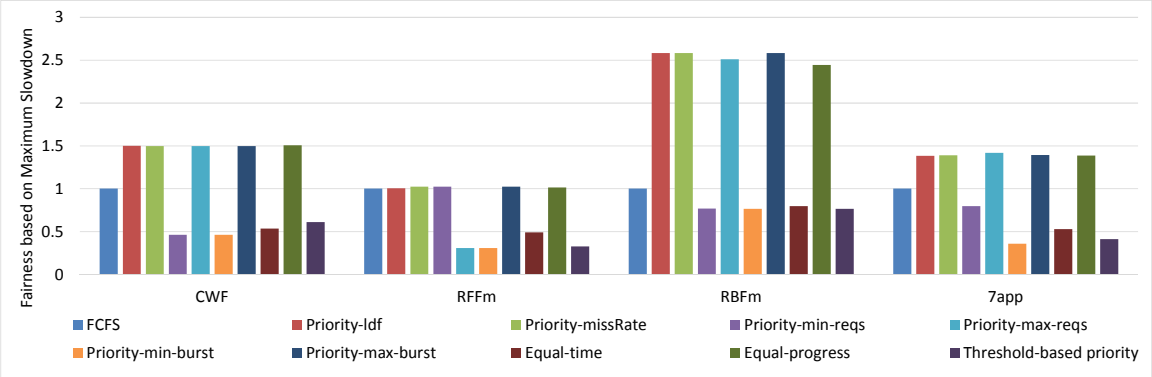


Figure 6.8. Fairness results based on maximum slowdown.

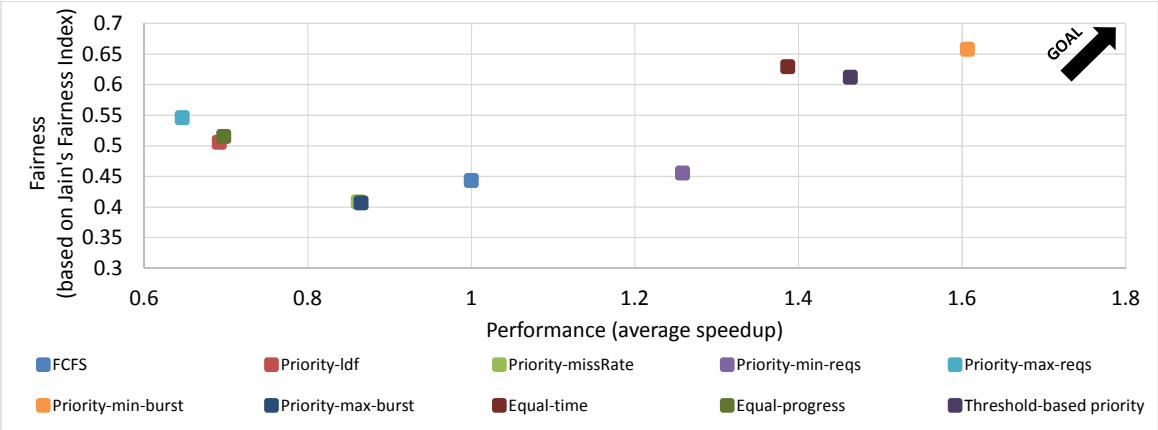


Figure 6.9. Pareto plot of performance and fairness for the 7-application workload.

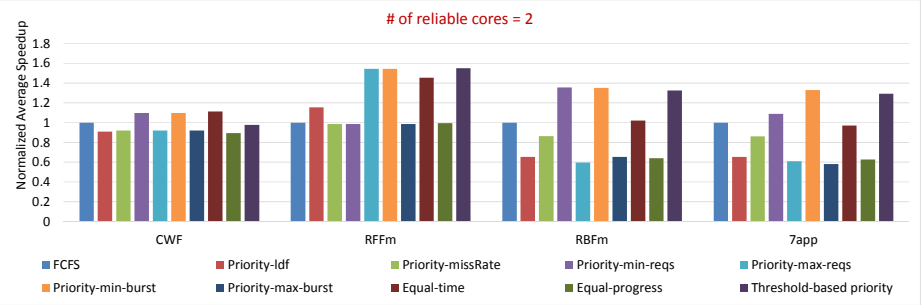
The 7-application workload with all of the applications presented here is a representative workload to make efficient inferences about different scheduling techniques. Therefore, the pareto plot of the performance (in terms of average speedup) and the fairness (based on Jain's fairness index) for the 7-application workload is presented in Figure 6.9. We observe that the priority-min-burst is the most optimal scheduling method in terms of both performance and fairness for the 7-application workload. Furthermore, the priority-min-reqs, equal-time, and threshold-based priority techniques are much better than the FCFS technique for the 7-application workload.

6.5.2.4. Experiments with Different Numbers of Protected Cores. In the previous sections, we report the experimental results by providing only one protected core for each workload. When we increase the number of protected cores, performance improvement is observed in each scheduling technique since multiple protected cores can take the requests in parallel. There is a single global queue for multiple protected cores and they can take the requests from the queue in a mutually exclusive way.

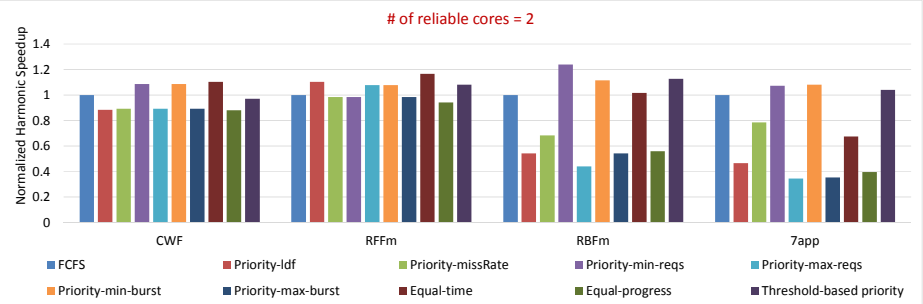
The performance and fairness results based on all metrics for each workload on the system, with two protected cores, are presented in Figure 6.10. Performance results of each workload show similar trends with providing one protected core; however, the magnitude of improvement is less in the system with two protected cores. For the CWF workload, the performance results of the priority-min-burst, priority-min-reqs, and threshold-based priority techniques are close to each other, where they show approximately 10% better performance than the FCFS technique on both harmonic and average speedup metrics. On the other hand, fairness results of these scheduling techniques are slightly better based on Jain's fairness index and roughly 20% better based on maximum slowdown. For the RFFm workload, the performance results of the priority-min-burst method are 54.3% better in terms of average speedup, and the fairness results of the same technique are 62.4% better in terms of Jain's fairness index. On the other hand, for the RBFm workload, the priority-min-reqs method has a 35.6% better average speedup and 5.5% better fairness value in terms of Jain's fairness index.

For the 7-application workload, the results are similar to the system with one protected core, where the priority-min-burst technique has 33% better performance and 42.3% better fairness values. In short, there is a similar behavior between the systems with one or two protected cores for the majority of the workloads based on all metrics.

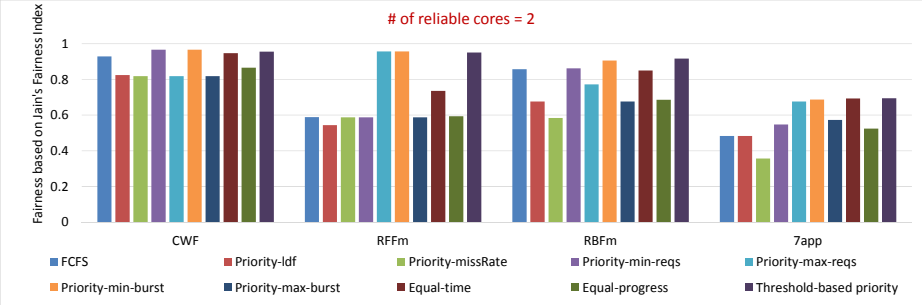
When we provide four protected cores, the performance and fairness results of different scheduling techniques become closer to each other (see Figure 6.11). Furthermore, the difference among the scheduling techniques disappears for the CWF workload with a small number of total requests.



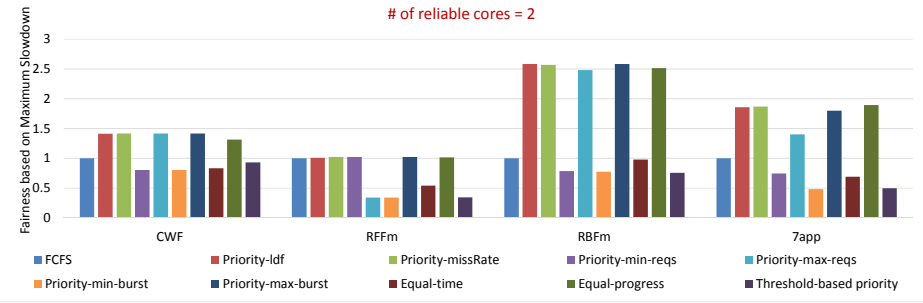
(a) Average speedup results.



(b) Harmonic speedup results.

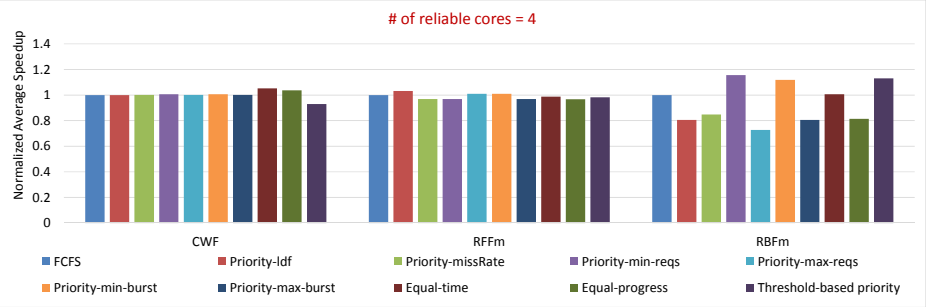


(c) Fairness results based on Jain's fairness index.

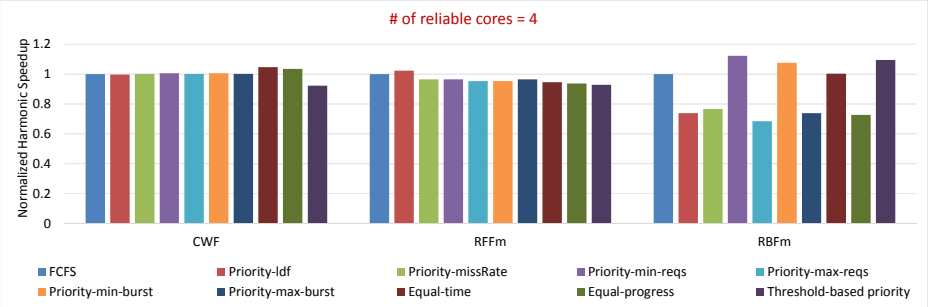


(d) Fairness results based on maximum slowdown.

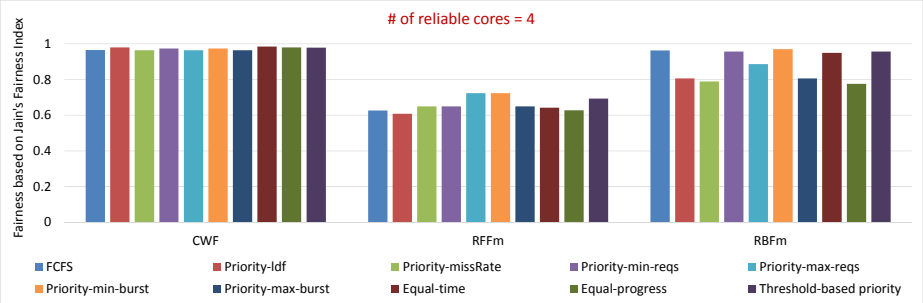
Figure 6.10. Results of different workloads for the system with two protected cores.



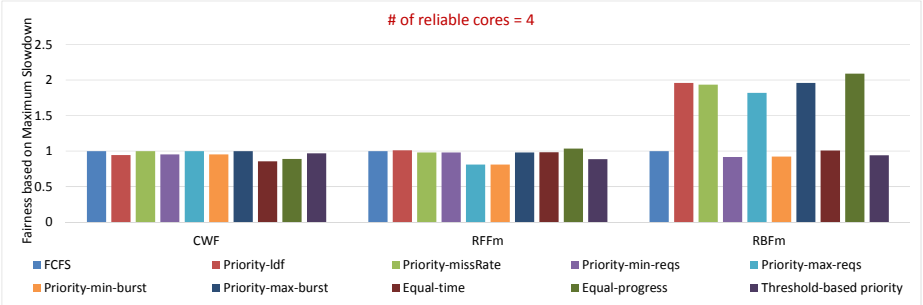
(a) Average speedup results.



(b) Harmonic speedup results.



(c) Fairness results based on Jain's fairness index.



(d) Fairness results based on maximum slowdown.

Figure 6.11. Results of different workloads for the system with four protected cores.

For this workload, the most optimal scheduling technique becomes as equal-time method, which has 5% better performance with close fairness results. Similar results with slight differences are observed for the RFFm with medium size workload. However, when we analyze the RBFm workload with relatively large number of requests, different scheduling techniques have different results based on all metrics. The priority-min-burst, priority-min-reqs, and equal-time methods seem to be optimal scheduling algorithms in terms of all performance and fairness metrics among the scheduling methods for this workload.

6.5.2.5. Detailed Analysis Based on 7-application Workload. In this section, we observe the behavior of the applications with different scheduling techniques for the 7-application workload with the case of one protected core. Figure 6.12 shows the differences in the improvement or worsening for execution time of the applications under each scheduling method relative to the FCFS. The priority-min-burst method has good performance values where four of the seven applications in the workload have smaller execution times, two of them have approximately the same execution time, and only one application has worse performance. While no application has worse performance in the priority-min-reqs method, the improvement in the execution times of the applications is not as much as the priority-min-burst method. When the priority-max-reqs method is analyzed, the application with only one request, the Cholesky application, is unfairly sent to the end of the queue, and the performance is aggressively affected by this delay.

The execution time distributions of the applications spent on the protected and the unprotected cores are shown in Figure 6.13 with three representative scheduling methods. Indeed, the total time spent on the protected core for each application with different scheduling technique is approximately equal; however, the total execution time of the applications may vary according to the scheduling method. Accordingly, the proportion of the execution time spent on the protected core may vary. Figure 6.13(a) shows the execution time distributions of the applications on the protected and the unprotected cores with the FCFS method.

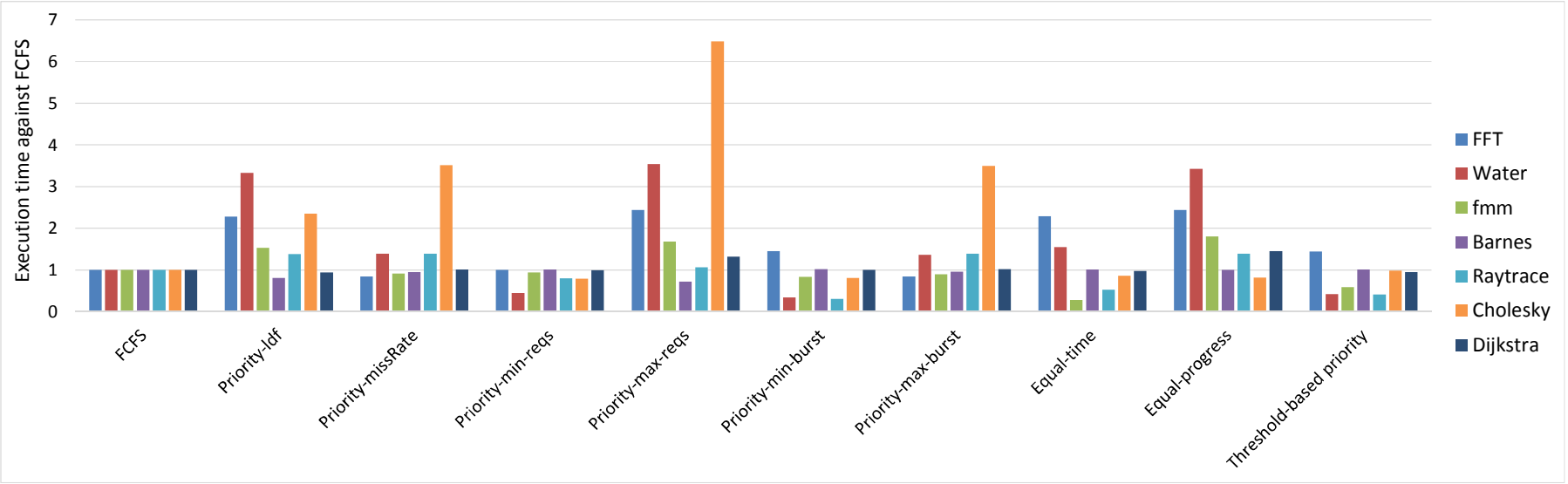
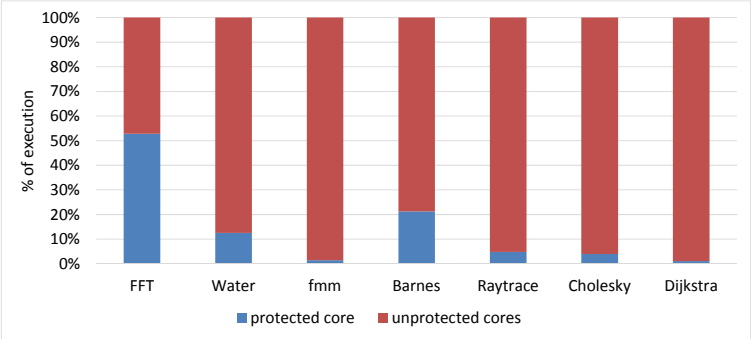


Figure 6.12. Execution time differences of applications against FCFS.

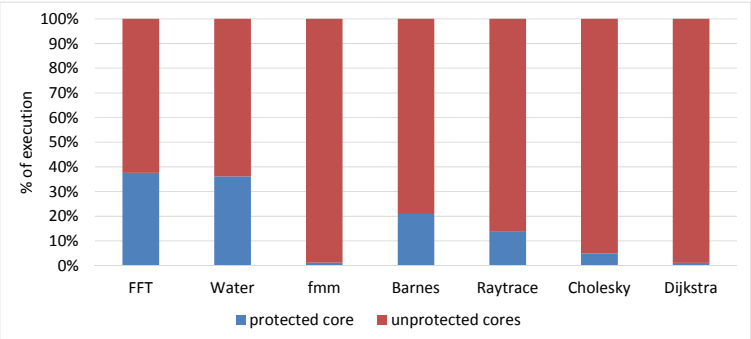
In this figure, the FFT application spends approximately half of the execution time on the protected core, while the same application passes 17.6% of its execution time on the protected core with the equal-progress method shown in Figure 6.13(c). Here, the numerator part in the execution time ratio is kept unchanged where the denominator part is increased for this method. On the other hand, in the priority-min-burst method, which has the best performance, it is seen that the percentage of the execution times of the applications on the protected core is increased (see Figure 6.13(b)).

Figure 6.14 shows the percentage of application requests served over time under a set of selected scheduling methods based on their execution behavior. Since the total number of the requests for each application might be different, the results are presented by the proportion of the requests served over time. According to the results of the FCFS method shown in Figure 6.14(a), when 10% of the time is completed, all requests of the applications, such as FFT, Water-nsquared, and Cholesky, can be served. On the other hand, when 50% of the time is completed, all requests of six applications out of seven can be served except the Barnes application. For the case of the priority-min-burst method, when 30% of the time is completed, all of the requests belonging to five of the seven applications can be served (see Figure 6.14(b)). When 40% of the time is passed, only the requests of the Barnes application, which has the highest burst time on the protected core, remain in the system.

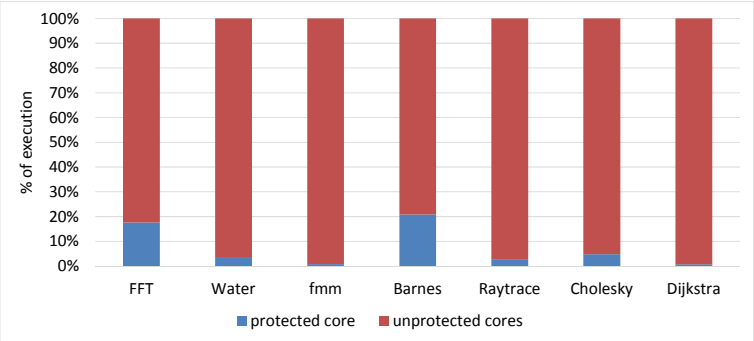
On the other hand, the results of the priority-max-burst method show a different behavior than the other methods (see Figure 6.14(c)). In this method, Barnes and FFT applications with the highest burst time on the protected core are prioritized over the remaining applications. When 50% of the time is passed, only these two applications could benefit from the protected core. When 90% of the time is completed, all requests of these applications can be served; however, the requests of the remaining applications can be served completely in the last interval (between 90% and 100% of the time). According to the results of the equal-progress method shown in Figure 6.14(d), the Cholesky application, which sends only one request throughout its lifetime, can be served in the first 10% time interval.



(a) Execution time distributions of the applications with FCFS technique.



(b) Execution time distributions of the applications with priority-min-burst technique.



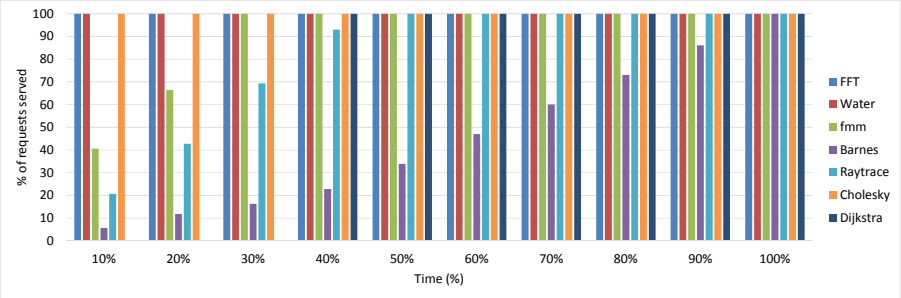
(c) Execution time distributions of the applications with equal-progress technique.

Figure 6.13. Execution time distributions of the applications in 7-application workload on the protected and unprotected cores with different scheduling techniques.

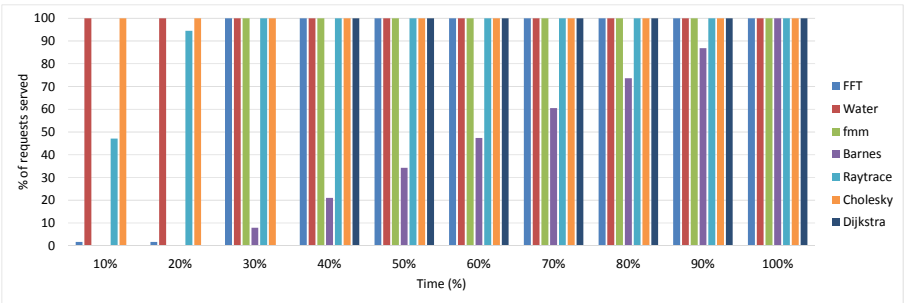
In this method, the goal is to enable equal progress of each application; hence, the requests of the applications can be served with approximately similar rates in each time interval.

Figure 6.15 shows the cumulative waiting times of the applications in the queue over time for the selected scheduling methods. For the FCFS method, the Barnes application has the largest values (see Figure 6.15(a)). The waiting times of the three applications become fixed when 20% of the time is passed, since their requests are served within this time interval. For the priority-min-burst method, when 30% of the time is completed, the waiting times of the five applications do not change. Only the waiting time of the Barnes application with the highest burst time keeps increasing since it is placed at the end of the queue.

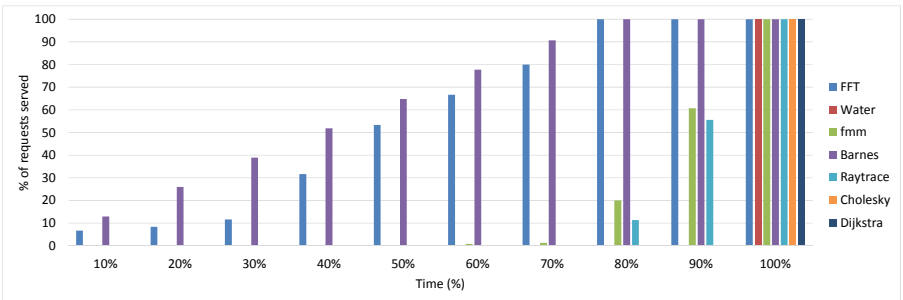
The priority-max-burst shows a different behavior than the others (see 6.15(c)). In this case, the application with the largest waiting time is not the Barnes application; it has less waiting time than the three applications instead, which are Raytrace, Water-nsquared, and fmm. Furthermore, the waiting time of the Barnes application has the lowest value compared with the other methods. On the other hand, the applications such as Cholesky and Dijkstra have noticeably higher waiting times under this scheduling method. According to the results of the equal-progress method shown in Figure 6.15(d), the waiting times for the majority of the applications show similar tendencies except for the Cholesky and Dijkstra applications, which have a smaller number of requests. This is an expected result according to the working principle of this scheduling method. It should be noted that the figures presented in this section contain only the subset of scheduling methods, which shows different execution behavior than the others.



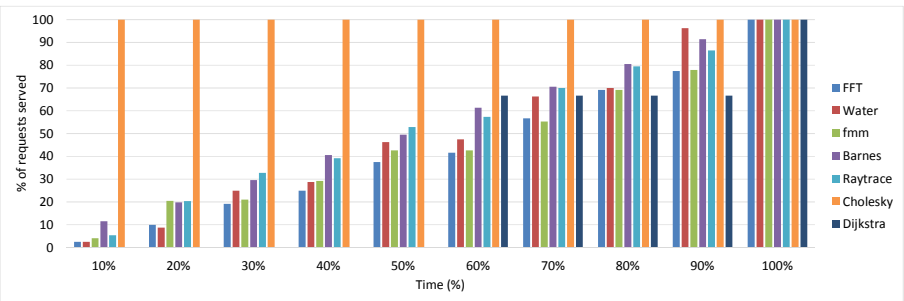
(a) FCFS technique.



(b) Priority-min-burst technique.

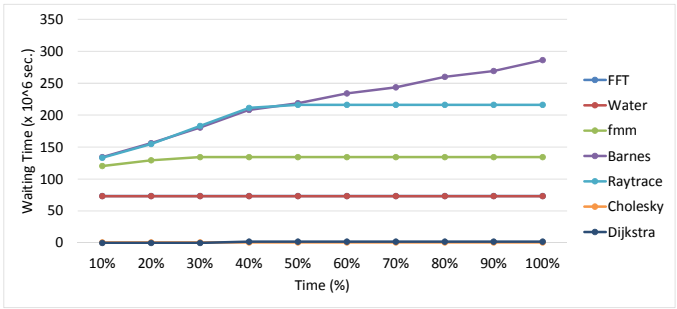


(c) Priority-max-burst technique.

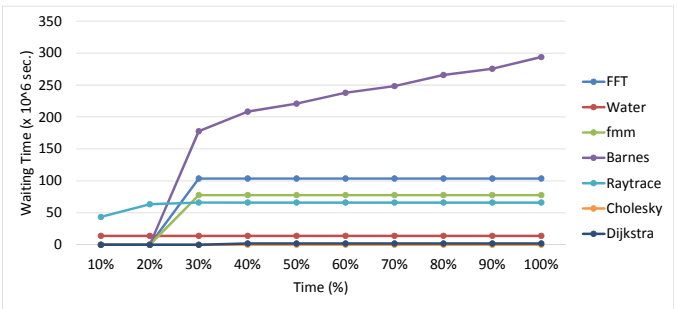


(d) Equal-progress technique.

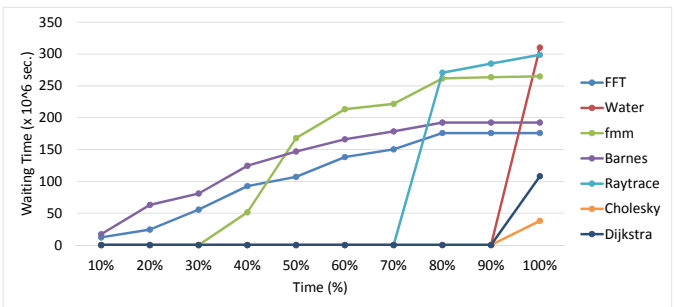
Figure 6.14. Percentage of application requests served over time for the 7-application workload under different scheduling techniques.



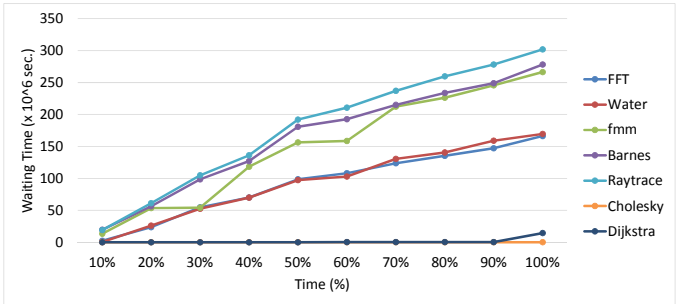
(a) FCFS technique.



(b) Priority-min-burst technique.



(c) Priority-max-burst technique.



(d) Equal-progress technique.

Figure 6.15. Waiting time of the applications for the 7-application workload in the queue with different scheduling techniques.

6.5.2.6. Discussions. The computational experiments presented in this section provides us the following key insights for various scheduling techniques implemented in our asymmetrically reliable caches:

- The priority-min-burst method, which prioritizes the applications with low total burst time on the protected core(s), offers the best results in terms of both performance and fairness for three of the four workloads.
- Likewise, the priority-min-reqs method, which presents better performance and fairness results for three of the four workloads, appears to be preferable when only the number of requests, not the total burst time, of the applications is known.
- The equal-time method presents better performance for three of the four workloads and it has better fairness results for all workloads. This scheduling method can be preferable when there is no prior knowledge on the applications or when the execution of the applications takes too long to obtain such prior knowledge.
- The results of the threshold-based priority are very close to the priority-min-burst method since it starts with the priority values offered in the priority-min-burst method by considering the last scheduled application.
- When we increase the number of protected cores, the differences among the scheduling methods disappear for the workloads with a small number of requests. Since multiple protected cores can take the requests in parallel, there are no waiting requests in the queue; hence, reordering the requests does not improve the performance of the system for the small-size workloads. On the other hand, for the workloads with a high number of requests, we still observe differences among the scheduling techniques.

It should be noted that the performance and fairness results of the scheduling methods applied in this work may vary according to the number of the applications used in the workload, the total resource usage of the applications, and the sending time of the requests at runtime.

6.6. Summary

In this section of our work, we provide a set of different scheduling algorithms with various characteristics for asymmetrically reliable caches. Managing queue structure and assigning application threads to the protected core(s) might be handled by the FCFS-based scheduling policy as the first attempt. We observe that the FCFS-based technique leads to low performance and low fairness in some workloads. Based on this observation, we utilize different types of scheduling techniques and evaluate them in terms of system performance and fairness relative to the FCFS algorithm, the baseline approach for the experimental comparisons. Six different priority-based techniques (priority-ldf, priority-missRate, priority-min-reqs, priority-max-reqs, priority-min-burst, and priority-max-burst) are provided where they differ in prioritization of applications with respect to execution order, cache usage, number of requests, or total burst time on the protected core(s). Additionally, equal-time scheduling technique is proposed which targets to equalize the amount of time spent on the protected core(s) for each application. Moreover, we propose equal-progress scheduling technique for equalizing the progress of the requests for each application, and threshold-based priority method for utilizing a priority-based technique by considering the last scheduled application.

In order to validate the proposed scheduling techniques, we utilize a set of multi-threaded multi-application workloads. Our evaluations with various workloads demonstrate that the priority-min-burst method provides better performance and fairness results than the FCFS algorithm for the majority of the workloads with the highest magnitude of improvement. The equal-time method presents better fairness results for all workloads and better performance results for most of the workloads. When we have preliminary information about the applications, the priority-min-burst method is the best alternative. However, in case of lack of priori information, the equal-time method might be preferred for its high fairness results.

7. CONCLUSIONS AND FUTURE WORK

The main focus of this thesis is to provide enhanced protection mechanism for reliability-based critical regions of applications using asymmetrically reliable cores. The cores have homogeneous speed and performance, but they differ in fault tolerance capabilities. There are at least one high reliability core which has ECC-protected L1 cache, and several low reliability cores having no protection. The software threads that execute critical regions are mapped to the high reliability core(s) dynamically during the execution. Therefore, sufficient reliability can be guaranteed using minimal fault tolerant hardware considering performance, power consumption and cost constraints.

In the first part, the main mechanics of our proposed approach are implemented and evaluated with selected applications. Reliability-based critical sections are determined based on critical data usage. An application input is analyzed statically a priori, and a portion of the input is determined as critical data specifically for an application. The code fragments that access critical data are determined as the critical regions that need to be protected. A hardware platform with one high reliability and three low reliability cores is utilized. Experimental studies validates that the partially safe cache configuration significantly outperforms the unsafe configuration with respect to reliability (given in failure rate), and the safe configuration with respect to performance (given in execution time) and energy (given in both cache energy and overall system energy) [6].

In the following part, we enlarge our proposed system with asymmetrically caches by utilizing two types of protected cores (high reliability core and medium-level reliability core). While the high reliability core has ECC-protected L1 cache, the middle-level reliability core has parity check on its L1 cache. Reliability-based critical sections are determined based on critical data usage having three classes: critical, semi-critical, and non-critical data. Application threads can use high, medium-level, or low reliability cores based on the criticality of the application's data.

Our detailed experimental work shows that both of the partially safe configurations (the system with high reliability and low reliability cores, and the system with high, middle-level, and low reliability cores) reduce the failure rate significantly compared to the unsafe configuration with much better performance and energy consumption results than the safe configuration [7].

In the second part, reliability-based critical code regions are determined based on user annotations. Selected applications are profiled and the functions that cover 90% of total execution time are used as the critical code regions. The applications are executed on the proposed system with asymmetrically reliable caches using different numbers of protected cores. Experimental studies with different function characteristics show that there is not an ideal number of protected cores for each application, but an ideal number of protected cores for each function. In this sense, in a system with a limited number of protected cores, the user can determine the number of protected cores used by considering the reliability-based critical regions. Further, our partially safe configuration takes the advantage of protecting only reliability-based critical code regions of the applications and offers significant performance and power savings compared to the safe configuration and lower failure rates compared to the unsafe configuration [8].

In the following part, the reliability-based critical code regions determined as high-priority functions of each application. The high-priority functions are identified by calculating the function vulnerability and criticality metrics based on the function execution times and call graphs, statically. The high-priority functions are protected more conservatively than the other functions by utilizing different partially safe configurations. The experimental results with a diverse set of applications confirm that our partially safe configuration offers close performance and reliability results to the safe configuration with lower power consumption and cost [9].

As part of this thesis, a set of different scheduling algorithms with various characteristics are proposed for asymmetrically reliable caches. In previous sections, assigning application threads to the protected core(s) is handled by the FCFS-based scheduling policy.

However, the FCFS-based technique may lead to low performance and low fairness in some workloads. Based on this observation, different types of scheduling techniques are implemented and evaluated in terms of system performance and fairness relative to the FCFS algorithm, the baseline approach for the experimental comparisons. A total of nine different techniques which contain six static priority-based techniques (priority-ldf, priority-missRate, priority-min-reqs, priority-max-reqs, priority-min-burst, and priority-max-burst), an equal-time scheduling, an equal-progress scheduling, and a threshold-based priority techniques are provided where they differ in prioritization of the applications. The priority-min-burst method provides better performance and fairness results than the FCFS algorithm for the majority of workloads with the highest improvement level. The equal-time method provides better fairness results for all workloads and better performance results for most of the workloads. When we have prior information about the applications, the priority-min-burst method is the best alternative. However, equal-time may be preferred for lack of prior knowledge and high fairness results [10].

As a future work, we can determine reliability-based critical sections dynamically based on thread data dependencies. By using a metric such as Thread Vulnerability Factor (TVF) [86, 87], we can isolate reliability-based critical code regions within the application, dynamically. TVF metric measures the vulnerability of a thread in multi-threaded applications to the possible transient faults by considering thread dependencies caused from data sharing. We can combine our approach with TVF and schedule the code regions that have the highest TVF values to the high reliability core(s). In such a case, the workload should be executed twice. In the first run, thread data dependencies will be analyzed and critical code regions will be determined dynamically for applications. In the second run, these critical sections will be protected from the possible soft errors by using asymmetrically reliable cores.

Another direction for future work is to provide fault tolerance methods in software level using redundancy methods rather than cache-oriented hardware approaches. The reliability-based critical code regions can be duplicated and executed on different processors as an example of space redundancy [88] to decrease thread migration costs.

In this case, the inputs of two threads should be copied on different cores, and the outputs generated by two cores are compared to detect error occurrence. On the other hand, we can execute critical code regions at two different times to provide time redundancy [89]. In this case, we perform the same operation at different times and compares the results to detect the error. These methods does not require any additional hardware addition or modification; hence, their costs will be low.

Another possible future research may be developing more sophisticated scheduling techniques. In this sense, we may focus on rate-based scheduling techniques [90] in which we can give priority to requesting threads and high reliability cores based on their service rates. We can analyze how well the applications use the high reliability cores and give priority to the applications, accordingly. On the other hand, we can give function call graph file to the scheduler to make more appropriate scheduling decisions by observing the next candidate functions for the high reliability core. As another approach, we can propose a scheduling method which takes into consideration thread waiting times in the queue; therefore, it may give priority to the applications based on their waiting times to decrease queuing cost of our approach.

REFERENCES

1. Asadi, G. H., V. S. Mehdi, B. Tahoori and D. Kaeli, "Balancing Performance and Reliability in the Memory Hierarchy", *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pp. 269–279, March 2005.

2. Rehman, S., M. Shafique and J. Henkel, *Reliable Software for Unreliable Hardware: A Cross Layer Perspective*, Springer Publishing Company, Incorporated, 1st edn., 2016.

3. Shivakumar, P., M. Kistler, S. Keckler, D. Burger and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic", *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pp. 389–398, 2002.

4. Ebrahimi, M., A. Evans, M. B. Tahoori, E. Costenaro, D. Alexandrescu, V. Chandra and R. Seyyedi, "Comprehensive Analysis of Sequential and Combinational Soft Errors in an Embedded Processor", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 34, No. 10, pp. 1586–1599, October 2015.

5. Naseer, R., Y. Boulghassoul, J. Draper, S. DasGupta and A. Witulski, "Critical Charge Characterization for Soft Error Rate Modeling in 90nm SRAM", *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pp. 1879–1882, May 2007.

6. Arslan, S., H. Topcuoglu, M. Kandemir and O. Tosun, "Performance and Energy Efficient Asymmetrically Reliable Caches for Multicore Architectures", *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 1025–1032, May 2015.

7. Arslan, S., H. R. Topcuoglu, M. T. Kandemir and O. Tosun, "Asymmetrically Reliable Caches for Multicore Architectures Under Performance and Energy Constraints", *Cluster Computing*, Vol. 19, No. 4, pp. 1819–1833, 2016.

8. Arslan, S., H. R. Topcuoglu, M. T. Kandemir and O. Tosun, *Protecting Code Regions on Asymmetrically Reliable Caches*, pp. 375–387, Springer International Publishing, Cham, 2016.

9. Arslan, S., H. R. Topcuoglu, M. T. Kandemir and O. Tosun, "A selective protection scheme of applications using asymmetrically reliable caches", *Journal of Systems Architecture*, Vol. 75, pp. 133 – 144, 2017.

10. Arslan, S., H. R. Topcuoglu, M. T. Kandemir and O. Tosun, "Scheduling Opportunities for Asymmetrically Reliable Caches", *In preparation*.

11. Meaney, P., L. Lastras-Montano, V. Papazova, E. Stephens, J. Johnson, L. Alves, J. O'Connor and W. Clarke, "IBM zEnterprise redundant array of independent memory subsystem", *IBM Journal of Research and Development*, Vol. 56, No. 1.2, pp. 4:1–4:11, January 2012.

12. Koren, I. and S. Y. H. Su, "Reliability Analysis of N-Modular Redundancy Systems with Intermittent and Permanent Faults", *IEEE Transactions on Computers*, Vol. C-28, No. 7, pp. 514–520, July 1979.

13. Alameldeen, A. R., I. Wagner, Z. Chishti, W. Wu, C. Wilkerson and S.-L. Lu, "Energy-efficient Cache Design Using Variable-strength Error-correcting Codes", *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, ACM, New York, NY, USA, 2011.

14. Wilkerson, C., A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar and S.-l. Lu, "Reducing Cache Power with Low-cost, Multi-bit Error-correcting Codes", *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 83–93, ACM, New York, NY, USA, 2010.

15. Yoon, D. H. and M. Erez, "Virtualized ECC: Flexible Reliability in Main Memory", *Micro, IEEE*, Vol. 31, No. 1, pp. 11–19, January 2011.

16. Yoon, D. H. and M. Erez, "Memory Mapped ECC: Low-cost Error Protection for Last Level Caches", *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 116–127, ACM, New York, NY, USA, 2009.

17. Zhao, H., A. Sharifi, S. Srikantaiah and M. Kandemir, "Feedback control based cache reliability enhancement for emerging multicores", *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pp. 56–62, November 2011.

18. Lee, K., A. Shrivastava, I. Issenin, N. Dutt and N. Venkatasubramanian, "Partially Protected Caches to Reduce Failures Due to Soft Errors in Multimedia Applications", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol. 17, No. 9, pp. 1343–1347, September 2009.

19. Borchert, C., H. Schirmeier and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures", *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pp. 1–12, June 2013.

20. Xu, X. and M.-L. Li, "Understanding Soft Error Propagation Using Efficient Vulnerability-driven Fault Injection", *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pp. 1–12, IEEE Computer Society, Washington, DC, USA, 2012.

21. Eltawil, A. A., M. Engel, B. Geuskens, A. K. Djahromi, F. J. Kurdahi, P. Marwedel, S. Niar and M. A. Saghir, "A Survey of Cross-layer Power-reliability Tradeoffs in Multi and Many Core Systems-on-chip", *Microprocess. Microsyst.*, Vol. 37, No. 8, pp. 760–771, 2013.

22. de Kruijf, M., S. Nomura and K. Sankaralingam, "Relax: An Architectural Frame-

work for Software Recovery of Hardware Faults", *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 497–508, ACM, New York, NY, USA, 2010.

23. Sampson, A., W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation", *SIGPLAN Not.*, Vol. 46, No. 6, pp. 164–174, June 2011.

24. Carbin, M., S. Misailovic and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware", *SIGPLAN Not.*, Vol. 48, No. 10, pp. 33–52, October 2013.

25. Misailovic, S., M. Carbin, S. Achour, Z. Qi and M. C. Rinard, "Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels", *SIGPLAN Not.*, Vol. 49, No. 10, pp. 309–328, October 2014.

26. Leem, L., H. Cho, J. Bau, Q. Jacobson and S. Mitra, "ERSA: Error Resilient System Architecture for probabilistic applications", *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 1560–1565, March 2010.

27. Rehman, S., F. Kriebel, M. Shafique and J. Henkel, "Compiler-driven dynamic reliability management for on-chip systems under variabilities", *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–4, March 2014.

28. Rehman, S., F. Kriebel, M. Shafique and J. Henkel, "Reliability-Driven Software Transformations for Unreliable Hardware", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Vol. 33, No. 11, pp. 1597–1610, November 2014.

29. Yetim, Y., S. Malik and M. Martonosi, "CommGuard: Mitigating Communication Errors in Error-Prone Parallel Execution", *SIGARCH Comput. Archit. News*, Vol. 43, No. 1, pp. 311–323, March 2015.

30. Huang, J., S. Barner, A. Raabe, C. Buckl and A. Knoll, "A Framework for Reliability-aware Embedded System Design on Multiprocessor Platforms", *Microprocess. Microsyst.*, Vol. 38, No. 6, pp. 539–551, August 2014.

31. Suleman, M. A., O. Mutlu, M. K. Qureshi and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures", *SIGARCH Comput. Archit. News*, Vol. 37, No. 1, pp. 253–264, March 2009.

32. Ungsunan, P., C. Lin, Y. Gai and X. Kong, "Improving Multi-Core System Dependability with Asymmetrically Reliable Cores", *Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09. International Conference on*, pp. 1252–1257, March 2009.

33. Luo, Y., S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory", *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pp. 467–478, June 2014.

34. Burtscher, M., B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke and J. Browne, "PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications", *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pp. 1–11, IEEE Computer Society, Washington, DC, USA, 2010.

35. Subotic, V., J. C. Sancho, J. Labarta and M. Valero, "Identifying Critical Code Sections in Dataflow Programming Models", *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 29–37, February 2013.

36. Carbin, M. and M. C. Rinard, "Automatically Identifying Critical Input Regions and Code in Applications", *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA, pp. 37–48, ACM, New York, NY, USA,

2010.

37. Duque, L. A. R., J. M. M. Diaz and C. Yang, "Improving MPSoC Reliability Through Adapting Runtime Task Schedule Based on Time-correlated Fault Behavior", *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pp. 818–823, EDA Consortium, San Jose, CA, USA, 2015.

38. Duque, L. A. R. and C. Yang, "Guiding fault-driven adaption in multicore systems through a reliability-aware static task schedule", *The 20th Asia and South Pacific Design Automation Conference*, pp. 612–617, January 2015.

39. Chantem, T., Y. Xiang, X. S. Hu and R. P. Dick, "Enhancing multicore reliability through wear compensation in online assignment and scheduling", *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 1373–1378, March 2013.

40. Coskun, A. K., R. Strong, D. M. Tullsen and T. Simunic Rosing, "Evaluating the Impact of Job Scheduling and Power Management on Processor Lifetime for Chip Multiprocessors", *SIGMETRICS Perform. Eval. Rev.*, Vol. 37, No. 1, pp. 169–180, June 2009.

41. Wang, Y., A. Nicolau, R. Cammarota and A. V. Veidenbaum, "A fault tolerant self-scheduling scheme for parallel loops on shared memory systems", *High Performance Computing (HiPC), 2012 19th International Conference on*, pp. 1–10, December 2012.

42. Lin, S. and G. Manimaran, "A Feedback-based Adaptive Algorithm for Combined Scheduling with Fault-tolerance in Real-time Systems", *Proceedings of the 11th International Conference on High Performance Computing*, HiPC'04, pp. 101–110, Springer-Verlag, Berlin, Heidelberg, 2004.

43. Li, D. and J. Wu, "Minimizing Energy Consumption for Frame-Based Tasks on

Heterogeneous Multiprocessor Platforms", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 26, No. 3, pp. 810–823, March 2015.

44. Tang, X., K. Li, R. Li and B. Veeravalli, "Reliability-aware Scheduling Strategy for Heterogeneous Distributed Computing Systems", *J. Parallel Distrib. Comput.*, Vol. 70, No. 9, pp. 941–952, September 2010.

45. Tang, X. and W. Tan, "Energy-Efficient Reliability-Aware Scheduling Algorithm on Heterogeneous Systems", *Sci. Program.*, Vol. 2016, pp. 14–, March 2016.

46. Tang, X., K. Li and G. Liao, "An effective reliability-driven technique of allocating tasks on heterogeneous cluster systems", *Cluster Computing*, Vol. 17, No. 4, pp. 1413–1425, 2014.

47. Xiaoyong, T., K. Li, Z. Zeng and B. Veeravalli, "A Novel Security-Driven Scheduling Algorithm for Precedence-Constrained Tasks in Heterogeneous Distributed Systems", *IEEE Transactions on Computers*, Vol. 60, No. 7, pp. 1017–1029, July 2011.

48. Wang, Y., K. Li, H. Chen, L. He and K. Li, "Energy-Aware Data Allocation and Task Scheduling on Heterogeneous Multiprocessor Systems With Time Constraints", *IEEE Transactions on Emerging Topics in Computing*, Vol. 2, No. 2, pp. 134–148, June 2014.

49. Mei, J., K. Li and K. Li, "Energy-aware Task Scheduling in Heterogeneous Computing Environments", *Cluster Computing*, Vol. 17, No. 2, pp. 537–550, June 2014.

50. Topcuouglu, H., S. Hariri and M. Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing", *IEEE Trans. Parallel Distrib. Syst.*, Vol. 13, No. 3, pp. 260–274, March 2002.

51. Li, D. and J. Wu, "Energy-Aware Scheduling for Frame-Based Tasks on Heterogeneous Multiprocessor Platforms", *2012 41st International Conference on Parallel Processing*, pp. 430–439, September 2012.

52. Wang, J., N. Abu-Ghazaleh and D. Ponomarev, "Controlled Contention: Balancing Contention and Reservation in Multicore Application Scheduling", *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 946–955, May 2015.

53. Craeynest, K. V., S. Akram, W. Heirman, A. Jaleel and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores", *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 177–187, September 2013.

54. Huh, S., J. Yoo, M. Kim and S. Hong, "Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm", *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pp. 606–614, IEEE Computer Society, Washington, DC, USA, 2012.

55. Kim, Y., D. Han, O. Mutlu and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers", *The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, January 2010.

56. Vandierendonck, H. and A. Seznec, "Fairness Metrics for Multi-Threaded Processors", *IEEE Computer Architecture Letters*, Vol. 10, No. 1, pp. 4–7, January 2011.

57. Das, R., O. Mutlu, T. Moscibroda and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks", *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 280–291, December 2009.

58. Subramanian, L., D. Lee, V. Seshadri, H. Rastogi and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost", *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 8–15, October 2014.

59. Subramanian, L., D. Lee, V. Seshadri, H. Rastogi and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, No. 10, pp. 3071–3087, October 2016.

60. Usui, H., L. Subramanian, K. K.-W. Chang and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators", *ACM Trans. Archit. Code Optim.*, Vol. 12, No. 4, pp. 65:1–65:28, January 2016.

61. Papazachos, Z. C. and H. D. Karatza, "Gang Scheduling in Multi-core Clusters Implementing Migrations", *Future Gener. Comput. Syst.*, Vol. 27, No. 8, pp. 1153–1165, October 2011.

62. Manickam, V. and A. Aravind, "A Fair and Efficient Gang Scheduling Algorithm for Multicore Processors", K. R. Venugopal and L. M. Patnaik (Editors), *Wireless Networks and Computational Intelligence: 6th International Conference on Information Processing, ICIP 2012*, pp. 467–476, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

63. Stavrinides, G. L. and H. D. Karatza, "Fault-tolerant Gang Scheduling in Distributed Real-time Systems Utilizing Imprecise Computations", *SIMULATION*, Vol. 85, No. 8, pp. 525–536, 2009.

64. Wiseman, Y. and D. G. Feitelson, "Paired Gang Scheduling", *IEEE Trans. Parallel Distrib. Syst.*, Vol. 14, No. 6, pp. 581–592, June 2003.

65. Zhang, Y., H. Franke, J. Moreira and A. Sivasubramaniam, "An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration", *IEEE Trans. Parallel Distrib. Syst.*, Vol. 14, No. 3, pp. 236–247, March 2003.

66. Mi, N., G. Casale and E. Smirni, "Scheduling for performance and availability in systems with temporal dependent workloads", *2008 IEEE International Conference*

on Dependable Systems and Networks With FTCS and DCC (DSN), pp. 336–345, June 2008.

67. Casale, G., N. Mi and E. Smirni, "CWS: A Model-driven Scheduling Policy for Correlated Workloads", *SIGMETRICS Perform. Eval. Rev.*, Vol. 38, No. 1, pp. 251–262, June 2010.

68. Iqbal, S., Y. Liang and H. Grahn, "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems", *Computer Architecture Letters*, Vol. 9, No. 2, pp. 45–48, February 2010.

69. González, A., C. Aliagas and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality", *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pp. 338–347, ACM, New York, NY, USA, 1995.

70. Warren, H. S., *Hacker's Delight*, Addison-Wesley Professional, 2nd edn., 2012.

71. Lee, K., A. Shrivastava, I. Issenin, N. Dutt and N. Venkatasubramanian, "Mitigating Soft Error Failures for Multimedia Applications by Selective Data Protection", *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pp. 411–420, ACM, New York, NY, USA, 2006.

72. Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The Gem5 Simulator", *SIGARCH Comput. Archit. News*, Vol. 39, No. 2, pp. 1–7, 2011.

73. Muralimanohar, N., R. Balasubramonian and N. P. Jouppi, "Architecting Efficient Interconnects for Large Caches with CACTI 6.0", *IEEE Micro*, Vol. 28, No. 1, pp. 69–79, January 2008.

74. Woo, S., M. Ohara, E. Torrie, J. Singh and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations", *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pp. 24–36, June 1995.

75. Woo, S. C., J. P. Singh and J. L. Hennessy, "The Performance Advantages of Integrating Block Data Transfer in Cache-coherent Multiprocessors", *SIGOPS Oper. Syst. Rev.*, Vol. 28, No. 5, pp. 219–229, 1994.

76. Cai, Y., M. Schmitz, A. Ejlali, B. Al-Hashimi and S. Reddy, "Cache size selection for performance, energy and reliability of time-constrained systems", *Design Automation, 2006. Asia and South Pacific Conference on*, pp. 6 pp.–, January 2006.

77. Leveugle, R., A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence", *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 502–506, April 2009.

78. Li, S., J. Ahn, R.D.Strong, J. Brockman, D. Tullsen and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures", *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, 2009.

79. Brooks, D., V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-level Power Analysis and Optimizations", *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pp. 83–94, ACM, New York, NY, USA, 2000.

80. Bienia, C., S. Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 72–81, ACM, New York, NY, USA, 2008.

81. Gupta, A. and V. Kumar, "The Scalability of FFT on Parallel Computers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, pp. 922–932, 1993.

82. Good, I. J., *Introduction to Cooley and Tukey (1965) An Algorithm for the Machine Calculation of Complex Fourier Series*, pp. 201–216, Springer New York, New York, NY, 1997.

83. Nemirovsky, M. and D. M. Tullsen, *Multithreading Architecture*, Morgan & Claypool Publishers, 1st edn., 2013.

84. Chang, J. and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors", *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pp. 242–252, ACM, New York, NY, USA, 2007.

85. Jain, R., D. Chiu and W. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared systems*, Tech. rep., Digital Equipment Corporation, DEC-TR-301, 1984.

86. Oz, I., H. Topcuoglu, M. Kandemir and O. Tosun, "Quantifying Thread Vulnerability for Multicore Architectures", *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2011.

87. Oz, I., H. Topcuoglu, M. Kandemir and O. Tosun, "Thread Vulnerability in Parallel Applications", *Journal of Parallel and Distributed Computing (JPDC)*, Vol. 72, pp. 1171–1185, 2012.

88. Reinhardt, S. K. and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading", *SIGARCH Computer Architecture News*, Vol. 28, No. 2, pp. 25–36, May 2000.

89. Goloubeva, O., M. Rebaudengo, M. S. Reorda and M. Violante, *Software-Implemented Hardware Fault Tolerance*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

90. Mello, A. and N. Calazans, "Rate-based scheduling policy for QoS flows in networks on chip", *2007 IFIP International Conference on Very Large Scale Integration*, pp. 140–145, October 2007.