ReSense: A Unified Framework for Improving Performance and Reliability in Multicore Architectures

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Tanima Dey

August 2014

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Tanima Dey

Tanima Dey

This dissertation has been read and approved by the Examining Committee:

Mary Lou Soffa

Mary Lou Soffa, Advisor

Jack W. Davidson

Jack W. Davidson, Advisor

Sudhanva Gurumurthi

Sudhanva Gurumurthi, Committee Chair

Mary Jane Irwin

Mary Jane Irwin, Pennsylvania State University

John Lach

John Lach, Minor Representative

Accepted for the School of Engineering and Applied Science:

James H. Aylor, Dean, School of Engineering and Applied Science

August 2014

Abstract

Chip-multiprocessors (CMPs) have become ubiquitous in modern computing and the mainstream architecture for various platforms, including laptops, desktops, and large server machines. As technology scaling continues and more transistors are accommodated on the chip, the number of cores on CMPs is growing, and multi-core machines are scaling up to many-core machines. With this multi-core scaling, two major problems arise: shared-resource contention and soft errors or transient faults. Shared-resource contention can degrade an application's performance significantly, and soft errors increase the probability of incorrect application execution and the production of visible errors. To realize the full potential of multi- and many-core platforms, it is critical to ensure that applications in a workload not only execute efficiently and fast, but also correctly.

In this dissertation, we develop a novel, general, and unified framework, ReSense, to address several challenges on multicore architectures including performance optimization, reliability improvement, power and thermal management. The framework includes five components: a general characterization methodology, a characterization metric, a sensitivity score, a thread mapping algorithm, and a run-time system. An instance of the framework is applied in two phases: characterization and mapping. The characterization phase utilizes the general characterization methodology and characterization metric to identify application characteristics without considering any co-runner(s). It generates a resource-sensitivity score for each application in a workload. In the mapping phase, the run-time system uses a thread-mapping algorithm and the sensitivity scores of the applications in a workload to

determine the thread-mappings that optimize the objective function of the targeted problem.

To demonstrate the utility and effectiveness of ReSense, we use it to address the problems of shared-resource contention and soft errors for multi-threaded applications. For the resource contention problem, the characterization methodology determines how a multi-threaded application's performance is affected as it shares a resource in the memory hierarchy. A sensitivity score based on resource contention is produced for each application in a workload. The run-time system uses the resource-contention sensitivity scores and a thread-mapping algorithm to allocate threads from a workload to core to mitigate shared-resource contention, thus improving response time and throughput.

For the soft error problem, the characterization methodology determines how a multi-threaded application's vulnerability to soft errors in shared caches is affected by its resource occupancy duration. A sensitivity score based on cache occupancy is produced for each application in a workload. The run-time system uses the cache-occupancy sensitivity scores and a thread-mapping algorithm to allocate workload threads to cores to reduce the occupancy in the shared caches, thus reducing cache vulnerability.

Both minimizing an application's vulnerability to soft errors and maintaining application performance are critical. The thread-mapping algorithm that ensures better reliability may not ensure better performance. To address this problem, we develop an integrated instance of the framework that combines application characterizations for both contention and vulnerability to determine a trade-off between the performance and reliability improvements.

The dissertation includes a comprehensive evaluation of all three instances, which indicates that the mapping of each application in a dynamic workload according to its solo-characterization is highly effective. For the resource contention instance, response time and throughput was improved up to 30% and 47%, respectively over the native operating system. For the soft error instance, cache vulnerability was reduced up to 70% over the native operating system. The integrated instance was able to achieve various trade-offs between response time and vulnerability reductions.

Acknowledgments

This dissertation would not have been possible without the help and support from many people in my life.

First and foremost, I acknowledge my PhD advisors, Mary Lou Soffa and Jack Davidson. Over the past six years, they have taught me how to think critically, write clearly, express ideas, give good talks, and most importantly how to do good research. They have mentored me, supported me both professionally and personally, whenever I went through any tough time in my life. They have always been patient with me whenever I struggled to find my research direction and with the countless draft of the papers I wrote with them. I cannot thank them enough and am very grateful for what they did for me. They truly have been my academic parents.

I would like to acknowledge the members of my dissertation committee, Sudhanva Gurumurthi, Mary Jane Irwin, and John Lach. They have given me feedback on my dissertation proposal to make the work better. Especially, I would like to thank Sudhanva. Whenever I wanted any advice from him about the reliability work, he always made time from his busy schedule to discuss with me.

I thank my lab and research mate, Wei Wang, for his support, feedback and honest opinion about my work. I acknowledge Jason Mars, who has been a good mentor to me. In the initial years of my PhD, both Jason and Lingjia had long discussions with me about my research and shared their thoughts about how to write a good paper. I also thank Kristen Walcott and Jing Yang for their feedback on my early ideas of the dissertation proposal.

I thank the systems staff, especially Scott Ruffner and Essex Scales, for their help and assistance whenever there was any problem with the machines and servers. They have always tried to accommodate any request I had. I also thank the CS department staff for keeping me on track in terms of official paper work.

I am grateful to all my teachers in Bangladesh, from the elementary to undergraduate school. They all have contributed to my intellectual ability, starting from how to get my hand writing (which I hardly get to do anymore) better to in-depth and broad knowledge in computer science.

I acknowledge all my friends and acquaintances at UVA and in Charlottesville. I am grateful to Taniya Siddiqua for being such a great mentor and sister to me. She has inspired me in numerous ways and gave me the courage and strength to survive through the hard graduate life. My life in Charlottesville would not have been the same without so many friends here, including my "bachcha-party": Yamina, Anindya, Emi, Asif, Samee. Special thanks to Juhi for keeping me active and full of spirit during the last six months and the most stressful time of my graduate life.

I acknowledge Charlottesville for the wonderful six years. I have loved every moment I have lived here, including the beauty of the Blue Ridge Mountains, the beautiful fall, every snowstorm in the winter, and the soothing rain in the summer. This is the place where two of my dreams came true. Charlottesville will always be my second home and remain special to me.

I am very blessed to have many dear friends in my life. Some have been friends with me for almost 20 years: Shemul (Mollick), Setu, Tithi. I acknowledge Laboni and Shantonu, for listening to my endless complaints about everything that goes wrong in my life, including paper rejections. I acknowledge Chayan, Sagar, Shafi, and Nabila for keeping my spirit high whenever I felt low. I am grateful for their continuous support and unconditional friendship, which gave me enough strength in my graduate life.

I acknowledge my family. Didi and Dida have been a constant source of encouragement.

Pishi taught me how to think positively in life, which helped me during my graduate life. I acknowledge all Kaka-Kakimas, Mama-Mamis, my cousins for supporting me during the hard times in my life. I acknowledge my family here in USA, MonDidi and family, SamarDada and family, Kumar, Ann, and Kent for making a foreign country feel like home. I acknowledge my Baba for inspiring me to overcome the difficulties and frustration in life.

I acknowledge my husband, Enamul: my best friend, harshest critic, biggest admirer, partner in crime, and co-pilot in life. Ever since I met him, he has been with me through every up and down of my life. He has made me a better version of what I am today. He has encouraged me, supported me, made me believe in my abilities and myself. I am more confident when I have him by my side. I can't wait to start the next phase of our lives together.

Last but not the least, I acknowledge my mother, Purabi Dey and Dadu, Anil Kumar Dey. I can't express in words their significance in my life. When I was in Bangladesh, Dadu used to stop by my study room every time I had any exam. I really wish he were alive to see me past the PhD finish line, where he always wanted me to be. Whenever I got frustrated and felt like giving up my PhD work, the biggest force that kept me moving was the thought of Maa and all the sacrifices she made for me in her life. I dedicate my dissertation to her.

To Maa

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chip-multiprocessors (CMPs) have become ubiquitous in modern computing and the mainstream architecture for various platforms, including laptops, desktops, and large server machines. These CMP machines are very powerful and useful for solving computation-intensive problems and provide high throughput through their instruction- and thread-level parallelism capabilities.

1.1 Challenges in Multicore Systems

1.1.1 Complex and Dynamic Workloads

As technology scaling continues and more transistors are accommodated on the chip, the number of cores on CMPs is growing, and multicore machines are scaling up to many-core machines. Because of CMP scaling, the trend of application design has shifted towards multi-threaded and parallel programming. To fully exploit the available computational resources, workloads, which consist of multiple multi-threaded applications, are executed. Each application uses the underlying resources differently depending on how the application threads are mapped to the available cores. Furthermore, the way the applications use different resources has an impact on the workload's execution and overall resource management, which can lead to performance optimization, reliability improvement, power consumption, and

thermal problems on CMPs. The continuous change in the execution environment caused by *dynamic* workloads, where any number of multi-threaded applications arrive, execute, and terminate in unpredictable ways, makes these problems especially challenging and difficult to address.

As multicore machines continue to scale up, the performance and reliability challenges on these architectures are becoming more severe because of the increased criticality of shared-resource contention and soft errors [2, 3]. To realize the full potential of CMP platforms, we need to ensure that multi-threaded workloads not only execute efficiently and fast, but also correctly. The following two sections describe the resource contention and soft error problems in more detail.

1.1.2 Resource Contention

There are various resources in CMPs that are shared by several and/or all processing cores, including on-chip shared and last-level caches (LLC), front side bus (FSB), memory bus, disk, and I/O-devices. When there are multiple multi-threaded applications executing on CMPs, there is increased contention among the applications for these shared resources. Shared-resource contention is a phenomenon that occurs when an application shares any resource (e.g., last-level cache) with its co-runner[1]. Because of contention, especially for the resources in the memory hierarchy, an application's performance can degrade by more than 50% [4], and scalable performance improvement is often not achieved on multicore and many-core machines [5]. Contention for the shared resources in the memory hierarchy can also lead to inefficient resource usage [3, 6].

Because of the resource usage behaviors, contention for the shared-memory resources created by multi-threaded applications in a workload can be severe for two reasons. First, a multi-threaded application can use more than one shared resource in the memory hierarchy, which increases the degree of contentiousness. Consider the contention problem for a shared

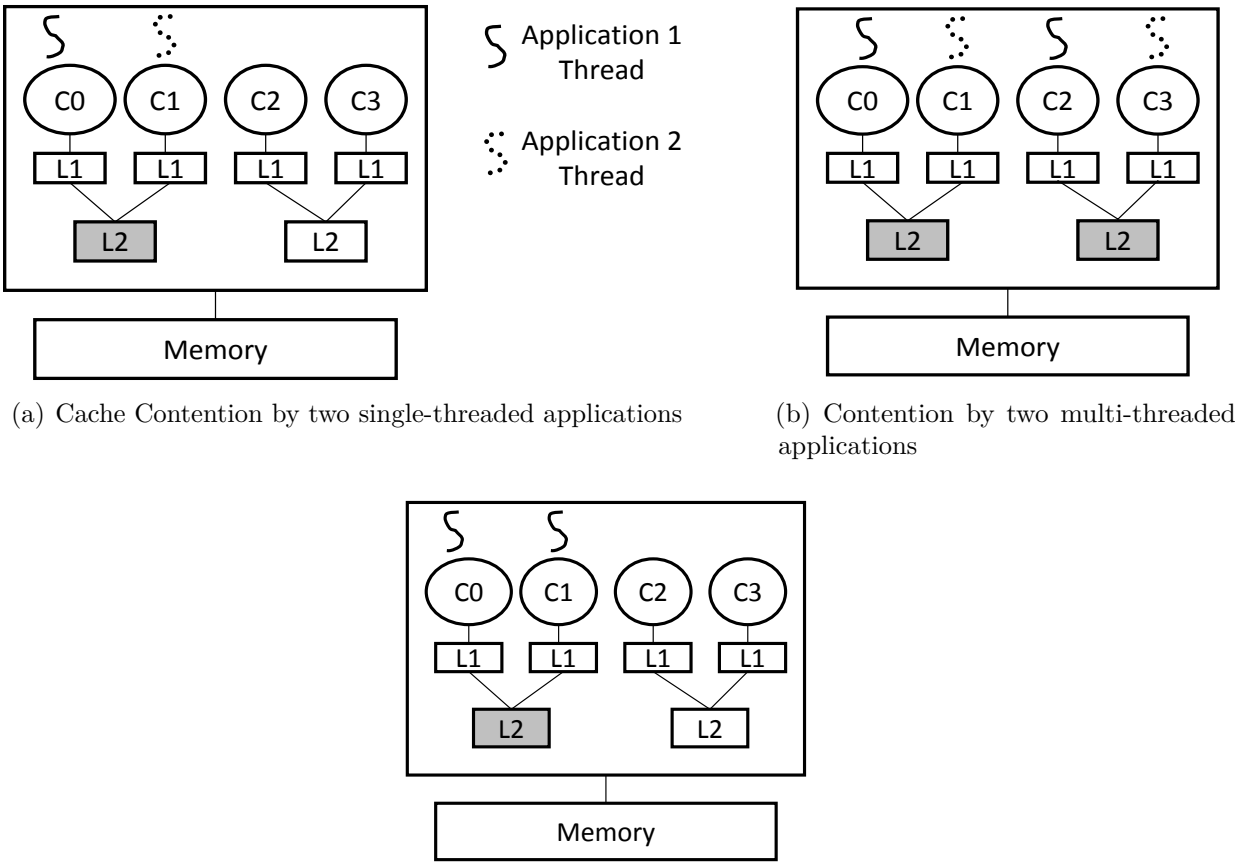
[1] A co-runner is a thread from a different application, which executes on the same or neighboring core and shares any resource.

L2-cache, which is one of the resources in the memory hierarchy. When two single-threaded applications in a workload share the same L2-cache (Figure 1.1(a)), the application that accesses L2-cache frequently can increase the cache contention and severely degrade the other application's performance. If the workload has two multi-threaded applications and the threads from one application share the same L2-cache with the other application's threads (Figure 1.1(b)), the cache contention created in two L2-caches by the cache-intensive application can also degrade the workload's overall performance. Second, the application threads can contend for the shared-memory resources among themselves even when the application runs solely. If the workload has only one multi-threaded application, the sibling threads[2] can contend for the shared L2-cache (Figure 1.1(c)) and degrade the application's performance. On the other hand, if the sibling threads share data, then sharing the same cache improves the application performance [7]. Thus a multi-threaded application's resource usage and inherent characteristics can impact the severity of resource contention and affect the workload's performance.

There has been significant research efforts to address shared-resource contention via both hardware and software techniques. Hardware techniques can provide performance improvement [8, 9]; however, they are hard to implement in practice, and existing hardware solutions might become obsolete or ineffective for emerging and new application behaviors. Several research efforts have addressed shared-resource contention via software techniques, including execution throttling, thread-mapping, and scheduling [6, 4, 10]. However, to the best of our knowledge, these efforts have not considered multi-threaded applications.

There are several differences between the characteristics of single- and multi-threaded applications as they contend for the shared resources. A single-threaded application can have one co-running thread on a neighboring core when it shares, for example, a L2-cache, as shown in Figure 1.1(a). Consequently, it contends for cache resources with that co-runner. On the other hand, a multi-threaded application can share multiple caches with multiple

[2]Sibling threads are defined as the threads that are created from the same multi-threaded application.



(a) Cache Contention by two single-threaded applications

(b) Contention by two multi-threaded applications

(c) Contention by one multi-threaded application

Figure 1.1: Shared cache contention for a workload with single- and multi-threaded applications (*CX* stands for the processor core, *L1* and *L2* stand for L1- and L2-caches, respectively.)

co-running threads, as shown in Figure 1.1(b), and contend for more than one cache with a multi-threaded co-runner. Moreover, multi-threaded applications can have contentious behavior among its own threads even when they do not have any co-runner [5], as shown in Figure 1.1(c). In addition, single-threaded applications do not have data sharing, whereas, multi-threaded applications can have data sharing [11]. In short, multi-threaded applications suffer from shared-resource contention differently than single-threaded applications, for both solo-execution and execution with co-runner(s). Existing techniques to address resource contention do not consider these differences and are thus not applicable for mitigating contention created by multi-threaded applications.

There are several challenges to mitigating contention created by simultaneously executing multi-threaded applications on CMPs. A multi-threaded application can suffer from contention created by its own sibling threads and by threads from co-running application(s). For workloads with multiple multi-threaded applications, mitigating contention depends on an application's characteristics, underlying resource topology of the platform, and importantly on the characteristics of the co-running application(s). Contention mitigation becomes more challenging for *dynamic* workloads. The continuous change in the execution environment created by the dynamic workloads with increasing and decreasing numbers of applications, arriving and terminating in unpredictable ways, makes contention mitigation even more challenging and difficult. Because of the dynamic nature of this problem and emerging workloads, contention mitigation is ideally done dynamically using software techniques based on the characteristics of the currently executing applications in the workload and the run-time environment.

1.1.3 Soft Errors

As the technology scaling continues, the size of the transistors shrinks each process generation, and the reliability of CMP platforms is becoming another critical issue [2, 12]. In particular, systems are increasingly susceptible to transient faults or soft errors. Soft errors are faults that occur randomly caused by various sources, including cosmic rays, power supply noise, and packaging defects. Soft errors do not cause permanent damage to any hardware component, but potentially cause applications to execute incorrectly and output wrong results. Soft errors affect a system's reliability with respect to micro-architectural resources, such as load-store unit (LSU), re-order buffer (ROB), instruction queue (IQ), as well as memory resources, such as caches [13]. The occurrence rate of these soft errors increases as multicore machines continue to scale up to many-core machines and the total bit count on these platforms increases, posing a significant risk for these computing platforms.

An application's resource usage can lead to vulnerable execution and have an impact on

reliability [13, 14]. If an application occupies a particular resource for a long time during its execution, then the long resource occupancy makes the application more vulnerable to soft errors caused by high energetic beam particles [15]. Such errors can change an application's execution and result in visible errors and incorrect output, reducing application reliability on the system. For example, if an application uses data from any resource very frequently and executes for a longer time than an application with a shorter execution time, then the data used by the long running application has a higher probability of being corrupted because of its susceptibility to soft errors.

Additionally, because a multi-threaded application uses multiple resources, it is more susceptible to soft errors. For example, each single-threaded application in a workload uses only one L1-cache (Figure 1.1(a)). On the other hand, the siblings threads from a multi-threaded application use two L1-caches in Figure 1.1(c). Thus, the multi-threaded application has twice the probability of being affected by soft errors in L1-caches than the single-threaded applications. Fundamentally, the resource usage characteristics of the multi-threaded applications, in terms of both duration and number of resources, can make the soft error problem on a multicore machine even more critical for a workload.

There has been much research effort addressing soft errors using both hardware and software techniques. Hardware techniques typically include redundant bits and device hardening to protect against soft errors. However, these techniques have significant power, area, cost, and latency overhead, which are not suitable for systems with large numbers of cores [16, 17]. Software techniques utilize redundant executions to ensure reliable execution. However, this mechanism has a very high performance overhead [18]. Other software techniques include compilation to reduce an application's susceptibility to soft errors [19]. Soft errors can affect an application's output during its execution, and the sensitivity to soft errors in modern systems is application dependent [13]. Therefore, dynamic and lightweight software measures that leverage application characteristics are more effective than the static compilation techniques in reducing application vulnerability to soft errors.

In general, addressing soft errors via a dynamic software technique is challenging for several reasons. First, soft errors corrupt the contents in the hardware units, which are used by the applications during their executions and can potentially result into visible errors. The software technique must be able to monitor the applications' executions continuously to detect such corrupted data usage by the applications, which can incur high overhead. Second, the software technique should reduce an application's susceptibility to soft errors so that the number of potential visible errors is minimized. Therefore, the technique should use a metric that represents and quantifies the probability of an application being affected by soft errors while it is still in execution. Determining this metric should be done accurately so that the software technique can effectively take measures to reduce the probability of soft errors affecting the application execution. Third, the effect of soft errors varies across applications and is dynamic, depending on the occupancy duration of a particular hardware resource by the applications in a workload [19]. Therefore, an effective software technique must be able to control the occupancy duration of the resources as the applications execute.

1.2 Application-driven Thread-mapping

An application's inherent characteristics of resource usage can impact both performance and reliability on multicore platforms. If an application in a workload uses a particular resource excessively, then this contentious behavior degrades the workload's performance when it shares the resources with its co-runners. If an application occupies a resource for a longer period of time, then the execution becomes highly vulnerable to soft errors, and the probability of visible errors becomes very high.

The performance and reliability challenges on a multicore platform can be addressed effectively by taking into consideration how the characteristics of the applications in a workload affect these problems of resource contention and soft errors, respectively. A characterization technique would be useful to identify the application behaviors for resource usage that are critical to address a targeted problem. The insights from such characterization and analyses

can be used to control the resource usage of the applications by intelligently mapping them on the appropriate cores such that the objective for the targeted problem is obtained.

For the resource contention problem, a contentious application in a workload can be identified by analyzing its resource usage behavior. Once the contentious application(s) is(are) identified, all the threads from a workload can be mapped such that the resource contention is mitigated and the contentious threads have minimal impact on the workload's performance, i.e., response time and throughput.

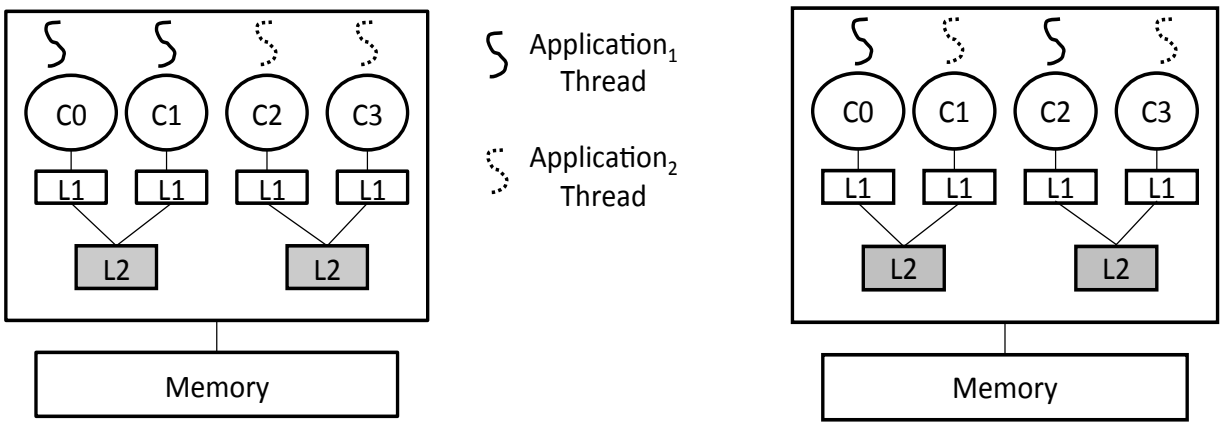
For the soft errors problem, a highly vulnerable application to soft errors can be identified by performing its resource occupancy behavior analyses. Once the highly vulnerable application(s) is(are) identified in a workload, the resource occupancy in the targeted resources can be reduced by mapping the application threads such that the overall vulnerability to soft errors is minimized.

Furthermore, this approach of mapping applications using prior characterization generalizes and can be used to improve performance and reliability for complex, dynamic workloads. Once the application characteristics are understood and identified, then as the applications start and finish their executions in a dynamic workload, thread-mapping can be used to adjust the resource usage of the currently executing applications to achieve the performance and/or reliability objective.

1.2.1 Contention/Performance Example

We describe two examples of how application characteristics can be used to address a resource contention problem - specifically cache contention. To achieve the goal of contention mitigation, application threads can be mapped appropriately by understanding their cache contentious behaviors.

Consider a workload that has two multi-threaded applications, and one of the applications, $Application_1$, has severe cache contention among its siblings threads. For the first example, if the second application, $Application_2$, of the workload has shared data among its sibling



(a) Example 1 (b) Example 2

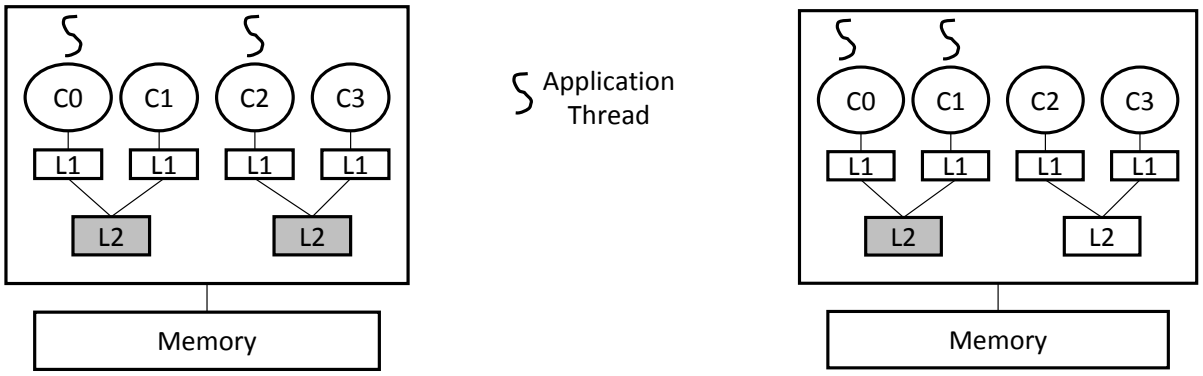
Figure 1.2: Mapping to mitigate shared-cache contention for a workload with two multi-threaded applications

threads, then the threads from $Application_1$ and $Application_2$ should be mapped such that each application uses its own L2-cache (shown in Figure 1.2(a)). This mapping ensures resource isolation so that the contentiousness from one application does not affect the cache sharing of the other application.

When such resource isolation is not possible, an application's sensitivity to the targeted resource, based on its characterization, can be used to determine the effective mapping, e.g., the most cache-intensive application can be mapped with the least cache-intensive application to share the same last-level cache [4]. For the second example, if $Application_2$ is computation-intensive (does not contend for cache), then the threads should be mapped such that the cache-intensive $Application_1$ threads share the same cache with the computation-intensive $Application_2$ threads (shown in Figure 1.2(b)). As $Application_2$ does not have much cache usage, this mapping would not degrade its performance significantly; however, it would mitigate contention for the cache-intensive application and improve its performance.

1.2.2 Soft Error Example

We now describe how an application's resource usage can be used to reduce its vulnerability to soft errors. Consider a workload with one multi-threaded application that executes for



(a) Example 1 (b) Example 2

Figure 1.3: Mapping to reduce application vulnerability to soft errors

a long time, and the application does not use the cache resources very frequently. If the application threads are mapped to use two separate caches (shown in Figure 1.3(a)), then the data in the two caches remain resident for a long period of time, and the application becomes more vulnerable to soft errors. This vulnerability can be decreased if the threads are mapped to use the same cache (shown in Figure 1.3(b)). This mapping can increase the cache miss rate, which lowers the residency time of the data and reduces the probability of being affected by soft errors. As a result, higher probability of correct application execution and output is obtained by this mapping.

To summarize, on a multicore platform, a multi-threaded application's characteristics and behaviors can be used to guide how to map application threads from a workload to improve performance and reliability.

1.3 The ReSense Framework

For the thread-mapping algorithm to be effective, it is critical to determine the applications' behaviors that affect the goal, i.e., mitigating contention and reducing occupancy duration in targeted resources (see Table 1.1). Determining such application behavior can be done *online* when the workload executes, by varying the resource usage for an application and analyzing how it affects the objective in the presence of co-runner(s). These online characterizations can

be used in an on-the-fly manner by the thread-mapping algorithm to achieve the objective of mitigating contention and reducing vulnerability.

For example, mapping different co-running applications from the given workload to use the same shared caches can vary the cache resource usage. An application's cache usage intensity creates contention in the shared cache, which eventually leads to application performance degradation. An application's cache usage characteristics can be determined online during its execution with the co-runner(s). The determined cache usage can be used to map the application threads with complementary cache behavior on the targeted machine during application execution to mitigate contention and ensure optimized performance [20, 4].

However, as the number of multi-threaded applications increases in a workload, the number of mapping configurations that must be evaluated to identify the application characteristics in the presence of co-runners and determine the optimal thread-mappings for performance and reliability improvement, can increase exponentially [21]. When the number of threads in a workload is very high, this exponential complexity makes the online resource usage detection and application thread-mappings very challenging and causes the characterization technique to have significant overhead. The online characterization becomes even more complex when dynamic workloads are considered because of the continuously changing and unpredictable set of co-running applications.

1.3.1 Application Characterization

To avoid the overhead and exponential complexity of an online characterization technique, an offline technique is used to determine a multi-threaded application's characteristics that are critical for addressing the targeted problem. This offline technique utilizes a general methodology to characterize a multi-threaded application when it runs solely without any co-runner. The characterization methodology isolates the effect of a targeted resource usage on a multi-threaded application's execution depending on the objective of performance optimization and reliability improvement. This isolation helps the application's behavior

analyses based on its usage of the targeted resource. Such analyses of isolated resource usage identify the application's key characteristics that precisely represent its sensitivity to that resource. Even though such characterization is performed without considering the presence of a co-running application, the sensitivity analyses help prioritize applications in a workload to determine the effective mappings for optimized performance and improved reliability.

1.3.2 Application Mapping

An application's inherent characteristics of resource usage influence how the performance and reliability on a targeted platform are evolved. Thread-mapping is an application-level software technique that can control an application's resource usage by changing the cores on which the threads from a workload would run. It does not require special hardware, is relatively easy to integrate into any system, and has very low run-time overhead. Therefore, thread-mapping is an attractive approach and is used in the framework to effectively address resource contention and soft errors for performance and reliability improvements on multicore machines.

1.3.3 Components of the Framework

Based on the insights and discussion in the previous sections, we develop a novel, unified, and general framework called **ReSense** that addresses several key problems on multicore architectures (see Table 1.1). The framework includes five components: a general characterization methodology, a characterization metric, a sensitivity score, a thread mapping algorithm, and a run-time system. An instance of the framework is applied in two phases: characterization and mapping. The offline *characterization* phase utilizes the general characterization methodology and characterization metric to identify a multi-threaded application's key behavior with respect to resource usage for a targeted problem. It generates a resource-sensitivity score for each application in a workload. In the online *mapping* phase, the run-time system uses a thread-mapping algorithm and the sensitivity scores of the applications in a workload to determine the thread-mappings that optimize a problem-specific objective function. The

run-time system is capable of handling complex dynamic workloads. Whenever the number of threads changes in the workload, the run-time system invokes the mapping algorithm to dynamically adjust the thread-mappings for the new or modified workload in the system.

1.3.4 Generality of the Framework

The ReSense framework can be used to address other problems on CMPs that are influenced by application characteristics, e.g., thermal and power consumption.

For example, an application's intensity of resource usage can have an impact on the processor temperature. Consider a computation-intensive application that uses integer and floating-point units can increase the processor temperature significantly. On the other hand, a memory-bound application can maintain lower processor temperature as it uses the core computational resources less intensely [22]. When these applications with different thermal profiles are considered in a workload, the intensity of their resource usage creates thermal variation inside the processor core and can cause thermal hot-spots, leading to a thermal emergency [22, 23]. An application's resource usage characteristics can also affect energy and power consumption of the targeted platform [24, 25].

The characterization phase of the ReSense framework can be used to identify how multi-threaded applications' resource usage affects the thermal and power issues on CMPs. Then, ReSense's dynamic thread-mapping algorithms can be used to reduce power consumption and minimize hot-spots for avoiding thermal emergencies.

Column 1 of Table 1.1 lists the general challenges on a multicore architecture and Column 2 lists the corresponding targeted problems that can be addressed using the ReSense framework. The table also includes the application behaviors to be determined in the characterization phase (Column 3), the overall goal of the mapping phase (Column 4), and the objective function used by the thread-mapping algorithm (Column 5). In this dissertation, we address the resource contention and soft errors problems (first two rows of Table 1.1) using the framework.

Challenges	Targeted Problem	Application Behavior	Goal/ Objective	Objective Function
Performance Optimization	Resource contention	Contentiousness in shared-memory resources	Mitigate contention	Response time and throughput
Reliability Improvement	Soft errors	Resource occupancy duration	Reduce occupancy duration in a resource	Vulnerability factor of a resource
Thermal Management	Thermal emergency	Computational resource usage	Distribute computation to minimize hot spots	Response time and throughput
Power Management	High power and energy consumption	Power usage in the resources	Reduce power and energy consumption	Energy delay product

Table 1.1: Targeted problems on modern CMPs to be addressed using the ReSense framework

1.3.5 Addressing Resource Contention and Soft Errors Using ReSense

An application's resource usage impacts its execution in a workload with respect to both resource contention and soft errors. Shared-resource contention makes the application execute slowly if the application uses the resources excessively. If an application occupies a resource for a long period of time, soft errors make it error prone. Both resource contention and soft errors can be addressed using the ReSense framework.

For the resource contention problem, the characterization methodology determines how a multi-threaded application's performance is affected as it shares a resource in the memory hierarchy. A sensitivity score based on resource contention is produced for each application in a workload. The run-time system uses the resource-contention sensitivity scores and a thread-mapping algorithm to allocate threads from a workload to core to mitigate shared-resource contention, thus improving response time and throughput (Table 1.1).

For the soft error problem, the characterization methodology determines how a multi-threaded application's vulnerability to soft errors in shared caches is affected by its resource occupancy duration. A sensitivity score based on cache occupancy is produced for each application in a workload. The run-time system uses the cache-occupancy sensitivity scores and a thread-mapping algorithm to allocate threads from a workload to core to reduce the overall occupancy in the shared caches, thus reducing cache vulnerability (Table 1.1).

For both contention mitigation and vulnerable occupancy reduction, the thread-mapping algorithms are designed to use the multi-threaded applications' prior characterizations that are determined offline. In particular, each multi-threaded application in any workload is characterized *only once* without considering the presence of any co-runner, and the algorithms map any combination of these multi-threaded applications using the pre-determined offline characterizations for optimizing the objective functions (Table 1.1). The offline characteristics for individual applications are represented as *sensitivity scores*, which quantitatively describe an application's solo-sensitivity to the targeted resources. When these applications are considered in a workload, these sensitivity scores are used to prioritize the applications when their thread-mappings are determined with respect to the targeted resources in the presence of co-runners(s). The most-sensitive applications affect the objective function more than the less-sensitive applications.

For example, when a workload consists of a cache-intensive and a computation-intensive application, the mapping that gives priority to the cache-intensive application results in a reduced response time and increased throughput because the cache-intensive application is more sensitive to the shared cache usage than the computation-intensive application [4]. Therefore, even though the application characteristics are not determined in the presence of co-runners, the sensitivity score captures an application's key behavior for the targeted resource, and applications in a workload can be prioritized and compared with each other based on how their sensitivity scores for the targeted resource usage. The thread-mapping algorithms thus work by prioritizing the most-sensitive applications in the workload based on

their sensitivity scores and dynamically determine the thread-mappings of the applications.

The performance and reliability improvements are often conflicting goals [26, 18]. The thread-mapping algorithm that ensures better performance may not ensure better reliability. By designing an algorithm that considers two objectives of the resource contention and soft errors problems (Table 1.1), in this thesis, we demonstrate the use of ReSense to determine a trade-off between reducing response time and vulnerability factors of shared caches for a workload.

1.4 Thesis Statement

On a multicore machine running multiple multi-threaded applications simultaneously, a workload's response time and throughput can be improved and vulnerability to soft errors in shared caches can be reduced via thread-mapping driven by the resource usage characteristics of the applications that are determined using an offline technique.

1.5 List of Contributions

This dissertation makes the following contributions.

- A novel and efficient offline technique for characterizing a multi-threaded application's resource usage by running it solely.
- The use of offline sensitivity characterization in determining the thread-mappings of multi-threaded applications in a workload to mitigate contention and reduce vulnerability to soft errors.
- ReSense, a unified framework that can be used to address the performance optimization, reliability improvement, thermal and power management challenges on a multicore platform.
- Development of the performance instance of ReSense, ReSense_Performance. This instance addresses the challenges of mitigating shared-resource contention in the memory

hierarchy and optimizes response time and throughput of the applications in a workload. Using dynamic workloads of different sizes, ReSense_Performance is able to improve the response time by up to 29.34% and throughput by up to 46.56% over the native operating system (OS) using real hardware.

- Development of the reliability instance of ReSense, ReSense_Reliability. This instance addresses the challenges of improving application reliability on multicore machines by using an application-level technique and minimizes the overall cache vulnerability to soft errors on a targeted multicore platform. ReSense_Reliability effectively reduces the overall cache vulnerability by up to 70% over the native OS.

- Development of the integrated instance of ReSense, ReSense_Integration. This instance combines application characteristics for both contention and vulnerability to soft errors and determines a trade-off between the performance and reliability improvement on a multicore machine. When applications' contentious behaviors are more preferred, ReSense_Integration reduces applications' response time, and when applications' vulnerability characterizations are more preferred, it reduces the overall shared cache vulnerability.

1.6 Thesis Outline

The thesis is organized as follows: Chapter 2 describes the state-of-the-art related work for resource contention and soft errors. Chapter 3 describes the design of the ReSense framework in more detail. Chapter 4 presents the performance instance of the ReSense framework, including the characterization process for resource contention and the mapping algorithm for performance optimization. Chapter 5 presents the reliability instance of ReSense framework, including the characterization methodology for application vulnerability and the mapping algorithm for reliability improvement. Chapter 6 describes the integration of the performance and reliability instances to determine a trade off between performance and

reliability improvements. Chapter 7 summarizes the findings, describes the future work, and concludes the thesis.

Chapter 2

Related Work

In this chapter, we discuss prior research related to resource contention and soft errors. First, we describe the research on application characterization for resource contention and vulnerability to soft errors. Then we describe prior research on the techniques for contention mitigation to improve performance and the techniques for reducing the effects of soft errors to improve reliability.

2.1 Application Characterization

2.1.1 Resource Contention Characterization

2.1.1.1 Characterization for Single-threaded Applications

There has been prior work on application characterization for shared-resource contention. Zhuravlev et al. analyzes the effect of cache contention created by co-runners and provides a comprehensive analysis of different cache-contention classification schemes [4]. Mars et al. synthesizes and analyzes cross-core performance interference for last-level caches (LLC) on two architectures [27, 28]. They characterize the applications in the presence of co-runners using synthetic workloads and determine the effects of contention for the memory resources [29]. Xie and Loh characterize and classify applications by measuring cache miss-rates for a dynamic cache partitioning scheme [30]. Zhao et al. investigates low overhead

mechanisms for fine-grained monitoring of shared cache usage, interference, and sharing to determine application characteristics [31]. These works mainly analyze cache contention for single-threaded applications. In contrast, we design a general methodology in ReSense to characterize any multi-threaded application for not only LLC contention, but also contention for private caches, front-side bus, memory controllers, and memory socket connections.

2.1.1.2 Characterization for Multi-threaded Applications

There has been research work on characterizing multi-threaded applications for contention. Jin et al. characterizes parallel workloads for resource contention in the memory hierarchy, but they mainly focus on comparing different platforms and run applications solely [32]. Whereas we focus on characterizing multi-threaded applications, both when an application runs alone and with co-runner(s), and determine its sensitivity to contention for a particular resource. Tang et al. analyzes and studies the performance impact of contention and sharing for the resources in the memory subsystem on multi-threaded data-center applications [33]. Kambadur et al. describes a methodology to measure interference between data-center applications while they are executing [34]. Luo et al. analyzes multi-threaded applications' memory footprints to understand the active data sharing behavior among the threads [35]. Natarajan et al. characterizes multi-threaded applications based on sharing-awareness in the replacement policies of last-level caches [36]. Krishna et al. analyzes data sharing among multi-threaded applications in a workload and extends existing analytical models for novel many-core chip designs [11]. These works mostly analyze a multi-threaded application's sharing or contentious behavior for cache usage, whereas we design a methodology in ReSense to analyze an application's both sharing and contentious behavior for any resource in the memory hierarchy.

There also has been prior work on characterizing multi-threaded applications from the PARSEC benchmark suite for resource contention. Bhaduria et al. describes the cache performance, sensitivity with respect to DRAM speed and bandwidth, thread scalability, and micro-architectural design choices for the benchmarks over a wide variety of real ma-

chines [37]. In the original PARSEC paper [38], the authors provide several characteristics of the benchmarks including working set, locality, effect of different cache block size, degree of parallelization, off-chip traffic, and programming models. Bhattacharjee et al. describes TLB behavior of these benchmarks and provides many useful insights about redundant and predictable inter-core I- and D-TLB misses, which are useful for better and novel TLB designs for emerging parallel workloads [39]. The authors transform the PARSEC benchmarks in a cache-sharing-aware manner during compilation time to improve the sharing behavior among sibling threads [7]. Most of the work analyze and characterize the PARSEC benchmarks for solo-execution. Our work complements the above research as we characterize the benchmarks both for solo-execution and execution with co-runners on real hardware.

2.1.2 Vulnerability Characterization

2.1.2.1 Different Vulnerability Factors

There have been prior work on characterizing applications using different types of vulnerability factors. Architectural vulnerability factor (AVF) is a metric first introduced by Mukherjee et al. to represent an application's susceptibility to soft errors without using an error injection methodology [18]. Yan and Zhang define a register vulnerability factor (RVF) and propose two cost-effective compiler-guided techniques to improve register reliability by minimizing RVF [40]. Jongeun and Shrivastava also analyze an application's RVF for different compiler optimizations [41]. Sridharan and Kaeli introduce a hardware vulnerability factor (HVF) to quantify the vulnerability of an individual system component [42]. Oz et al. defines a thread vulnerability factor (TVF) for multi-threaded applications to analyze how the communication among threads changes TVF and determines a trade-off between performance and reliability for different algorithm implementations [26]. In this work, we consider a multi-threaded application's resource occupancy in the shared cache and do not consider their communication behaviors among sibling threads. Therefore, we do not use TVF as the characterization metric.

Sridharan and Kaeli define and describe a program vulnerability factor (PVF) metric to capture architectural-level fault masking inherent in an application, eliminating the micro-architectural dependency. They propose two uses of PVF, which are helpful in application development to reduce the failure rate [43]. In this work, ReSense maps multi-threaded applications based on an application's resource usage behavior that is dependent on the hardware configurations of a particular platform. Therefore, we do not use PVF as the characterization metric because it is defined to be independent of the hardware features.

Wibowo et al. describes a cross layer approach to calculate system vulnerability factor for the register file using AVF and application's code vulnerability [44]. Wang et al. proposes a new analytical model to estimate the system-level vulnerability factor for on-chip instruction caches [45]. Zhang defines a cache vulnerability factor (CVF) and evaluates the reliability for different cache memories, including a write-through and a write-back cache [1]. In this thesis, ReSense_Reliability targets to reduce applications' vulnerability to soft errors in caches. Therefore, it uses cache vulnerability factor as the characterization metric to determine how a multi-threaded application's cache vulnerability changes for its shared cache usage and occupancy. Biswas et al. describes a methodology to calculate a quantized AVF (Q-AVF) online for a specific time interval using linear regression [46]. On the other hand, the characterization phase of ReSense_Reliability determines an application's vulnerability to soft errors for its entire duration of execution.

2.1.2.2 Vulnerability Analyses for Micro-architectural Resources

There have been several works analyzing the vulnerability of different systems and applications. Soundararajan et al. analyzes a multicore system's vulnerability by varying different numbers of cores and threads configuration [47]. Fu et al. analyzes an application's phase-wise vulnerability in different micro-architectural structures, including instruction window, re-order buffer, and functional units in a high-performance out-of-order super-scalar processor [48]. Zhang et al. analyzes application vulnerability for simultaneous multi-threaded (SMT) architectures and proposes potential opportunities to reduce vulnerability for SMT micro-

architecture by exploiting thread-aware reliability optimization technique [14]. Duan et al. characterizes application vulnerability for shared micro-architectural resources and L1-caches using a multi-threaded processor [49]. Nair et al. characterizes an application based on its worst-case AVF using a stress-test analyses to identify the reliability bottleneck [50]. Tankhi et al. determines and analyses an application's RVF for embedded processors using a power efficient approach [51]. Li et al. develops a novel methodology and analysis tool to determine the effect of soft errors on extreme-scale scientific applications [52]. All these works analyze vulnerability mainly for micro-architectural resources, whereas, we design a general methodology in ReSense to characterize a multi-threaded application for its vulnerability to soft errors in both micro-architectural and memory resources. ReSense's characterization methodology is complementary to these approaches.

2.1.2.3 Vulnerability Analyses for Memory Resources

There has been also research on analyzing application vulnerability for memory resources on different systems. Luo et al. analyzes the memory error vulnerability for data-center applications by quantifying the applications' tolerance to soft errors in memory and proposes heterogeneous memory systems [53]. Ma et al. characterizes an application based on its vulnerability to soft errors in caches at different level of the memory hierarchy using a fault injection methodology [54]. Wang et al. proposes a framework for conducting comprehensive studies and characterizations of the reliability behaviors of L1-data and instruction cache memories for novel designs [55]. Other research efforts determine an application's susceptibility to soft errors in both micro-architectural and memory resources using different compiler optimization flags [40, 56, 41]. In contrast, in this work, the general methodology of the ReSense framework for ReSense_Reliability characterizes multi-threaded applications for any cache resource on any multicore machine via thread-mapping, which is an application-level technique.

2.2 Shared-resource Contention Mitigation

There have been research efforts that address contention for co-located applications using thread-mapping as a technique. Table 2.1 summarizes the important differences between ReSense_Performance and a few state-of-the-art systems for mitigating contention for the shared-memory resources. Some of these systems pursue the goals to independently optimize energy, thread throughput, minimize lock contention, allocate optimal number of cores to the applications in a workload, or improve only the latency-sensitive application's performance. ReSense_Performance's goal is to improve the overall performance of the co-located multi-threaded applications from dynamic workloads and mitigate contention for all the shared resources in the memory hierarchy by determining the effective thread-mapping. All the systems mentioned in Table 2.1 determine application characterizations in the presence of a co-runner. In comparison, ReSense_Performance is able to characterize applications without considering co-runners. Thus, ReSense_Performance operates in linear time to determine application characteristics and is able to achieve similar performance improvements and perform competitively with the other systems, where the characterization operates in polynomial time. Details of some systems and the comparisons with ReSense_Performance are described in Section 2.2.2.

2.2.1 Contention Mitigation for Single-threaded Applications

A number of prior works have addressed resource contention (mostly shared caches) for single-threaded applications using execution throttling, scheduling, and thread-mapping [6, 4]. In particular, Mars et al. describes an optimization technique that detects cross-core contention in shared resources online and ensures quality-of-service for single-threaded SPEC benchmarks ensuring contention aware execution [10]. Fedorova et al. describes a co-runner dependent cache allocation via OS scheduling [57]. The authors address LLC pollution and propose a page coloring based technique to eliminate the cache pollution [58]. Jiang et al. proposes a hierarchical matching algorithm to co-schedule threads for reducing inter-thread latency by

Systems	ReSense_ Performance	Pusukuri et al. TACO'13	Bhadauria et al. ICS'10	Tang et al. ISCA'11	Zhuravlev et al. ASPLOS'10
Resources	All shared resources in memory hierarchy	Last-level cache and shared lock	Shared cache and bus	All shared resources in memory hierarchy	All shared resources in memory hierarchy
Objective Function	Minimize workload's response time and maximize throughput	Minimize workload's turnaround time maximize throughput	Minimize system's energy and maximize thread throughput	Satisfy quality of service of one latency-sensitive application	Reduce workload's completion time
Application characteristics detection complexity	Linear, no co-runner considered	Polynomial, increases with the number of co-runners	Polynomial, increases with the number of co-runners	Polynomial, increases with the number of co-runners	Polynomial, increases with the number of co-runners
Multi-threaded application	Yes	Yes	Yes	Yes	No
Dynamic workloads	Yes	Yes	No	No	No
Performance compared with optimal/oracle	Average less than 1%	Unknown	Average less than 1%	Within 3%	Within 2%
Baseline comparison	Native operating system	Native operating system	Suleman et al. ASPLOS'08	None	Native operating system

Table 2.1: Comparison between ReSense_Performance and some state-of-the-art systems

considering LLC-contention and sharing [59]. All these works address contention for single-threaded applications by determining its contentious behavior in caches in the presence of a co-runner. The run-time system in ReSense_Performance dynamically maps multi-threaded applications, considering not only contention for shared caches, but also for bus and memory interconnections, without considering any co-runner(s).

2.2.2 Contention Mitigation for Multi-threaded Applications

There have been a few works on contention mitigation for multi-threaded applications. Bhadauria et al. schedules threads from multiple multi-threaded applications at a time

quantum to optimize thread throughput and energy [60]. At a particular time quantum, their algorithm selects a number of threads from each application in the workload and the applications to run together by considering an application's cache miss rates (FAIRMIS policy) or bus occupancy (FAIRCOM policy). They do not consider an application's characteristics for both cache and bus in the same policy. On the other hand, ReSense_Performance mitigates contention for all the shared resources in the memory hierarchy considering both cache and bus characteristics of the applications at the same time.

Pusukuri et al. allocates cores to multi-threaded applications using an application's cache and lock contention characteristics that are determined in the presence of a co-runner [61]. In the presence of r co-runners, the number of characterizations grows polynomially, $O(n^r)$, for n applications. Their system determines the number of cores to be allocated to each application using a supervised learning technique, which requires more offline analyses compared to ReSense_Performance. In contrast, ReSense_Performance does not require any training phase and characterizes application without considering any co-runner. This solo-characterization has linear complexity of $O(n)$, which is much lower than $O(n^r)$.

Pusukuri et al. describes a scheduling policy for minimizing lock contention for multi-threaded applications [62]. Emani et al. determines the thread count to improve an application's performance in the presence of external workloads [63]. Das et al. describes application-to-core mapping for NoC systems [64]. Garcia et al. investigates dynamic scheduling for "embarrassingly" parallel applications for CMPs [65]. Broquedis et al. describes a scheduling framework for OpenMP applications [66]. These works primarily pursue the goals to minimize lock contention, optimize a single application's performance, minimize communication overhead or focus on core-allocation, NoC, and data-parallel application. These goals are different than the goal pursued by ReSense_Performance, which targets optimizing the average response time and throughput of *every* multi-threaded applications from a dynamic workload by determining the thread-mappings.

Chen et al. proposes scheduling threads that share data to use the same cache to

improve performance for multi-threaded applications [67]. ReSense_Performance considers both contention and data sharing in caches to map multi-threaded applications. Xu et al. presents a scheduling technique to minimize the bandwidth contention fluctuation [68]. This online scheduling technique minimizes the bandwidth contention by maintaining bandwidth utilization at the level of average bandwidth requirement of the workload. On the other hand, ReSense_Performance uses offline characterization to mitigate the bandwidth contention for the multi-threaded applications in a workload.

2.2.3 Mapping Applications using Prior Characterization

Some research efforts have proposed the idea of mapping application threads or managing shared resources based on prior characterization. Mars et al. analyzes cross-core performance interference to determine a contention conscious scheduling [27]. They also schedule and determine the mapping of co-located applications by characterizing the application in the presence of co-runners and using scores from stress-test via synthetic workloads [29]. Tang et al. studies the impact of memory subsystem resource sharing on data-center applications and schedules the threads to meet the quality-of-service requirements of the latency-sensitive applications [33]. Jaleel et al. maps applications using a run-time classification based on cache replacement policy [69]. In this work, ReSense_Performance utilizes an application's prior performance characterization for individual shared resources by running it solely, without using any co-running applications or special hardware policies, and is capable of determining the thread-mappings for dynamic workloads consisting of multiple multi-threaded applications.

2.2.4 Mapping Applications using Performance Prediction

There have been several efforts at designing different analytical models that can be used to map application threads effectively. Tipp et al. describes a linear regression model from hardware performance counters for shared-resource contention, including functional units, issue bandwidth, caches, for hyper-threaded machines. Because multicore machines have separate micro-architectural units, we focus on the shared resource-contention in the

memory hierarchy. Chandra et al. proposes analytical probabilistic models to analyze inter-thread contention in the shared L2-cache in hyper-threaded CMPs for single-threaded applications [70]. Xu et al. proposes a shared-cache aware performance model using reuse distance histogram, cache access frequencies, and the relationship between throughput and cache miss-rate to map processes on cores [71]. In contrast, ReSense_Performance uses an application's solo performance characterization for each shared resource in the memory hierarchy to predict the effective thread-to-core mappings in the presence of co-runner(s).

2.2.5 Cache Partitioning Techniques

Several works address shared resource management in CMPs via hardware cache partitioning to mitigate contention [8, 9] and software methods to partition the cache and allocate memory pages [72, 3, 73, 74, 58]. Ding et al. uses an approach similar to cache partitioning to contain the operating system file buffers into a small portion of the shared LLC to less pollute it [75]. The ReSense approach is compatible and can be combined with both hardware and software cache partitioning techniques to further reduce cache contention and improve performance.

2.3 Techniques for Addressing Soft Errors

There has been prior work addressing soft errors in processor and memory resources using both hardware and software techniques. These techniques can be further classified into two categories: (a) error detection and correction, and (b) error prevention.

2.3.1 Error Detection and Correction

2.3.1.1 Hardware Techniques

Hardware techniques for error detection and correction typically include *redundancy in space* to protect an application's outputs from erroneous execution because of soft errors. Space redundancy includes the addition of error correcting codes (ECC) in the low-level circuits, particularly in memory systems [76]. These ECCs can be used to re-compute the results from an application's computation and determine incorrect execution by checking for any output

mismatch. These redundant codes can ensure very high accuracy in application execution. However, these techniques have significant power, area, cost, and latency overhead, and are not efficient and practical for systems with a large number of cores [17].

Several research efforts address soft errors by proposing error detection and recovery via hardware-based redundant multi-threading [18, 77]. These approaches use special hardware to replicate an application execution to identify errors in the output, which increases hardware design complexity and cost, including high performance overhead.

2.3.1.2 Software Techniques

Software techniques for error detection and correction include redundant execution, which is *redundancy in time*, to ensure reliability against soft errors [17, 13, 78, 79, 80]. In general, this approach creates multiple copies of the same execution to detect any mismatch in the application output, and consequently chooses the right output to ensure reliable execution. Although this technique is highly accurate in ensuring correct application output, it has a very high performance overhead: 30% for single-threaded and 32% for multi-threaded applications [18]. Wang et al. proposes another software-based redundant multi-threading for soft error detection using compiler analysis and optimization techniques [81]. Reis et al. presents a software-only fault tolerant technique to manage execution redundancy via an enhanced control flow mechanism [82]. In contrast to these works, the thread-mapping technique used by ReSense_Reliability reduces the probability of visible wrong output caused by soft errors without any redundant execution during an application's execution.

2.3.2 Error Prevention

2.3.2.1 Hardware Techniques

There has been much research on reducing an application's vulnerability to soft errors in caches using hardware and software techniques for error prevention. Hardware techniques for error prevention include *device hardening* that increases the capacitance of the critical nodes in the SRAM cells, increasing its reliability [16, 83]. There are a number of other hardware

mechanisms that can be used to increase the capacitance of the SRAM cells. They include increasing the size of the transistors [84], isolating critical nodes in a circuit [85], tweak the critical threshold voltage of the circuit [86]. However, all these techniques have area and performance overhead [16].

Other error prevention techniques are architecture-level, including cache flushing, layout interleaving, scrubbing, early write-back. For cache flusing, the operating system or the hardware flushes the cache to reduce its vulnerable lifetime, resulting in lower AVFs [87, 88]. Zhang et al. describes a technique that performs early write-backs of the dirty cachelines to the memory at periodic interval to reduce the cache vulnerability [1]. Similar to memory scrubbing [89], cache scrubbing is used to calculate the error correction code to avoid accumulated bit-flips that can result in corrupted data in caches [90]. Sridharan et al. describes an architecture-level technique to reduce L1-cache vulnerability by selectively refetching cachelines from L2-caches for single-threaded applications [16]. Jeyapaul et al. proposes a smart cache cleaning methodology that copies only specific vulnerable cache blocks into the memory at chosen times to ensure data cache protection with minimal memory writes [91].

Most of these hardware techniques are implemented and evaluated using a uniprocessor system in the architectural and design level, whereas, in this research, we address soft errors for multicore machines. The cache flushing technique can be used to prevent errors in each cache on a multicore platform. However, as a multicore platform has multiple caches, flushing the content of all these caches can lead to a significant increase in cache miss rates, resulting in severe application performance degradation. Refetching the data from the memory into the caches on a multicore platform has to be done carefully because the same data can be present in multiple caches in different states for the cache coherency protocol, which adds additional complexity. Consequently, cache-line refetching for caches on a multicore system can be difficult to implement. Thus, we conclude that these techniques for uniprocessor systems may not be effective and easily extended for addressing soft errors on multicore systems.

In this research, ReSense_Reliability utilizes an application-level technique using operating system calls to reduce a multi-threaded application's vulnerability to soft errors in caches, which can be combined with these techniques.

2.3.2.2 Software Techniques

Prior research using software measures to prevent soft errors includes different compilation techniques. Jones et al. analyzes the effect of different compiler optimizations on AVF and determines the best combination of compiler optimization flags to improve application performance with negligible effect on AVF [19]. Demertzi et al. analyzes an application's reliability based on the compiler optimizations' impact on the occupancy in three processor structures, including re-order buffer, instruction queue and load-store units [92]. Martinez-Alvarez et al. describes a compiler-directed soft error mitigation technique and demonstrates two case studies for embedded processors [93]. Instruction scheduling by compilers is applied to make the application more resilient towards soft errors [94, 95]. In this thesis, ReSense_Reliability minimizes the cache vulnerability of the applications in a workload using dynamic thread-mapping technique, which is orthogonal to these static compiling approaches. Walcott et al. proposes linear regression models for dynamic prediction of AVF from key processor metric using statistical analysis [17]. Kadayif and Kandemir propose a soft error model for caches and explore three architectural schemes to enhance reliability [96]. On the other hand, ReSense_Reliability uses cache vulnerability models to predict the effective thread-mappings to reduce the probability of soft errors affecting application execution and producing visible errors.

There has been a few work on designing system-level techniques to reduce application vulnerability to soft errors. Duan et al. proposes a machine-learning based model to determine soft error resilient thread-to-core scheduling and reduce application vulnerability to soft errors in micro-architectural resources [49]. On the other hand, instead of using complex machine learning models, ReSense_Reliability uses prior characterization of the applications to determine the thread-mappings that minimize shared cache vulnerability to soft errors for

workloads consisting of multiple multi-threaded applications.

Chapter 3

ReSense: A Unified Framework

In this chapter, we describe the unified framework, ReSense, which is designed to tackle several challenges on a CMP platform. These challenges relate to effective application execution and resource management on a multicore platform for optimizing performance, improving reliability, better thermal and power management (Table 1.1). For example, to minimize a workload's response time and maximize throughput on a CMP, the targeted problem of resource contention is critical to address [6, 59, 27].

When multiple multi-threaded applications simultaneously execute on a CMP platform, these applications use the underlying resources differently depending on how the application threads are mapped on different cores and the resource topology of the platform. The way these applications use different resources has an impact on the workload's execution and overall resource management, leading to the performance optimization, reliability improvement, power consumption, and thermal challenges on CMPs. For example, a cache-intensive application can increase a workload's response time by creating contention in the shared caches [4]. A computation-intensive application that uses integer and floating-point units can increase the processor temperature significantly and cause thermal emergency [22, 23]. Similarly, an application's resource usage characteristics can also increase energy and power consumption on the targeted platform [24, 25]. If an application occupies a particular resource

for a long time during its execution, it is more susceptible to the bit-flips caused by soft errors induced by high energetic beam particles. Thus, an application's resource usage behaviors can also lead to increased vulnerability to soft errors during its execution and have an impact on reliability [13, 14]. Fundamentally, the intensity of resource usage of the applications in a workload critically impacts the performance, power, reliability, and thermal problems on a multicore machine.

The challenges related to performance optimization, reliability improvement, thermal and power management can be individually addressed by understanding and analyzing how the resource usage characteristics of the applications in a workload affect a targeted problem. The insights from such analyses can be used to achieve the corresponding goal or objective for that problem. For example, when the targeted problem is cache contention, understanding the applications' contentious behaviors for cache usage is helpful to determine the thread-mappings of the applications in a workload for contention mitigation and response time and throughout improvements (See Table 1.1) [4, 59]. Similarly, when applications with different thermal profiles are considered in a workload, understanding the intensity of their computational resource usage helps distributing the computation of the threads such that thermal emergencies can be avoided and applications' response time and throughout can be improved (Table 1.1). To address these problems, it is important to understand and analyze how the resource usage characteristics of the multi-threaded applications in a workload impact the overall application execution in terms of performance, reliability, power consumption, and thermal problems for a targeted platform. The insights from such characterization and analyses can be used to control the resource usage of the applications by intelligently mapping them on the appropriate cores such that the objective for the targeted problem is obtained.

In this research, we develop the ReSense framework to address several challenges on multicore architectures including performance optimization, reliability improvement, thermal and power management. An overview of the framework is described below in the following section, with more details in Section 3.2 and 3.3.

3.1 Overview and Example of the Framework

Figure 3.1 shows the components of the ReSense framework. The framework includes five major components: a general characterization methodology, a characterization metric, a sensitivity score, a thread mapping algorithm, and a run-time system. An instance of the framework is applied in two phases: characterization and mapping. The *characterization* phase includes the first three components of the framework. This phase of the framework is applied offline and takes the *applications in a workload* and a *targeted problem* as input. This phase identifies the characteristics of the applications for its resource usage based on some objective. This objective determines the application behaviors that are critical to address the targeted problem on a platform P. The applications' characteristics are determined offline by applying the general *characterization methodology* that runs each multi-threaded application in the workload by itself. This methodology isolates the effect of the application's usage of the targeted resource and determines a *sensitivity score* using a *characterization metric*. The sensitivity score represents a multi-threaded application's sensitivity towards the targeted resource usage. These scores are used to compare the resource sensitivities of the applications when they execute in a dynamic workload.

The *mapping* phase is performed online and includes the remaining two components of the framework: a thread-mapping algorithm and run-time system. The *thread-mapping algorithm* uses the sensitivity scores of the applications and the resource *topology of* P to dynamically determine the mappings of the threads from the input workload to optimize the problem-specific *objective function*. The *run-time system* takes as input the sensitivity scores and the *dynamic workloads* that consist of multiple multi-threaded applications executing in any order. It detects any execution changes in the workload in terms of number of threads and employs the thread-mapping algorithm to map the applications on P.

We describe an instance of the ReSense framework using cache contention as the targeted problem. Assume a workload has two multi-threaded applications, and these applications are

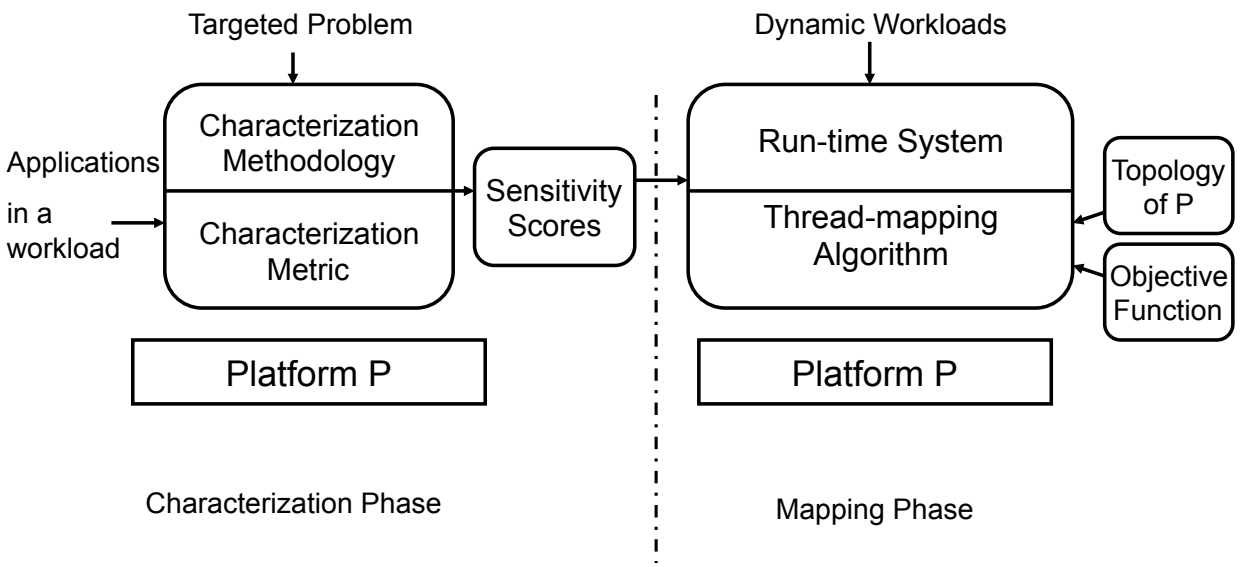


Figure 3.1: Components of the ReSense framework

to be mapped on a multicore machine. When multiple applications are executed on a CMP platform, extensive cache usage by these applications can cause high levels of contention. The created contention for caches can lead to severe application performance degradation. Here, the goal is to mitigate cache contention and the objective function is to minimize the workload's response time and maximize throughput (Table 1.1). Understanding how the applications use cache resources is critical for the contention mitigation. Therefore, the objective of the characterization phase is to determine how an application's contentious usage of cache resources affects its performance, which is used as the characterization metric. The methodology characterizes each multi-threaded application in the workload based on how shared-cache usage among its sibling threads affects its performance by running it solely.

If two applications in the workload are characterized to have contentious behavior among its sibling threads, then the applications are mapped to use separate caches to mitigate this contention, as shown in Figure 1.2(b). If one of the applications is more cache contentious, then the most cache-intensive application threads are mapped with the least cache-intensive threads of the other application to share the same caches to mitigate as much contention as possible. The most- and least-sensitive application for cache contention are identified

based on the resource usage and sensitivity scores determined in the characterization phase. Thus, even though the applications are not characterized in the presence of co-runner(s) from a given workload, application threads can be mapped based on its *solo* resource usage characteristics for contention mitigation and response time and throughput optimization.

The following two sections describe the components of the framework used in the characterization and mapping phase in more detail.

3.2 Characterization Phase

An application can be characterized based on its numerous behaviors, e.g., memory access behavior, cache usage behavior, total power consumption, and computational resource usage. An important component of ReSense is a general characterization methodology that is used to determine the inherent characteristics of a multi-threaded application, which are critical to address the targeted problem. This general methodology includes a criteria or *objective* to determine the application characteristics that influence or affect the targeted problem. For example, if the problem is cache contention among multiple applications, then the objective of the methodology is to identify an application's contentiousness for its cache usage (Table 1.1) and determine how it affects the overall contention problem.

Modern multicore machines have multiple instances of the same resource, e.g. caches, thread contexts for simultaneous multi-threaded processors, memory controllers, and memory socket connections. If the problem is influenced by the number of resources used by an application, the methodology determines the effect of using the resources on application execution and how much they impact the targeted problem. The methodology controls the number of targeted resources a multi-threaded application uses by placing the threads on the cores of the targeted resources. Then it characterizes the application based on its resource usage behavior by varying the number of targeted resources used by the application.

For the cache contention problem, to characterize an application for its shared-cache usage on a machine with multiple L2-caches, the application threads are placed on the cores that

use one or two L2-caches. The application is characterized by determining how the number of L2-caches used by the application impacts the severity of contention and application execution.

For the thermal management problem, a multi-threaded application's threads are placed on the same die (core C0 and C1 of Figure 1.1) or on different dies (core C0 and C2). The application is characterized for its thermal behavior by comparing the processor temperature as the threads use the computational resources of the cores on the same or different dies.

There are three major components of the framework used in the characterization phase: characterization metric, methodology and sensitivity score. These components are described in the following sections.

3.2.1 Characterization Metric

To understand and analyze an application's behavior for the targeted resource usage, it is important to quantify how much the application's execution is affected by the targeted problem. Such quantification of the effect is used to characterize the application and is helpful to understand the severity of the problem being exacerbated by the behaviors of multiple applications in a workload. To perform the characterization, the methodology utilizes a *characterization metric*, which represents the effect of the targeted problem. Different configurations of the threads' placement on the cores are used to isolate the effect of a multi-threaded application's usage behavior for a particular resource, and the metric is used to quantify this effect. For cache contention, if a multi-threaded cache-intensive application threads are placed on the cores that use the same cache, then the contention for the shared cache can degrade its performance. Here, performance is used as the characterization metric. The actual value of an application's performance degradation quantifies the severity of contention. This characterization metric is later used to analyze and determine a multi-threaded application's sensitivity for the targeted resource.

3.2.2 Characterization Methodology

On a particular platform, a general characterization methodology places application threads in different characterization configurations to isolate the effects of resource usage on an application's execution. Two different characterization configurations measure the effect of an application's resource usage on its characterization metric. In the baseline or *non-sharing* configuration, the application threads are placed on the cores that use two different targeted resources. In the *sharing* configuration, the threads are placed on the cores that use the same targeted resource. We compute the characterization metric in both configurations. Based on the difference between the characterization metrics from the non-sharing to the sharing configuration, we characterize a multi-threaded application. For example, when we place two threads from an application on the same core and the processor temperature increases compared to the placing of the same threads on different cores, then we conclude that the sibling threads use the computational resources extensively and characterize the application to create a potential thermal emergency.

To determine the effect of targeted resource usage on an application's execution, the methodology requires using a platform that has multiple numbers of targeted resources with the same parameter values. For example, if we characterize an application for L3-cache contention, we need a system that has three levels of caches and at least two L3-caches so that we can use the characterization configurations to apply the methodology. The parameters values are cache parameter, including cache size, number of ways, and cache block size. In addition, the capacity sizes of the other related resources and how they are connected to the targeted resources are required to be the same. For the previous example, the sizes of L1-, L2-cache and how they share the L3-cache must be the same in the resource topology of the platform.

The characterization methodology can be used as a stand-alone technique to determine the resource usage behavior of any multi-threaded application.

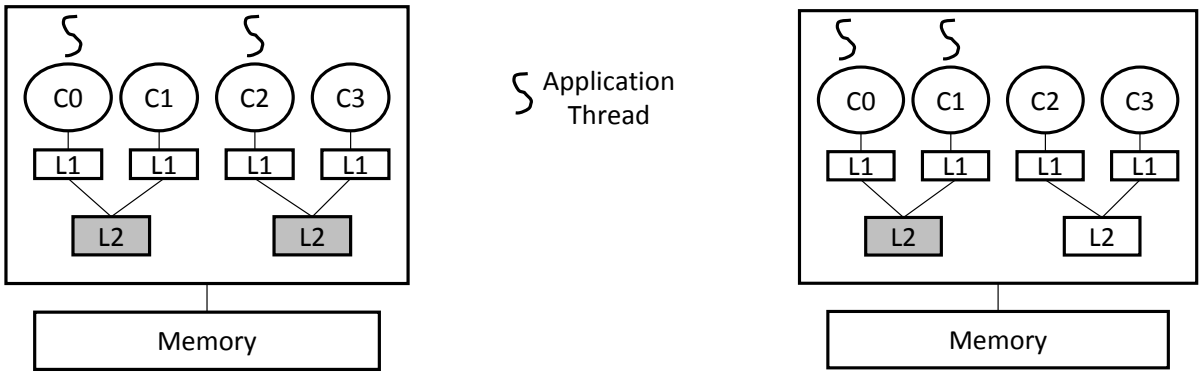
3.2.3 Sensitivity Score

An application's sensitivity for a targeted resource is the difference between the characterization metrics in different characterization configurations. Each application's characteristics are represented as a *sensitivity score* for the targeted resource, which is calculated according to Equation 3.1. Each application in a workload needs to be characterized *only once* offline for a targeted platform.

$$SensitivityScore = \frac{(CharacterizationMetric_{non-sharing} - CharacterizationMetric_{sharing}) * 100}{CharacterizationMetric_{non-sharing}} \quad (3.1)$$

Here, $CharacterizationMetric_{non-sharing}$ and $CharacterizationMetric_{sharing}$ are the values of the characterization metric in the two characterization configurations. Each sensitivity score has two components: a sign and a magnitude. The sign represents whether the application's characterization metric increases or decreases in the sharing configuration, and the magnitude represents the actual value of the metric being increased or decreased. A positive sign of SensitivityScore means the characterization metric decreases when the application threads use the same targeted resource. A negative sign of SensitivityScore means the characterization metric increases when the application threads use the same targeted resource. The magnitudes are used to compare the sensitivity of the applications in a workload for the targeted resource and help prioritize the applications when their thread-mappings are determined in the online mapping phase.

For example, to characterize a multi-threaded application for contention, we execute the application in two characterization configurations using L2-cache as the targeted resource and performance as the characterization metric. In the non-sharing configuration, two application threads are placed on the two cores that have separate L2-caches, e.g., C0, C2 (shown in Figure 3.2(a)) or C1, C3. In the sharing configuration, two application threads share and contend for one L2-cache with each other. Here, the threads are placed on the two cores that



(a) Non-sharing Configuration (b) Sharing Configuration

Figure 3.2: Configurations to characterize a multi-threaded application L2-cache contention

share the same L2-cache, e.g., C0, C1 (shown in Figure 3.2(b)) or C2, C3. As we characterize an application for L2-cache usage, we avoid L1-cache contention by allowing only one thread to access one L1-cache in both configurations. We measure the application's performance in both configurations and calculate its sensitivity score for L2-cache. If the sensitivity score is positive, the application's performance improves in the sharing configuration, and the application is characterized to have data sharing in the cache. If the sensitivity score is negative, the performance degrades and the application is characterized to have contention for the shared cache among the sibling threads.

If the co-runners were considered in the characterization phase, the characterization complexity would increase polynomially. For example, for one co-running application, there are $O(n^2)$ pair-wise characterizations and for $(r-1)$ co-running applications, there are $O(n^r)$ characterizations for n applications for a targeted resource. Depending on the number of applications and co-runners, the number of configurations to determine an application's resource usage behavior can be high during the characterization phase. To avoid the characterization overhead associated with this high number of configurations, each multi-threaded application in a workload is characterized without considering the presence of any co-runner. The signs and magnitudes of the sensitivity scores help rank the applications

according to their sensitivity for the targeted resources and determine the thread-mappings to optimize the objective function.

The sensitivity scores of the applications in a workload based on different characterization objectives can be combined to address multiple targeted problems. For example, we characterize a multi-threaded application based on shared cache usage for both its contention and vulnerability behaviors separately. These characterizations result into two separate sensitivity scores. We then can combine these characterizations and determine a combined sensitivity score for each application, which is used in the mapping phase to determine a trade-off between performance and reliability objective (Chapter 6). Similar integration is also possible for other problems.

3.3 Mapping Phase

The mapping phase is composed of two major components of the framework: a thread-mapping algorithm and run-time system. These are described in the following sections.

3.3.1 ReSensor$_{\text{Generic}}$ Thread-mapping Algorithm

The thread-mapping algorithm maps any combination of the multi-threaded applications in a workload using the signs and magnitudes of their pre-determined sensitivity scores to optimize an *objective function* for the targeted problem. For the cache contention problem, the objective function is to minimize response time and maximize throughput for the applications in a given workload (Table 1.1).

Algorithm 1 shows the pseudocode for the ReSensor$_{\text{Generic}}$ thread-mapping algorithm. The algorithm maps the threads from the multi-threaded applications in the given workload WL using the sensitivity scores of the applications on platform P. The algorithm stores the total number of multi-threaded applications and the applications in $nApps$ and $[Apps]$ variable, respectively (line 2-3).

The algorithm considers each targeted resource R on platform P, for which each multi-threaded application has been characterized. It counts the number of targeted resources,

Algorithm 1 ReSensor$_{\text{Generic}}$ Thread-mapping Algorithm: Mapping application threads to optimize an objective function

```
1:  INPUT: Workload WL, Topology of the experimental platform P, Sensitivity scores of
    the applications in WL on P
2:  nApps ← number of multi-threaded applications in WL
3:  [Apps] ← multi-threaded applications in WL
4:  for each resource R on P do
5:     NR ← number of R
6:     [C+] ← set of cores that use or share the same R on P
7:     [C−] ← set of cores that do not use or share the same R on P
8:     [SS] ← SensitivityScore of the applications in [Apps] for R
9:     sort [SS] array in descending order of the magnitude of the SensitivityScore and
    re-arrange [Apps] accordingly
10:    if NR >= nApps then
11:       /* Scenario 1: equal or more resources than the number of applications */
12:       if there is a special case then /* Special Case */
13:          for ( i = 0 ; i < nApps ; i++ ) do
14:             map Apps[i]-threads according to its special characteristics
15:          end for
16:       else /* General Case */
17:          for ( i = 0 ; i < nApps ; i++ ) do
18:             if SS[i] > 0 AND [C+] has available core(s) then
19:                map Apps[i]-threads on the available cores from [C+]
20:             else if SS[i] < 0 AND [C−] has available core(s) then
21:                map Apps[i]-threads on the available cores from [C−]
22:             else /* [C+] or [C−] does not have available core(s) */
23:                map Apps[i]-threads on any core on P
24:             end if
25:          end for
26:       end if
```

NR (line 5). It computes two arrays from the topology of the platform P: $[C_+]$, the set of cores that share or use the same R, and $[C_-]$, the set of cores that do not share the same R (lines 6, 7). These two arrays are later used to look up the cores on which the threads will be mapped. The algorithm saves the sensitivity scores of the applications in the $[SS]$ array (line 8). If WL has only one application, then its SensitivityScore is used directly to choose the mapping that optimizes the objective function. If WL has multiple applications, then the sensitivity scores of all applications are used to identify the more sensitive applications for the resource R when determining the mappings of the application threads. Therefore,

the algorithm sorts the $[SS]$ array according to the magnitude of the SensitivityScore in descending order so that the more sensitive applications are placed at the beginning of the array and get mapped earlier than the less sensitive applications (line 9). The $[Apps]$ array is reorganized accordingly.

Depending on the number of applications in WL and number of targeted resources NR, there are two scenarios.

Scenario 1: There are same or more targeted resources on P than the number of applications in the workload at a particular time. In this scenario, there can be some *problem-specific* special cases that depend on application characteristics. The algorithm considers and handles these special cases first (line 12 - 15). If there is no such special case, it considers the sensitivity scores to map the application threads for the general cases (line 17 - 25). As the platform has enough targeted resources such that each application can use separate resources, the thread-mapping algorithm considers *one application* at a time from the $[Apps]$ array and maps them according to the sign of its SensitivityScore. If the SensitivityScore is positive, then it maps the application threads on the cores from $[C_+]$ (line 19) because this mapping configuration improves the application's characterization metric and consequently, optimizes the objective function. If the SensitivityScore is negative, then it maps the application threads on the cores from $[C_-]$ (line 21) because this mapping configuration improves the application's characterization metric, which eventually contributes to optimized objective function. If there is no core available, then the remaining application threads are mapped on any cores (line 23). These remaining threads are from the less-sensitive applications, so the arbitrary mapping would not significantly impact the objective function.

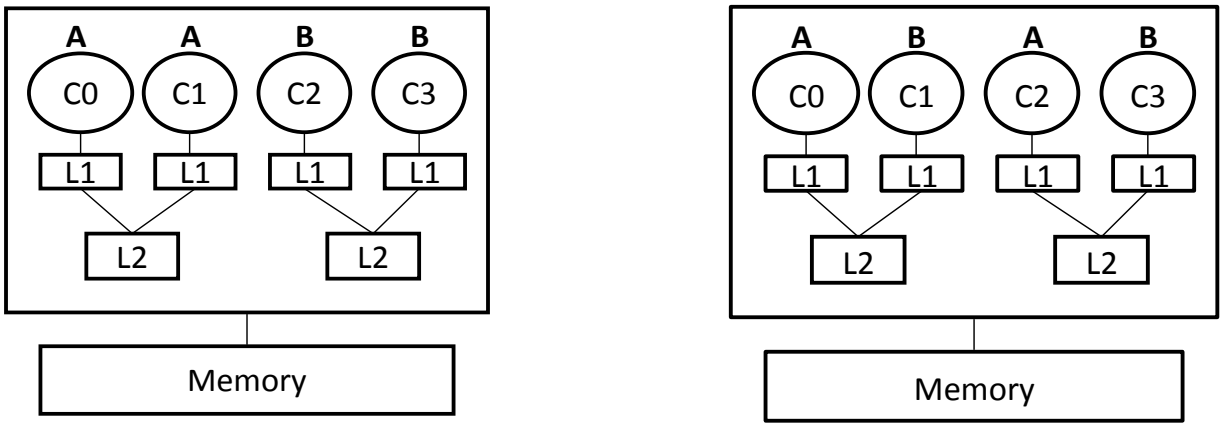
Scenario 2: There are a fewer number of targeted resources on P than the number of applications in the workload at a particular time. In this scenario, there can be also some problem-specific special cases that are handled by the algorithm first (line 28 - 31). If there is no such special case, the algorithm considers the sensitivity scores to map the application threads (line 33 - 41). As the platform does *not* have enough targeted resources such that

Algorithm 1 ReSensor$_{\text{Generic}}$ Algorithm: Continued

```
27:    else /* Scenario 2: fewer resources than the number of applications */
28:      if there is a special case then /* Special Case */
29:        for ( i = 0 ; i < nApps ; i++ ) do
30:          map Apps[i]-threads according to its special characteristics
31:        end for
32:      else/* General Case */
33:        for ( i = 0 ; i < nApps / 2 ; i++ ) do
34:          if SS[i] > 0 AND [C+] has available core(s) then
35:            map Apps[i]- and Apps[nApps − i − 1]-threads on the available [C+]-cores
36:          else if SS[i] < 0 AND [C−] has available core(s) then
37:            map Apps[i]- and Apps[nApps − i − 1]-threads on the available [C−]-cores
38:          else/* [C+] or [C−] does not have available core(s) */
39:            map Apps[i]- and Apps[nApps − i − 1]-threads on any core on P
40:          end if
41:        end for
42:      end if
43:    end if
44: end for
```

each application can use separate resources, the thread-mapping algorithm maps *multiple applications* to use the same targeted resource, prioritizing the more-sensitive applications' behaviors. The more-sensitive application is prioritized because it has a higher impact on the objective function. The magnitude of the SensitivityScore represents an application's sensitivity for a particular resource-sharing and the extent of how the application is benefited or penalized from certain thread-mappings. Therefore, the prioritization is determined considering the magnitudes of the applications' sensitivity scores.

As $[SS]$ array is sorted in the descending order, the algorithm maps the most-sensitive (highest magnitude) application from the first half of $[Apps]$ with the least-sensitive ones (lowest magnitude) from the second half of $[Apps]$, prioritizing the characteristics of the most-sensitive application. If the SensitivityScore of the most-sensitive application is positive, it maps its threads and least-sensitive application threads to the available cores from $[C_+]$ (line 35). This mapping prioritizes the characteristics of the most-sensitive application for R. If the SensitivityScore of the most-sensitive application is negative, the algorithm maps its threads and the least-sensitive application threads to the available cores from $[C_-]$ (line 37).



(a) Contention Mitigation

(b) Temperature Management

Figure 3.3: Mapping decision for two problems

If there are no available cores from $[C_+]$ or $[C_-]$, the algorithm maps threads on any core on P (line 39). The mapping prioritization towards the most-sensitive application does not affect the least-sensitive application's execution significantly and results into overall improvement of the objective function for the workload. The algorithm terminates when there are no applications left in the workload whose thread-mappings are not determined.

For example, consider cache contention as the targeted problem and a workload with two multi-threaded applications, A and B. These application threads are to be mapped on a quad-core platform, shown in Figure 3.3. The applications are characterized according to the general methodology using L2-cache as the targeted resource. Assume that application A has a positive SensitivityScore of $+a_P$ for L2-cache, which means the application has data-sharing in the caches. Application B has a negative SensitivityScore of $-b_P$, which means it has contentious behavior for the shared L2-cache on the same platform. Here, $[C_+]$=[{C0, C1}, {C2, C3}] and $[C_-]$=[{C0, C2}, {C1, C3}]. After sorting, if $|a_P| > |b_P|$, the algorithm maps A's threads on the cores from $[C_+]$ (C0 and C1) to take advantage of the sharing characteristics and B's threads on the available cores from $[C_-]$ (C2 and C3), as shown in Figure 3.3(a). This mapping may degrade B's performance as it forces B's contentious threads to use the same cache. Because B is comparatively less sensitive for L2-cache, the

degradation is less significant than A's degradation if the alternate thread-mapping is selected. As the algorithm prioritizes A's characteristics, A has better performance using this mapping, which results in response time and throughput improvements of the workload.

On the other hand, if $|a_P| < |b_P|$, the algorithm maps B's threads on the cores from $[C_-]$ (C0 and C2) to mitigate L2-cache contention and A's threads on the available cores from $[C_+]$ (C1 and C3), as shown in Figure 3.3(b). This mapping may degrade A's performance because the data-sharing application threads are mapped on different caches. However, application B's cache contentiousness is prioritized over A because B is more sensitive to L2-cache usage, resulting in mitigated contention and improved response time and throughput of the workload.

For the same two applications, consider the thermal management problem. Let us assume that application A has a negative SensitivityScore of $-a_T$ for temperature, which means the application uses the computational resources highly and increases the temperature of the processors on the same die in the sharing characterization configuration. Application B has a positive SensitivityScore of $+b_T$ for temperature, which means the application uses computational resources less aggressively and decreases the processor temperature in the sharing characterization configuration. Comparing the magnitudes of the sensitivity scores, if $|a_T| > |b_T|$, the algorithm maps the application threads such that the computation-intensive threads from A use two separate dies so that the computational resource usage is evenly distributed, and thermal emergencies are avoided. The mapping is shown in Figure 3.3(b).

3.3.2 Run-time System

The run-time system of the framework manages application execution by employing the thread-mapping algorithm that uses the sensitivity scores of the applications in a workload to optimize the objective function.

Algorithm 2 shows the pseudocode of the run-time system. The run-time system executes an infinite loop and detects any change in the total number of threads in the workload

Algorithm 2 Run-time System

1: INPUT: Dynamic workload WL, Sensitivity scores of the applications in WL on P
2: **while** (1) **do**
3: **if** there is any change in total number of threads in WL **then**
4: invoke problem-specific instantiation of $\text{ReSensor}_{\text{Generic}}$ with WL and Sensitivity scores of the applications in WL
5: **end if**
6: **end while**

WL, i.e., whenever a multi-threaded application creates and destroys a worker thread or an application starts or terminates its execution (line 2, 3). If there is any change in the number of threads, the run-time system employs the problem-specific instantiation of $\text{ReSensor}_{\text{Generic}}$ and sensitivity scores to determine the mappings of the currently executing threads from WL on the appropriate cores (line 4). This run-time system works in the same way for different instances of the framework. Only the thread-mapping algorithm and the sensitivity scores are different in the instances depending on the targeted problem and objective function.

Figure 3.4 shows an operational overview of the ReSense framework. Consider a workload consisting of n multi-threaded applications, $\{A_1, A_2, ..., A_n\}$ arriving (represented by + sign) at time $\{t_1, t_2, ..., t_n\}$, respectively on platform P. The offline *characterization* phase uses the general methodology and characterization metric to identify the potential behaviors of each multi-threaded application and determines each application's sensitivity scores (SS_A_1, SS_A_2, ..., SS_A_n) for the resources on platform P. This characterization is done for each application in isolation and consequently needs to be done *only once* for a particular resource on the targeted platform.

In the online *mapping* phase, the run-time system employs the thread-mapping algorithm that determines the thread-to-core mappings of the multi-threaded applications using the sensitivity scores for each application on P. The run-time system invokes this algorithm when there is a change in the number of threads or applications in the system. That is, as the execution of each application *starts* or *terminates* or any thread is *created* or *destroyed*, the run-time system dynamically adjusts the thread-mappings of the applications. For example

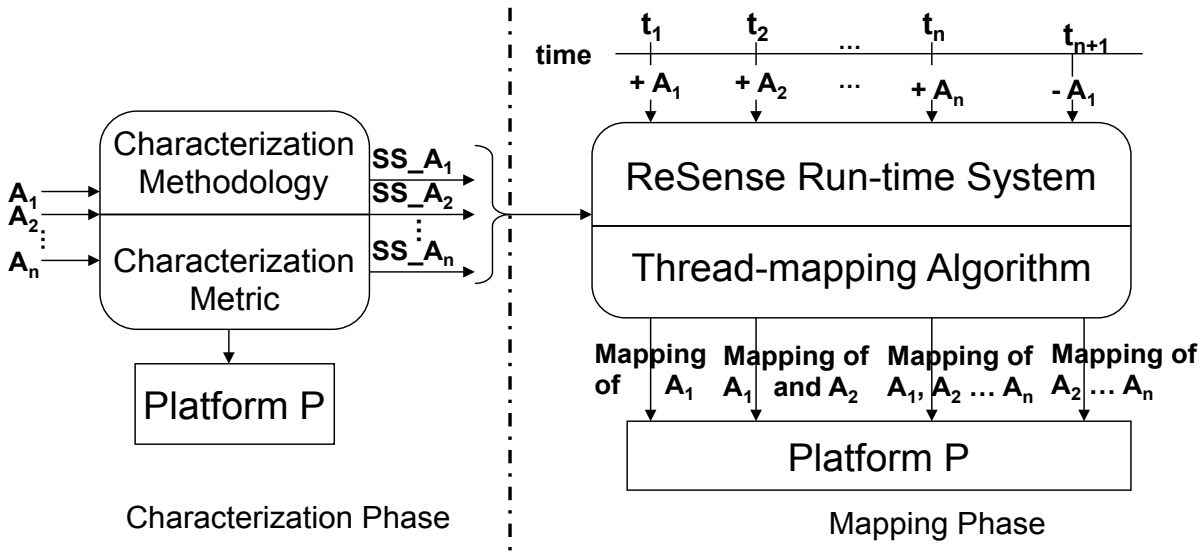


Figure 3.4: An operational overview of ReSense

in Figure 3.4, at time t_1, there is only one application A_1, and the run-time system maps A_1's threads on P. At time t_2, A_2 starts execution, and the run-time system determines and adjusts the thread mappings of A_1 and A_2 dynamically. At time t_n, there are n multi-threaded applications running, and the run-time system maps all n applications on P using the mapping algorithm. If at time t_{n+1} any application A_i (e.g., A_1 in Figure 3.4) terminates, the run-time system adjusts the mappings of the remaining executing applications. Thus, the run-time system of the ReSense framework continually manages the execution of the applications in the workload using the corresponding thread-mapping algorithm and sensitivity scores to optimize the objective function for addressing the targeted problem.

This chapter concludes the general description of the ReSense framework. In the following chapters, we describe the instances that are developed using the framework to address two targeted problems on multicore architectures: shared-resource contention and soft errors.

Chapter 4

Using ReSense for Performance

This chapter describes how the ReSense framework is used to develop the performance instance, ReSense_Performance, to optimize a workload's performance on modern multicore platforms. ReSense_Performance addresses the challenges of mitigating resource contention for the shared resources in the memory hierarchy on a given CMP platform and targets obtaining scalable performance.

4.1 Introduction

With the continuous growth of the number of cores on modern CMPs, the number of simultaneously executing multi-threaded applications is increasing to utilize the multiple execution cores. When there are multiple applications executing on CMPs, there is contention among the applications for the shared resources on the targeted platform. In particular, contention for the shared resources in the memory hierarchy can dramatically impact the performance of applications, as shown in several recent studies [4, 10, 6, 59, 27]. To utilize these resources to their full potential and obtain scalable performance improvements on CMPs, it is critical to determine intelligent techniques to optimize performance [64, 4].

A number of techniques have been proposed to address the shared-resource contention

problem for *single-threaded* applications and *static* workloads[2] via thread-mapping and scheduling [57, 59, 6, 27, 4, 10]. However, there are several differences between the mapping of single- and multi-threaded applications when they contend for shared resources, as mentioned in Chapter 1. Existing techniques to address resource contention for single-threaded applications do not consider these differences and are thus not applicable for mapping multi-threaded applications.

For multi-threaded applications, there are two categories of contention for shared resources. We define *intra-application* contention as the contention for a resource among sibling threads when the application runs solely (without co-runners). In this situation, the application threads compete with each other for the shared resources. We define *inter-application* contention as the contention for shared resources among threads from different applications. In this case, threads from one multi-threaded application compete for the shared resources with the threads from its co-running multi- or single-threaded application(s). Both types of contention can severely degrade a multi-threaded application's performance [32, 4].

There are several challenges to effectively map multi-threaded applications and mitigate shared-resource contention on CMPs. For workloads with multiple applications, the most effective thread-mapping, which minimizes contention in the shared resources, depends on an application's behaviors, underlying resource topology of the platform, and the behaviors of the co-running applications. One approach is to develop a thread-mapping algorithm that detects and mitigates contention online in the shared resources created by co-running applications. Online contention detection involves performance comparison of different thread-to-core-mapping configurations, which vary the contention for the shared resources. Mitigation involves mapping the applications in the thread-mapping configuration that ensures the lowest contention and performance degradation [33, 20]. However, as the multi-threaded applications in a workload can create a varying number of threads, the number of thread-to-core-mapping configurations can increase exponentially [21]. As a result, determining the thread mapping

[2]A static workloads is one in which all applications start execution at the same time, and the set of simultaneously executing applications does not change during execution.

that minimizes contention by online contention detection in all possible thread-mapping configurations has exponential complexity and makes mapping threads optimally to mitigate contention an NP-complete problem [59].

Another issue arises when thread-mapping algorithms consider multi-threaded applications in realistic *dynamic* workloads, where any number of multi-threaded applications arrive, execute and terminate in unpredictable ways. Online detection and minimization of the contention created by dynamic workloads is very challenging because of the continuous change in the total number of applications and the intensity of contention in the execution environment, resulting in exponentially varying numbers of thread-mapping configurations.

Another approach and the one used in this research to mitigate shared-resource contention is to first determine the inherent characteristics and potential behaviors of each multi-threaded application in a workload for how it creates and suffers from the contention on the underlying platform, using an offline technique, and then develop a thread-mapping algorithm to mitigate the shared-resource contention. This approach leads to the creation of ReSense_Performance, the *performance instance* of the ReSense framework.

ReSense_Performance addresses the challenges of mapping multiple multi-threaded applications and mitigating contention for the shared resources in the memory hierarchy. Figure 4.1 shows the components of ReSense_Performance. The characterization phase of ReSense_Performance instantiates the general methodology of the ReSense framework to characterize a multi-threaded application based on its contentiousness in the targeted shared resources. It uses performance as the characterization metric and calculates $\text{SensitivityScores}_{\text{performance}}$ of the applications in a workload for each shared resource in the memory hierarchy.

The general characterization methodology of the framework is instantiated to determine the performance implications of multi-threaded applications in a workload and characterize them with respect to both intra- and inter-application contention. This characterization process measures the potential impact of contention for a specific shared resource in the

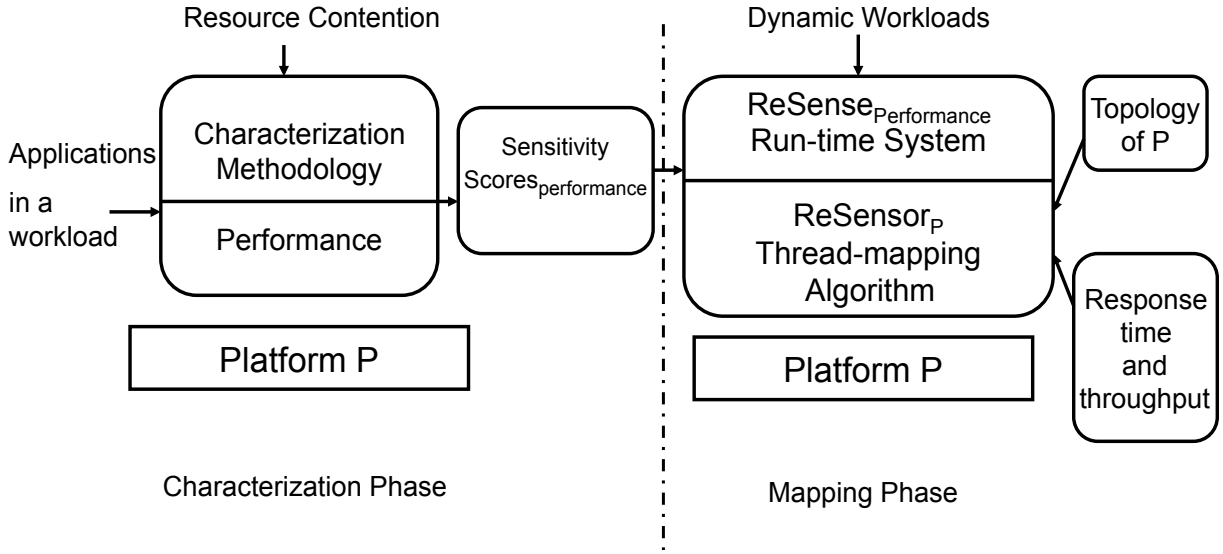


Figure 4.1: Components of the ReSense_Performance Instance

memory hierarchy, including shared caches, last-level caches (LLC), front-side bus (FSB), on-chip memory controller, and memory-socket interconnection, on a multi-threaded application's performance. This characterization methodology can also be used as a stand-alone technique to determine the contentious behaviors of any multi-threaded application for any targeted resource.

A $\text{SensitivityScore}_{\text{performance}}$ represents a multi-threaded application's contentious behavior for a particular shared resource and is determined offline *only once* for each targeted platform. The characterization is performed by running the application by itself based on intra-application contention on a particular CMP, which keeps the number of characterizations in linear order of the total number of multi-threaded applications in a workload. A $\text{SensitivityScore}_{\text{performance}}$ identifies different behaviors (e.g., data sharing or contentiousness) of a multi-threaded application and is precise enough to evaluate the relative importance of a shared resource for an application. The scores are used to compare the contentiousness among co-running applications, as well as the contentiousness among sibling threads of the co-running applications.

In the mapping phase of ReSense_Performance, we instantiate the ReSense run-time

system as $\text{ReSense}_{\text{Performance}}$ (Figure 4.1). $\text{ReSense}_{\text{Performance}}$ is capable of handling dynamic workloads and mapping any number of threads from any number of applications arriving and terminating non-deterministically, avoiding the performance overhead of using online contention detection mechanisms. The $\text{ReSense}_{\text{Performance}}$ run-time system applies the $\text{ReSensor}_{\text{P}}$ thread-mapping algorithm to dynamically map application threads whenever the number of thread changes in the input dynamic workload. The $\text{ReSensor}_{\text{P}}$ algorithm uses the $\text{SensitivityScore}_{\text{performance}}$ of each application in a workload to determine the thread-mappings of the multi-threaded applications in the presence of any number of co-runners. It optimizes the objective function of the performance instance, which is to minimize the workload's response time and maximize throughput by mitigating contention for the shared-memory resources.

The outline of this chapter is as follows: Section 4.2 describes the characterization phase of the instance including the methodology to characterize any multi-threaded application for intra- and inter-application contention for the shared resources in the memory hierarchy. Section 4.3 describes the mapping phase, which includes the $\text{ReSensor}_{\text{P}}$ algorithm and the $\text{ReSense}_{\text{Performance}}$ system. Section 4.4.1 describes the experiments performed to characterize the PARSEC and NBP benchmarks for shared-resource contention in the memory hierarchy using four different CMP platforms to demonstrate the characterization methodology. Section 4.4.2 summarizes the characterization results. Section 4.4.3 describes experimental methodology, evaluation metrics and discusses the evaluation results of the mapping phase. Section 4.5 concludes the chapter.

4.2 Characterization for Shared-Resource Contention

In this section, we describe how we apply the general methodology of the framework to characterize multi-threaded applications for shared-resource contention in the memory hierarchy and determine $\text{SensitivityScores}_{\text{performance}}$ of the applications.

Multi-threaded applications demonstrate different behaviors for different resource usage

on CMPs, which we can determine in the characterization phase. In this work, we consider the shared resources in the memory hierarchy, e.g., shared caches, LLC, memory controller, front-side bus (FSB), and memory socket connection.

For shared caches, multi-threaded applications can show sharing and contentious characteristics. The sibling threads of a multi-threaded application typically use the same input data. When the threads have such data sharing, if one of the sibling threads loads the data from the memory into a shared cache, the sibling threads can directly use the data without suffering from the penalty of a cache miss.

On the other hand, sibling threads can demonstrate contentious characteristics when the working set too large to fit into the same cache and the threads use different data. For such scenario, one of the sibling threads replaces the cache-lines populated by another thread and causes cache misses. We can determine these characteristics and resource usage behaviors of an application by comparing its performance when its threads share the same cache relative to when they do not. If the performance of the application improves as it shares the same cache, we characterize the application to have data sharing among its threads. If the performance degrades, we characterize the threads to have contention for the shared cache.

For a memory controller or bus, multi-threaded applications can have different bandwidth requirement, which can be determined from its usage behavior of memory controller or memory socket connection. We can compare an application's performance when its threads use more bandwidth relative to when it uses less bandwidth. If the performance of the application improves as it uses more bandwidth, we infer that the application is memory intensive and has high bandwidth requirement. If the performance does not change significantly, we infer that the application is not memory-intensive and has low bandwidth requirement.

4.2.1 Characterization Metric

As performance is a direct measure of contention [27], we use performance as the characterization metric to determine a multi-threaded application's contentious behaviors for shared

resources. Because contention for a shared resource degrades a multi-threaded application's performance, this characterization metric also represents and quantifies the effect of the targeted problem for this instance.

4.2.2 Characterization Methodology

We instantiate the general characterization methodology of the ReSense framework to identify the application characteristics for resource contention. We customize the methodology to characterize a multi-threaded application by determining the effect of intra- and inter-application contention for the shared resources on application performance.

4.2.2.1 Characterization for Intra-application Contention

To characterize a multi-threaded application based on intra-application contention for a targeted shared resource, we need to analyze how sharing the targeted resource among threads from the same multi-threaded application affects its performance, compared to when they do not share. To accomplish this measurement, the application is run solely with at least two threads in two characterization configurations according to the general methodology. The *non-sharing* configuration places the threads on the cores such that the threads do not share the targeted resource and run using two separate dedicated resources. The *sharing* configuration places the application threads on the cores such that the threads do share the targeted resource and execute while using the same resource. Because the sharing configuration maps the threads to use the same resource, it creates the possibility that the threads compete with each other for that resource causing intra-application contention that degrades the application's performance. In both configurations, the mapping of threads keeps the effect on the other resources the same. For example, if we characterize an application for intra-application L1-cache contention, the placement of threads in both characterization configurations must maintain the same effect on the rest of the memory hierarchy, including L2/L3-cache and the memory controller or memory socket connection. Application performance is measured in both characterization configurations.

When we compare the performances of the two configurations, the difference indicates the effect of contention for the targeted resource on the application's performance. If the performance difference between the non-sharing and sharing configuration is negative (degradation), then the application is characterized to have intra-application contention for that resource among the sibling threads. However, if the performance difference is positive (improvement), the application is characterized to have data sharing among the sibling threads and *not* have intra-application contention.

4.2.2.2 Characterization for Inter-application Contention

To characterize a multi-threaded application based on inter-application contention for a targeted shared resource, we need to determine and analyze the effect of sharing the targeted resource among threads from different applications on the application performance. To accomplish this analysis, multi-threaded applications are run with a co-runner, which can be another multi- or single-threaded application. A pair of multi-threaded applications are run in two characterization configurations. The *non-sharing* configuration places the application threads on the cores such that each application has an exclusive access to the targeted resource. In this configuration, the applications do not share the targeted resource and there is no interference or contention for that resource from the co-running application. The *sharing* configuration places the application threads on the cores such that threads from one application share the targeted resource with the co-runner's thread creating the possibility of inter-application contention. Similar to the intra-application contention, both configurations place the application threads such that the placement of threads to the other shared resources remains the same and the effect of contention for the targeted resource can be precisely determined. For example, if we characterize an application for inter-application L2-cache contention, the mapping of the threads in both configurations should maintain the same effect on L1-cache and the memory controller or memory socket connection. Application performance is measured in both characterization configurations.

When we compare the performances of both configurations, the difference indicates

the effect of contention for the targeted resource on each application's performance. If the performance difference between the non-sharing and sharing configuration is negative (degradation), then the application is characterized to have inter-application contention for that resource with the co-runner's threads. However, if the performance difference is positive (improvement), the application is *not* characterized to have inter-application contention caused by the co-runner's threads.

4.2.3 SensitivityScore$_{\text{performance}}$: Sensitivity Scores for Performance

According to the methodology, the difference between the characterization metric, which is performance in this instance, is used to compute a sensitivity score. This sensitivity score of a multi-threaded application is represented as a SensitivityScore$_{\text{performance}}$ for each shared resource in the memory hierarchy on a particular CMP. These sensitivity scores are calculated using the following equation:

$$\text{SensitivityScore}_{\text{performance}} =$$

$$\frac{(NumberOfCycles_{non-sharing} - NumberOfCycles_{sharing}) * 100}{NumberOfCycles_{non-sharing}} \tag{4.1}$$

Here, $NumberOfCycles_{non-sharing}$ and $NumberOfCycles_{sharing}$ are the total number of cycles (by reading hardware performance counter) in the non-sharing and sharing configuration, respectively. The number of cycles represents an application's performance.

For the performance instance, SensitivityScores$_{\text{performance}}$ capture the contentious characteristics of a multi-threaded application for shared caches and memory bandwidth and how application performance is affected by shared-memory resource contention. The score is represented as a floating-point number, which has both a sign and magnitude. The sign indicates whether the application's performance improves (positive sign) or degrades (negative sign) as its threads share a particular resource.

For example, *canneal* has a positive SensitivityScore$_{\text{performance}}$ for the L2-cache on Intel-Yorkfield (see Table 4.6) indicating that *canneal*'s performance improves in the sharing

configuration. This improvement is because of the data-sharing among the threads and fewer number of coherency-based memory transactions when its threads share the same L2-cache. *Streamcluster* has a negative $\text{SensitivityScore}_{\text{performance}}$ for FSB on Intel-Harpertown (see Table 4.6), which indicates its performance degrades when its threads are mapped to use the same FSB and the application requires more FSB bandwidth. *Streamcluster* requires more FSB bandwidth because it is a streaming application, which accesses many consecutive memory locations and its performance degrades when we use the thread-mapping that reduces the bus bandwidth. Therefore, from the sign of the sensitivity score, we can identify the key characteristics of an application as to whether it benefits from certain resource-sharing.

On the other hand, the magnitude indicates the degree of application's sensitivity for a specific shared resource. The higher the magnitude, the more sensitive the application is to sharing that resource. For example, *canneal* has a higher magnitude of $\text{SensitivityScore}_{\text{performance}}$ and is more sensitive to L2-cache sharing than *dedup* because *canneal* accesses more shared data. From the magnitude of the sensitivity score, we determine how much an application benefits or is penalized from certain resource-sharing.

The targeted multicore platform can have shared resources at multiple levels of the memory hierarchy. A multi-threaded application is characterized and its $\text{SensitivityScore}_{\text{performance}}$ is determined for each shared resource in the memory hierarchy applying the characterization methodology. These scores are stored in $SV_{performance}$. A $SV_{performance}$ of a multi-threaded application is a vector containing the $\text{SensitivityScores}_{\text{performance}}$ of the applications for each shared resource on a platform. For example, if the platform has N types of shared resources, then $SV_{performance}$ is an N-element vector for each multi-threaded application in a workload. This vector is used as an input to the $\text{ReSensor}_{\text{P}}$ thread-mapping algorithm.

4.3 Mapping Co-located Multi-threaded Applications for Performance

We describe the mapping phase of the performance instance of ReSense, which includes the $\text{ReSensor}_{\text{P}}$ algorithm and the $\text{ReSense}_{\text{Performance}}$ run-time system.

4.3.1 The $\text{ReSensor}_{\text{P}}$ Thread-mapping Algorithm

To optimize the performance of a workload and mitigate contention for the shared resources, it is critical to determine the thread-mapping, considering the characteristics of the applications and the underlying architecture of the platform. Existing thread-mapping techniques map an application by considering its characteristics in the presence of a co-runner [10] [4]. On the other hand, $\text{ReSensor}_{\text{P}}$ determines the thread-mappings of a multi-threaded application in the presence of *any* co-runner(s) by utilizing the characteristics determined without considering the presence of the co-runner(s). As $\text{SensitivityScores}_{\text{performance}}$ identify the key characteristics of multi-threaded applications for the shared resources on a particular platform, these scores are used to determine the effective thread-mappings of each application in a workload to optimize the objective function of this instance, i.e., minimize response time and maximize throughput by mitigating contention.

Algorithm 3 contains the pseudocode of the $\text{ReSensor}_{\text{P}}$ algorithm. The $\text{ReSensor}_{\text{P}}$ algorithm is instantiated from the $\text{ReSensor}_{\text{Generic}}$ thread-mapping algorithm of the framework, where applications' $\text{SensitivityScores}_{\text{performance}}$ are used as the sensitivity scores. The algorithm maps threads from a workload WL consisting of any number of multi-threaded applications on a particular platform P. Platform P can have shared resources in multiple levels of the memory hierarchy. In most platforms, a resource lower in the memory hierarchy contains multiple number of resources that are at higher level in the memory hierarchy. Once the mapping with respect to the resources lower in the memory hierarchy is determined, the mapping with respect to resources at a higher memory hierarchy can be easily determined

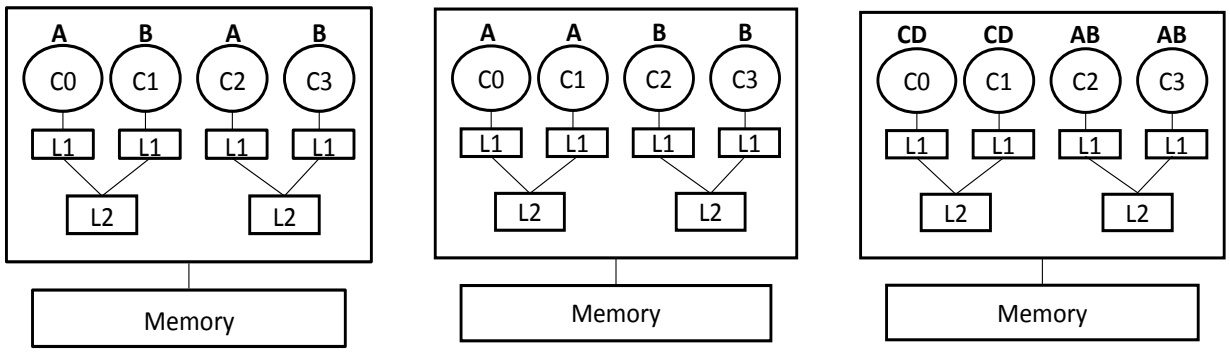
Algorithm 3 The ReSensor$_{\mathrm{P}}$ Algorithm: Mapping application threads to mitigate contention for the shared resources in the memory hierarchy

INPUT: Workload WL, Topology of the experimental platform P, Sensitivity vector $SV_{performance}$ of the applications on P
$nApps \leftarrow$ number of multi-threaded applications in WL
$[Apps] \leftarrow$ multi-threaded applications in WL
for each level MHL in the memory hierarchy of P **do**
  $R \leftarrow$ shared resource at MHL
  $NR \leftarrow$ number of R at MHL
  $[C_+] \leftarrow$ set of cores that use or share the same R on P
  $[C_-] \leftarrow$ set of cores that do not use or share the same R on P
  $[SS_P] \leftarrow SV_{performance}[R]$ of the applications in $[Apps]$
  sort $[SS_P]$ array in descending order of the magnitude of the SensitivityScore$_{\mathrm{performance}}$ and re-arrange $[Apps]$ accordingly
  if $NR >= nApps$ **then**
    /* Scenario 1: equal or more shared resources than the number of applications */
    for (i = 0 ; i < $nApps$; i++) **do**
      if $SS_P[i] > 0$ AND $[C_+]$ has available core(s) **then**
        map $Apps[i]$-threads on the available cores from $[C_+]$
      else if $SS_P[i] < 0$ AND $[C_-]$ has available core(s) **then**
        map $Apps[i]$-threads on the available cores from $[C_-]$
      else
        /* $[C_+]$ or $[C_-]$ does not have available core(s) */
        map $Apps[i]$-threads on any core on P
      end if
    end for
  else
    /* Scenario 2: fewer shared resources than the number of applications */
    for (i = 0 ; i < $nApps$ / 2 ; i++) **do**
      if $SS_P[i] > 0$ AND $[C_+]$ has available core(s) **then**
        map $Apps[i]$- and $Apps[nApps - i - 1]$-threads on the available cores from $[C_+]$
      else if $SS_P[i] < 0$ AND $[C_-]$ has available core(s) **then**
        map $Apps[i]$- and $Apps[nApps - i - 1]$-threads on the available cores from $[C_-]$
      else
        /* $[C_+]$ or $[C_-]$ does not have available core(s) */
        map $Apps[i]$- and $Apps[nApps - i - 1]$-threads on any core on P
      end if
    end for
  end if
end for

because the number of thread-mapping configurations reduces to half. In addition, if the algorithm considers the resources from the top of the memory hierarchy to the bottom, the mapping determined on the basis of resource at a higher level can violate the mapping that will be determined based on the characterization of the shared resource in the lower level. Therefore, ReSensor_P considers the shared resources from the bottom to the top of the memory hierarchy, i.e., from memory bus or memory controller to the shared caches, to determine the thread-mappings.

For example, Intel-Harpertown (shown in Figure 4.3(b)) has four L2-caches and two FSB-connections as the shared resources. Assume we have to map an application A with two threads on this platform. A has a negative $\text{SensitivityScore}_\text{performance}$ for L2-cache and a positive $\text{SensitivityScore}_\text{performance}$ for FSB. From the $\text{SensitivityScores}_\text{performance}$, we can conclude that A's performance improves when A is mapped on the the cores that use the same FSB and separate L2-caches. If ReSensor_P determines the thread-mapping of the application by considering the characteristic of the L2-cache (higher at the memory hierarchy) first, then A can be mapped on the cores that use separate cache, any one from the 24 possible thread-mapping configurations. If the mapping $\{C0, C1\}$ is chosen, then this mapping uses separate cache to avoid cache contention. But at the same time, this mapping causes A to use separate FSB-connections, which violates the mapping that leverages A's FSB characterization to use the same FSB. Therefore, the mapping $\{C0, C1\}$ can degrade A's performance. On the other hand, if ReSensor_P determines the thread-mapping of the applications by considering the characteristic of FSB (lower in the memory hierarchy) first, then A can be mapped on cores that use same FSB-connection, which reduce the number of mapping configurations for L2-cache to 8. Therefore, ReSensor_P considers each shared resource R from the bottom of the memory hierarchy to the top (line 4). For the Intel-Harpertown example, ReSensor_P considers the FSB first then L2-cache characteristics to determine the final thread-mapping.

Next, ReSensor_P counts the number of shared resources, NR, at each memory hierarchical level (MHL) (line 6). It computes two arrays: the set of cores that share or use the same R,



(a) Scenario 1: Example 1 (b) Scenario 1: Example 2 (c) Scenario 2

Figure 4.2: Mapping decision for the two scenarios

$[C_+]$, and the set of cores that do not share the same R, $[C_-]$ (lines 7, 8). These two arrays are later used to determine the cores on which the threads will be mapped. It collects the $\text{SensitivityScores}_{\text{performance}}$ of the applications for R into the $[SS_P]$ array (line 9). According to $\text{ReSensor}_{\text{Generic}}$, to optimize the objective function, the application that has the highest sensitivity for R should be prioritized and should have its thread-mapping earlier than the least-sensitive applications. To ensure the prioritization, the algorithm sorts the $[SS_P]$ array in the descending order of the $\text{SensitivityScore}_{\text{performance}}$'s magnitude and re-arranges the applications in the $[Apps]$ array accordingly (line 10).

Depending on the number of applications in a workload and number of shared resources at a particular memory hierarchical level on a platform, there are two scenarios as the $\text{ReSensor}_{\text{Generic}}$ algorithm. The scenarios and the corresponding actions of the algorithm for this performance instance are described as follows.

Scenario 1: There is the same or more shared resources than the number of applications in the workload (line 11). In this scenario, P has enough resources to be allocated to each application for isolated execution. There is no problem-specific special case to consider for the algorithm. Therefore, $\text{ReSensor}_{\text{P}}$ determines the thread-mappings of each application considering the sign of the $\text{SensitivityScore}_{\text{performance}}$ for the shared resource at that level. The sign of the $\text{SensitivityScore}_{\text{performance}}$ determines if sharing R improves the application performance, i.e., response time and throughput. If the application's $\text{SensitivityScore}_{\text{performance}}$

is positive (line 14), it has sharing behavior and its performance improves when its threads executes while using or sharing the same resource R. ReSensor$_{\text{P}}$ maps the application threads on the available cores (the cores on which any thread is not mapped yet) from $[C_+]$ considering its sharing characteristics (line 15). Sharing the same resource, especially the caches, also reduces the number of memory transactions for maintaining the cache coherency and results in better response time and throughput. If the application's SensitivityScore$_{\text{performance}}$ is negative (line 16), its performance degrades when the application threads use the same resource R because of intra-application contention for R. Therefore, to avoid and mitigate the contention, ReSensor$_{\text{P}}$ maps the threads on the available cores from $[C_-]$ (line 17) so that the threads use separate R. If there are no available cores from $[C_+]$ or $[C_-]$, ReSensor$_{\text{P}}$ maps threads on any core on P (line 20). As ReSensor$_{\text{P}}$ maps the application threads considering its performance characterization for each shared resource, it always guarantees the mapping that improves the workload's response time and throughput.

For example, consider a workload that has two multi-threaded applications, A and B. Assume both applications have negative SensitivityScores$_{\text{performance}}$ of $-a_P$ and $-b_P$, respectively and contentious behavior for shared L2-cache on the quad-core platform (e.g., Intel-Yorkfield), shown in Figure 4.2. Here, both $nApps$ and NR equal 2 and $[C_+]$=[{C0,C1}, {C2,C3}] and $[C_-]$=[{C0,C2}, {C1,C3}]. As both applications have contentious behavior for the shared L2-cache, they are both mapped on the cores from $[C_-]$. A is mapped on $C0$ and $C2$, and B is mapped on the two remaining cores from $[C_-]$, as shown in Figure 4.2(a). These applications, with negative *sensitivity scores*, suffer from relatively higher intra-application contention than inter-application contention for cache resources [5], and the performance degrades more when the sibling threads are co-located with each other compared to when threads are co-located with the threads from the co-running application to use the same cache. Therefore, for such workloads with negative SensitivityScores$_{\text{performance}}$, it is beneficial to share the resource with the co-runner's threads than sharing the resource with its sibling threads to improve the workload's overall response time and throughput.

Let us now consider, application A that has a positive $\text{SensitivityScore}_{\text{performance}}$ $+a_P$ and sharing behavior, and application B has a negative $\text{SensitivityScore}_{\text{performance}}$ $-b_P$ and contentious behavior for the shared L2-cache on the same platform. Similar to the last example, both $nApps$ and NR equal 2 and $[C_+]$=[{C0,C1}, {C2,C3}] and $[C_-]$=[{C0,C2}, {C1,C3}]. After sorting, if $|a_P| > |b_P|$, ReSensor_P maps A's threads on the cores from $[C_+]$ (C0 and C1) to take advantage of the sharing characteristics and B's threads on the available cores from $[C_-]$ (C2 and C3), shown in Figure 4.2(b). This mapping may degrade B's performance as it forces B's contentious threads to use the same cache. Because B is comparatively less sensitive for L2-cache, the degradation is less significant than A's degradation if the opposite thread-mapping was selected. As ReSensor_P prioritizes A's characteristic, A has better response time and throughput compared to the alternative mapping when it shares the same L2-cache with B's threads, and the overall response time and throughput of the workload improves.

Scenario 2: There are more applications than the number of resources at a particular level of the memory hierarchy. There is no problem-specific special case to consider by the algorithm. In this scenario, P does *not* have enough resources to be allocated to each application for isolated execution. Therefore, ReSensor_P needs to select more than one application to use the same resource R. ReSensor_P selects the most-sensitive application with the least-sensitive application to share the same resource and chooses the mapping that benefits the most-sensitive application. ReSensor_P prioritizes the most-sensitive applications because its performance has a higher impact on the workload's overall response time and throughput than that of the least-sensitive applications. Lines 25-34 contain the pseudo-code for mapping applications in such cases. After sorting $[SS_P]$ in descending order, the algorithm maps the most-sensitive (highest magnitude) application from the first half of $[Apps]$ with the least-sensitive ones (lowest magnitude) from the second half of $[Apps]$, prioritizing the characteristics of the most-sensitive application. If the $\text{SensitivityScore}_{\text{performance}}$ of the most-sensitive application is positive (line 26), it maps its threads and least-sensitive application threads to

the available cores from $[C_+]$ (line 27) prioritizing the sharing characteristics of the most-sensitive application for R. If the $\text{SensitivityScore}_{\text{performance}}$ of the most-sensitive application is negative (line 28), $\text{ReSensor}_{\text{P}}$ maps its threads and the least-sensitive application threads to the available cores from $[C_-]$ (line 29) to avoid the intra-application contention among sibling threads of the most-sensitive applications. If there are no available cores from $[C_+]$ or $[C_-]$, $\text{ReSensor}_{\text{P}}$ maps threads on any core on P (line 32). The mapping prioritization towards the most-sensitive application does not affect the least-sensitive application's performance significantly and results into overall response time and throughput improvements of the workload.

For example, let us consider a workload that has four applications $[A,\ B,\ C,\ D]$ to be mapped on the same platform from the previous example. The $\text{SensitivityScores}_{\text{performance}}$ of the four applications for L2-cache after sorting is $[+c_P, -a_P, +b_P, -d_P]$, where $|c_P| > |a_P| > |b_P| > |d|_P$. $\text{ReSensor}_{\text{P}}$ selects the most-sensitive application C and the least-sensitive application D to map them together. As C has a positive $\text{SensitivityScore}_{\text{performance}}$, $\text{ReSensor}_{\text{P}}$ maps C and D on the cores from $[C_+]$ (C0 and C1) prioritizing C's sharing behavior for L2-cache. Then $\text{ReSensor}_{\text{P}}$ maps A and B on the remaining cores (C2 and C3). The final mapping is shown in Figure 4.2(c). Because C is the most sensitive application, this mapping prioritizes C's characteristic to ensure its improved performance. D being the least sensitive, the mapping does not degrade D's performance significantly and improves the workload's overall response time and throughput.

The algorithm terminates when there are no applications left in the workload whose thread-mappings are not determined.

4.3.2 The $\text{ReSense}_{\text{Performance}}$ Run-time System

$\text{ReSense}_{\text{Performance}}$ is instantiated from the run-time system of the ReSense framework. The $\text{ReSense}_{\text{Performance}}$ run-time system manages the execution of the applications in the input workload. Whenever an application creates a new thread or destroys an existing one,

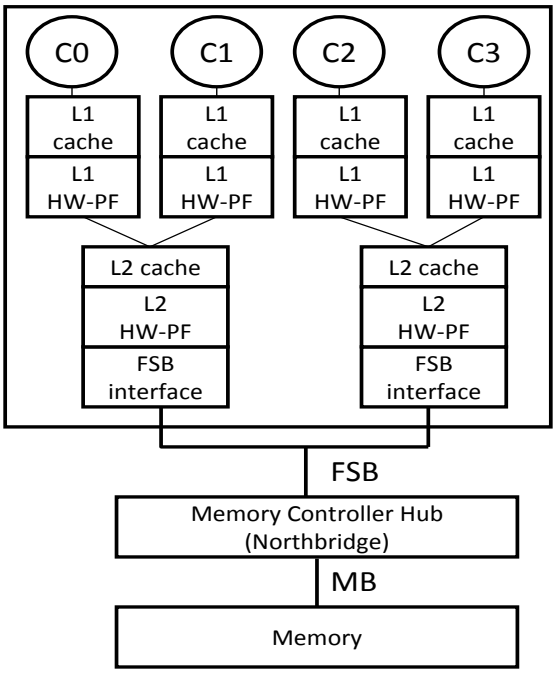
or an application starts or terminates the execution, the run-time system detects such changes and employs $\text{ReSensor}_{\text{P}}$ to determine the thread-mappings of the currently running applications. $\text{ReSense}_{\text{Performance}}$ passes the $\text{SensitivityScores}_{\text{performance}}$ of the applications in the input dynamic workload to the $\text{ReSensor}_{\text{P}}$ algorithm. The $\text{ReSensor}_{\text{P}}$ algorithm dynamically determines the thread-mappings of the multi-threaded applications in a workload in the presence of any number of co-runners using each application's $\text{SensitivityScore}_{\text{performance}}$ for a particular resource. The thread-mapping algorithm optimizes the objective function of this instance, which is to mitigate shared-resource contention in the memory hierarchy and minimize workload's response time and maximize throughput. $\text{ReSense}_{\text{Performance}}$ uses the mapping algorithm to optimize the objective function of this instance.

4.4 Evaluation of the ReSense_Performance Instance

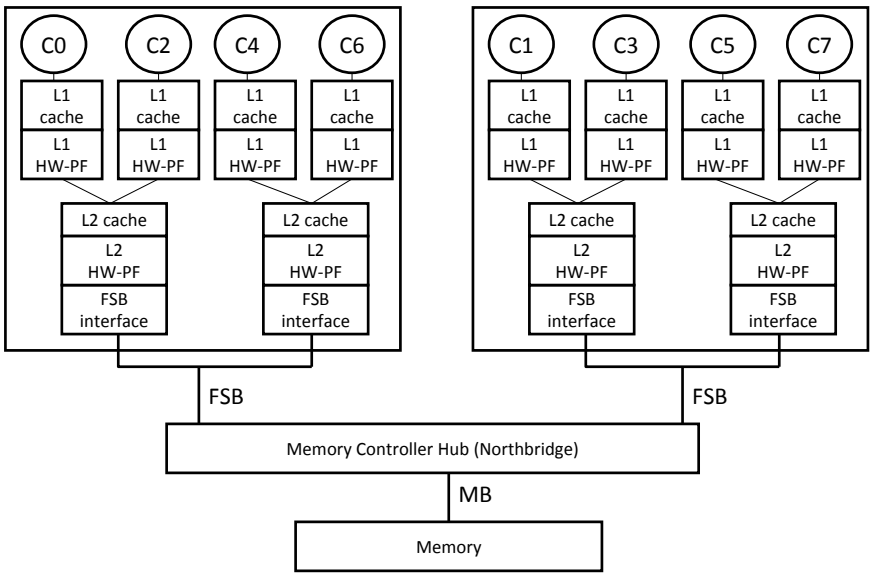
In this section, we describe the experimental results of the characterization and mapping phase for the ReSense_Performance instance of the framework. In the characterization phase, we characterize the PARSEC and NPB benchmarks using the methodology described in Section 4.2.2. We present the characterization results in Section 4.4.1 and summary in Section 4.4.2. We present the evaluation results of the mapping phase in Section 4.4.3.

4.4.1 Characterization: Experimental Details and Results

According to the methodology, to characterize a multi-threaded benchmark based on both types of contention, we perform two categories of experiments: (1) we run a benchmark solely, and (2) we run each benchmark with a co-runner, which is another multi-threaded benchmark. Each category contains multiple sets of experiments, in which each set is designed to target a specific resource in the memory hierarchy and measures the impact of contention on performance for that resource. The resources in the memory hierarchy that are considered in the experiments are: L1-cache, shared L2-cache, shared L3-cache, FSB, on-chip memory controller (MC), and memory socket connection.



(a) Intel-Yorkfield



(b) Intel-Harpertown

Figure 4.3: Experimental Platforms (*CX* stands for the processor core, *L1 HW-PF* and *L2 HW-PF* stand for hardware prefetcher for L1- and L2-caches, respectively and *FSB* and *MB* stand for Front Side Bus and Memory Bus, respectively.)

The multi-threaded applications that we use in our experiments are from the latest PARSEC 2.1 [38] and NAS parallel benchmark suites (NPB) [97]. We use the PARSEC

benchmark suite as it is composed of multi-threaded applications designed to be representative of next-generation shared-memory programs for multicore architectures. We use the NAS benchmark suite as it consists of representative applications for high performance computing. Both benchmark suites have multi-threaded applications of diverse characteristics. The PARSEC benchmarks used in the experiments are: *blackscholes* (BS), *bodytrack* (BT), *canneal* (CN), *dedup* (DD), *facesim* (FA), *ferret* (FE), *fluidanimate* (FL), *freqmine* (FQ), *raytrace* (RT), *streamcluster* (SC), *swaptions* (SW), *vips* (VP), and *x264* (X2). We use the largest *native* input set for the PARSEC benchmarks. We use nine benchmarks from NPB-OMP-3.3. We use the input *B* for the benchmark *DC* and input *D* for all other benchmarks as these are the largest inputs for the experimental platforms. We do not use *BT* from NPB as it does not execute multiple threads.

We keep profiling overhead as low as possible and employ a simple technique to instrument the benchmarks by identifying *pthread-create* system calls. By detecting new thread creation, we gather each thread's *threadID* information, which is necessary to get the per-thread profile information.

To collect different run-time statistics, as each benchmark in each experiment is run, profile information is collected by reading hardware performance counters using the Perfmon2 tool [98]. The interfaces defined in the tool's library allow user-level programs to read hardware performance counters on thread-basis (per-thread) and system-basis (per-core). These interfaces enable users to access the counters' values with a low run-time overhead. The initial set up for the counters takes 318μsec and reading one counter's value takes 3.5μsec, on average. After the initialization of the performance counters for each thread, the values of the counters are read via signal handlers when periodic signals are sent (every second) to each thread using the *threadID* information. As performance is used as the characterization metric in this instance, we collect the counter, UNHALTED_CORE_CYCLES's sampling values for each thread in the characterization configurations in all experiments to determine the effect of both intra- and inter-application contention.

We use statically linked binaries for our experiments, compiled with the default GCC version on the experimental platforms. The OpenMP benchmarks are compiled using *-fopenmp* flag. We do not include any additional compiler optimization flag other than the ones used in the benchmarks' original makefiles. The threads are affinitized by pinning them to cores via Linux *sched_setaffinity()* system call. When we co-schedule two applications and one application finishes before the rest, we immediately restart it. We perform this restart until the longest running application completes five iterations. We collect the application profile information for five iterations to ensure low variability in the collected values.

4.4.1.1 Experimental Platforms

To measure the contention among threads for different levels of caches, we require a machine that has multiple levels of cache. Because we need to map the application threads to the cores sharing one cache to determine the effect of cache contention, the platform must have both private (per-core) L1- and shared L2-caches. It must have single socket memory connection, so the contention for the FSB is expected to be the same and we are able to measure only contention for the caches. Intel-Yorkfield, shown in Figure 4.3(a), satisfies the requirements for such experiments and we use this platform to characterize applications for cache contention. This platform has four cores and each core has private L1-data and L1-instruction cache, each of size 32KB. It has two 6MB 24-way L2-caches and each L2-cache is shared by two cores. It has 2 GB of memory connected by single socket to the L2-cache, so there is one FSB. It runs Linux kernel 2.6.25.

To characterize a multi-threaded application for FSB and measure contention for this resource among threads, we need a platform that has multiple socket connections to memory, i.e., multiple FSBs. The cache hierarchy in each FSB connection must be the same so that we can isolate contention for the FSB by keeping the other factors (L1-/L2-cache contention) unchanged. Intel-Harpertown, shown in Figure 4.3(b), fulfills these requirements. Therefore, we choose this platform to determine and characterize applications for FSB contention. It has two processors each having four cores. Each core has private L1-data and L1-instruction

cache, each of size 32 KB. Each pair of cores share one of the four 6MB 24-way L2-caches. This platform has dual sockets, and each processor has a separate FSB connected to 32GB memory. It runs Linux kernel 2.6.30.

To characterize a multi-threaded application for on-chip memory controller contention, we need a platform that has multiple numbers of memory controllers. The number of other resources, such private and shared caches for each on-chip memory controller connection must be the same so that we can apply the methodology to isolate the effect of the contention for the memory controller. Intel-Xeon, shown in Figure 4.4(a), fulfills these requirements and is used to characterize PARSEC and NPB benchmarks based on the memory-controller contention. This platform has four processors. Each processor has eight cores and one 24-way 18MB L3-cache with integrated memory controller. Each core has private 32KB L1- and 256KB L2-cache. This platform is hyper-threaded, and each core has two thread contexts. Therefore, it has a total of 32 cores and 64 thread contexts. This platform has 250GB memory.

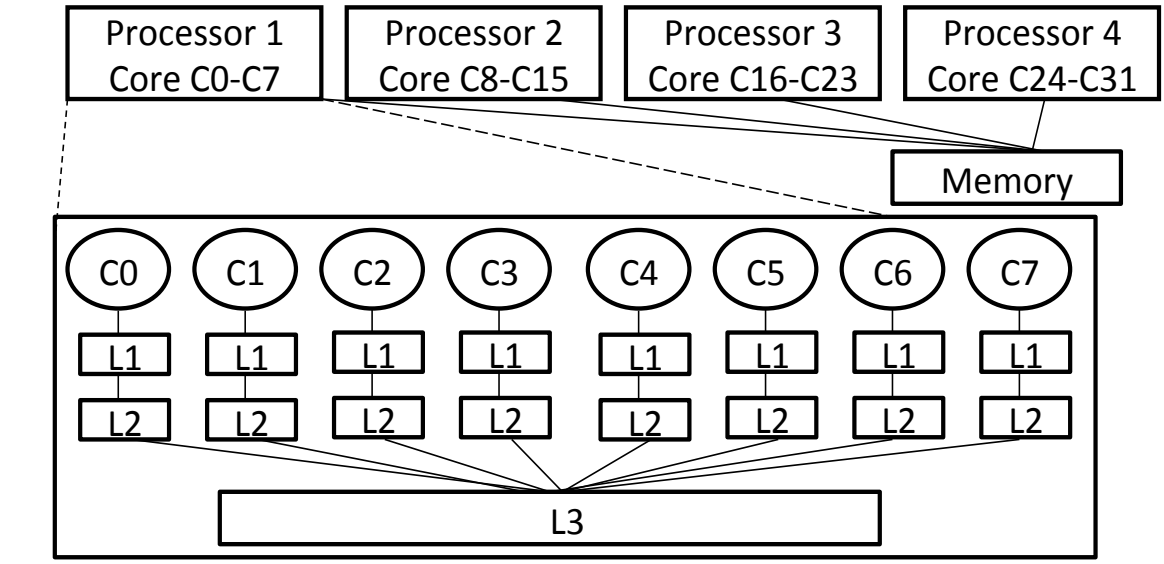
Similarly, to characterize applications based on contention for L3-cache and memory-socket, we select AMD-Opteron machine, shown in Figure 4.4(b) because this platform has multiple numbers of the targeted resources, and the L1- and L2-caches are private. It has four processors, each having twelve cores. Six cores share one 5MB L3-cache. Each core has private 64K L1- and 512KB L2-cache. Four processors are connected to the 95GB memory via four memory sockets.

Table 4.1 summarizes the configurations of the experimental platforms.

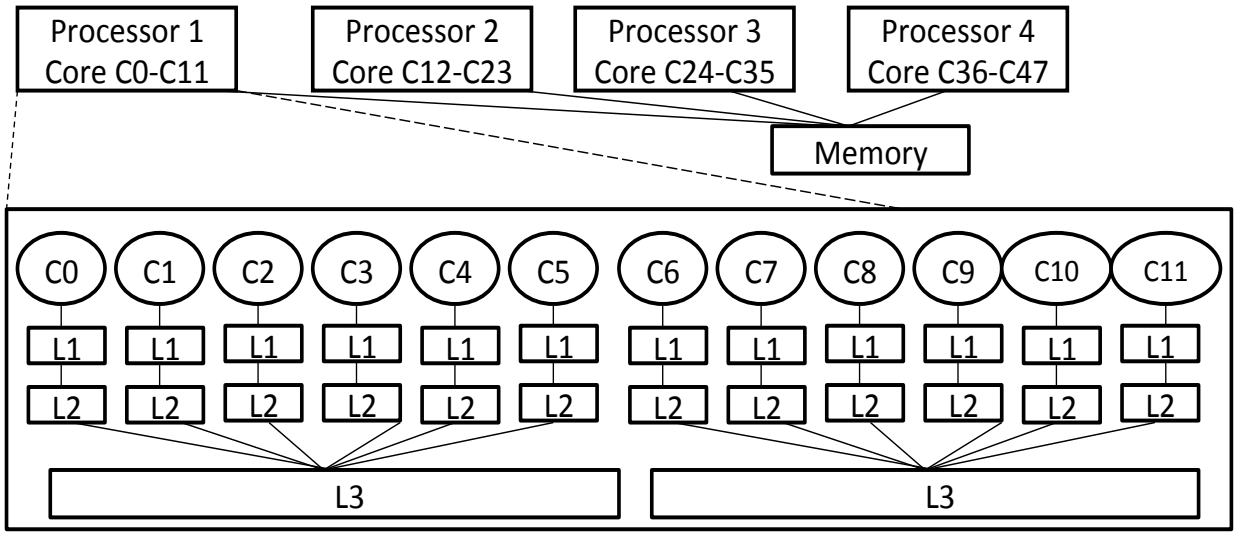
4.4.1.2 Characterization for Intra-application Contention

To characterize a multi-threaded application based on intra-application contention for a shared resource in the memory hierarchy, we run each benchmark solely according to the characterization methodology. The experiments are described below.

L1-cache: According to the methodology, to determine the effect of intra-application contention for L1-cache on application performance, we run each benchmark in two configura-



(a) Intel-Xeon



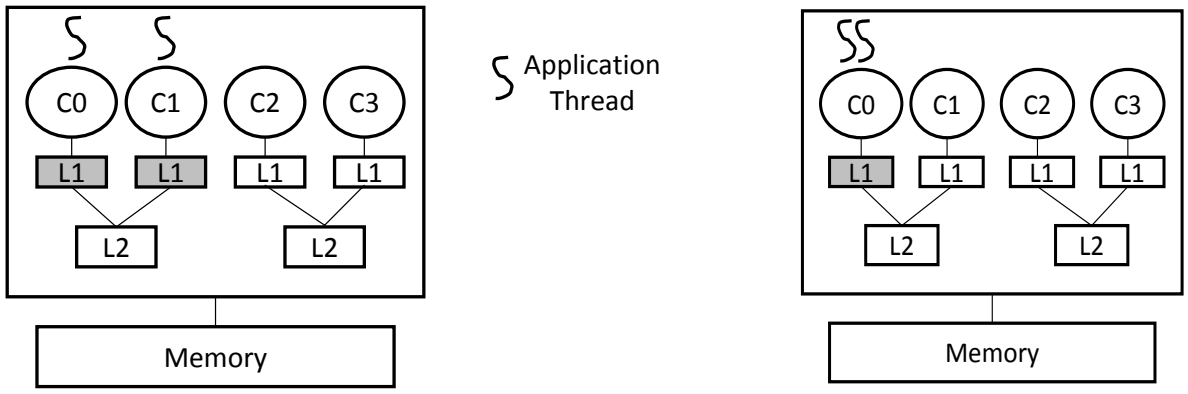
(b) AMD-Opteron

Figure 4.4: Topology of Experimental Platforms

tions. In each configuration, the number of threads to run equals the number of cores sharing one L2-cache. In the non-sharing configuration, two threads from a benchmark use their own private L1-caches, and there is no intra-application L1-cache contention. These threads are placed on the two cores that share one L2-cache, e.g., C0 and C1 (shown in Figure 4.5(a)). In the sharing configuration, two threads from the benchmark share one L1-cache compared to the exclusive access. In the presence of intra-application L1-cache contention, the threads

Platform	Topology	Linux kernel /GCC version	Number of cores/contexts	Memory system	Target Resource
Intel-Yorkfield	Figure 4.3(a)	2.6.25/4.2.4	4/4	private 32KB L1 2 shared 6MB L2 2GB memory	L2-cache
Intel-Harper-town	Figure 4.3(b)	2.6.30/ 4.2.4	8/8	private 32KB L1 4 shared 6MB L2 32GB memory	L2-cache, FSB
Intel-Xeon	Figure 4.4(a)	2.6.32/4.4.3	32/64	private 32KB L1 private 256KB L2 4 shared 18MB L3 250GB memory	L3-cache + Memory controller (MC)
AMD-Opteron	Figure 4.4(b)	2.6.32/4.4.3	48/48	private 64KB L1 private 512KB L2 8 shared 5MB L3 95GB memory	L3-cache, Memory socket

Table 4.1: Configuration of the experimental platforms

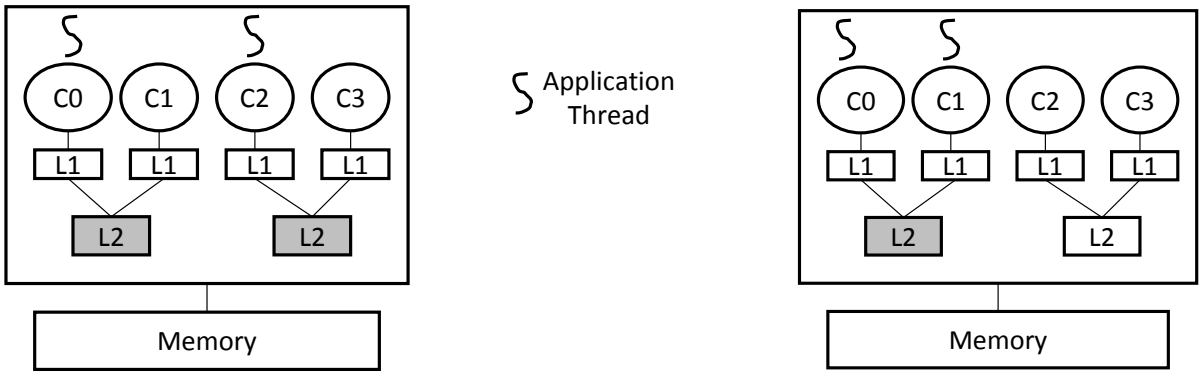


(a) Non-sharing Configuration

(b) Sharing Configuration

Figure 4.5: Configurations to characterize a multi-threaded application for intra-application L1-cache contention

compete for L1-cache space when they share the L1-cache and access conflicting cache-lines. Here, these threads are placed on one core, e.g., C0 (shown in Figure 4.5(b)) or C1. As we determine the effect of L1-cache contention, we keep the effect of L2-cache contention the same by placing threads to the cores that share the same L2-cache. Furthermore, we make sure that contention for FSB remains unchanged and choose Intel-Yorkfield that has one



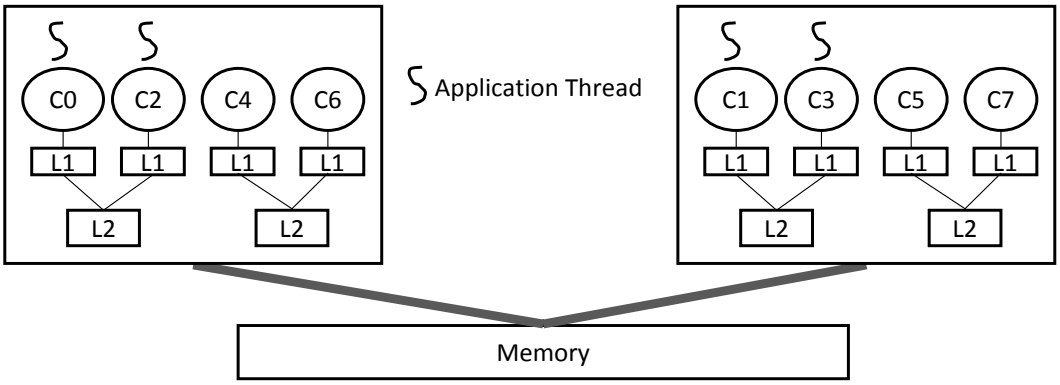
(a) Non-sharing Configuration (b) Sharing Configuration

Figure 4.6: Configurations to characterize a multi-threaded application for intra-application L2-cache contention

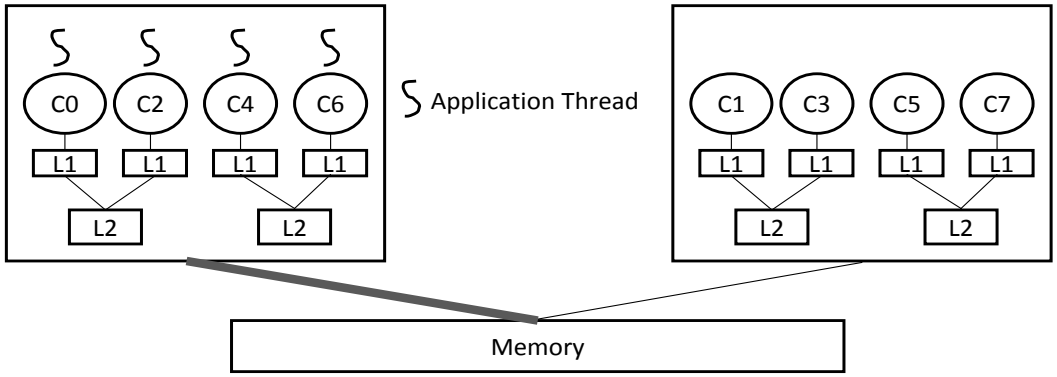
FSB, for this experiment. The only difference between these two configurations is the way L1-cache is shared between the threads, and we are able to measure how L1-cache contention affects the benchmark's performance.

L2-cache: Similar to L1-cache contention, to determine the effect of intra-application contention for L2-cache on application performance, we run each benchmark in two configurations. Each configuration runs threads equal to the number of L2-caches sharing one FSB. In the non-sharing configuration, two threads from a benchmark use their own L2-cache, avoiding intra-application contention for L2-cache. The threads are placed on the two cores that have separate L2-cache, e.g., C0, C2 (shown in Figure 4.6(a)) or C1, C3. In the sharing configuration, two threads from the benchmark share one L2-cache and contend for L2-cache with each other. Here, the threads are placed on the two cores sharing one L2-cache, e.g., C0, C1 (shown in Figure 4.6(b)) or C2, C3. As we measure contention for L2-caches, we avoid intra-application L1-cache contention by allowing only one thread to access one L1-cache and keep the FSB contention unchanged between configurations by choosing Intel-Yorkfield, which has one FSB.

Front Side Bus: To determine the effect of intra-application contention for the FSB on application performance, we need to understand how sharing the FSB among appli-



(a) Non-sharing Configuration



(b) Sharing Configuration

Figure 4.7: Configurations to characterize a multi-threaded application for intra-application FSB contention

cation/benchmark threads affects its performance compared to using separate FSBs. For this experiment, we use Intel-Harpertown as it has more than one FSB. According to the methodology, we run each benchmark in two configurations. In each configuration, the number of threads equals the number of cores sharing one FSB to fully utilize its bandwidth. In the non-sharing configuration, four threads from a benchmark use separate FSBs equally and do not compete for this resource. Four threads are placed on the four cores that have separate socket connections (separate bus) to memory (via shared L2-cache), e.g., C0, C2, C1 and C3 (shown in Figure 4.7(a)). In the sharing configuration, four threads use only one FSB and there is potential contention among them for this resource. In this case, four threads are placed on the four cores sharing one socket connection to memory, e.g., C0, C2, C4 and C6 (shown in Figure 4.7(b)). As both configurations use the same number of threads as cores

and L1-caches are private to each core, there is no intra-application contention for L1-cache. Similarly, as both configurations use two L2-caches shared by an equal number of threads, the contention for L2-cache remains the same. So comparing the performance output of the configurations, we are able to determine how bus bandwidth and FSB contention affect the performance of each benchmark.

Memory Controller: To determine the effect of intra-application contention for the on-chip memory controller on application performance, we need to understand how sharing the same memory controller among application threads affects its performance compared to using separate memory controllers. For this experiment, we use Intel-Xeon as it has multiple memory controllers. According to the methodology, we run each multi-threaded benchmark in two configurations. In each configuration, the number of threads equals the number of cores sharing one memory controller to fully utilize its bandwidth. In the non-sharing configuration, eight threads from a benchmark use two separate memory controllers equally and do not compete for this resource. Eight threads are placed on the eight cores that have separate memory controller connections to memory, e.g., C0 - C3 and C8 - C11 of the platform shown in Figure 4.4(a). In the sharing configuration, eight threads use only one memory controller and there is potential contention among them for this resource. In this case, eight threads are placed on the eight cores that share one memory controller connection to memory, e.g., C0 - C7 of the platform shown in Figure 4.4(a). As both configurations use the same number of threads as cores, and L1- and L2-caches are private to each core, there is no intra-application contention for L1- and L2-cache. So comparing the performance output of the configurations, we are able to determine how bus bandwidth and contention for memory controller affect the performance of each benchmark.

L3-cache: To determine the effect of intra-application contention for the L3-cache on application performance, we need to understand how sharing the same L3-cache among application threads affects its performance compared to using separate L3-caches. For this experiment, we use AMD-Opteron as it has multiple L3-caches on the same memory socket

connection. According to the methodology, we run each multi-threaded benchmark in two configurations. In each configuration, the number of threads equals the number of cores sharing one L3-cache. In the non-sharing configuration, six threads from a benchmark use two separate L3-caches equally and do not compete for this resource. Six threads are placed on the six cores that use two L3-caches, e.g., C0 - C2 and C6 - C8 of the platform shown in Figure 4.4(b). In the sharing configuration, six threads use only one L3-cache and there is potential contention among the sibling threads for this resource. In this case, six threads are placed on the six cores that share one L3-cache, e.g., C0 - C5 of the platform shown in Figure 4.4(b). As both configurations use the same number of threads as cores and L1- and L2-caches are private to each core, there is no intra-application contention for L1- and L2-cache. Both characterization configurations use the same memory socket. So comparing the performance output of the configurations, we are able to determine how contention for L3-cache affects the performance of each benchmark.

Memory Socket: To determine the effect of intra-application contention for the memory-socket connection on application performance, we need to understand how sharing the same memory-socket among application threads affects its performance compared to using separate memory sockets. For this experiment, we use AMD-Opteron as it has multiple memory-socket connections. According to the methodology, we run each multi-threaded benchmark in two configurations. In each configuration, the number of threads equals the number of cores sharing one memory socket. In the non-sharing configuration, twelve threads from a benchmark use two separate memory socket connections to memory equally and do not compete for this resource. Twelve threads are placed on the twelve cores that use two separate memory socket connection, e.g., C0 - C5 and C12 - C17 of the platform shown in Figure 4.4(b). In the sharing configuration, twelve threads use only one memory socket creating potential contention among them. In this case, twelve threads are placed on the cores that share one memory socket, e.g., C0 - C11 of the platform shown in Figure 4.4(b). As both configurations use the same number of threads and the placement of threads on the cores with respect to

Platform: Targeted resource	Number of threads	Non-sharing Configuration	Sharing Configuration
Intel-Yorkfield: L2-cache	2	Maps on cores C0 and C2	Maps on cores C0 and C1
Intel-Harpertown: L2-cache	2	Maps on cores C0, C4	Maps on cores C0, C2
Intel-Harpertown: FSB	4	Maps on cores C0, C2, C1, C3	Maps on cores C0, C2, C4, C6
Intel-Xeon: L3-cache+MC	8	Maps on cores C0 - C3 and C8 - C11	Maps on cores C0 - C7
AMD-Opteron: L3-cache	6	Maps on cores C0 - C2 and C6 - C8	Maps on cores C0 - C5
AMD-Opteron: Memory-socket	12	Maps on cores C0 - C5 and C12 - C17	Maps on cores C0 - C11

Table 4.2: Characterization configurations on the experimental platforms

L1-, L2- and L3-caches are the same, there is no intra-application contention for L1-, L2-, and L3-caches. So comparing the performance output of the configurations, we are able to determine how contention for memory socket affects the performance of each benchmark.

The characterization configurations for each targeted shared resources on the experimental platforms are summarized in Table 4.2. Each benchmark's $\text{SensitivityScore}_{\text{performance}}$ is calculated using Equation 4.1.

4.4.1.3 Characterization Results and Analyses for Intra-application Contention

Figure 4.8-4.16 show the characterization results of the PARSEC and NPB benchmarks, represented as $\text{SensitivityScore}_{\text{performance}}$, based on intra-application L1-, L2-cache, FSB, on-chip memory controller, L3-cache, and memory socket contention. The positive and negative $\text{SensitivityScore}_{\text{performance}}$ indicate an application's performance improvement and degradation, respectively. We do not characterize the NPB benchmarks on Intel-Yorkfield and Intel-Harpertown as these platforms are too resource constrained for these long-running applications.

L1-cache: The results of intra-application contention of the PARSEC benchmarks for L1-caches on Intel-Yorkfield, represented as $\text{SensitivityScore}_{\text{performance}}$, are shown in Figure 4.8.

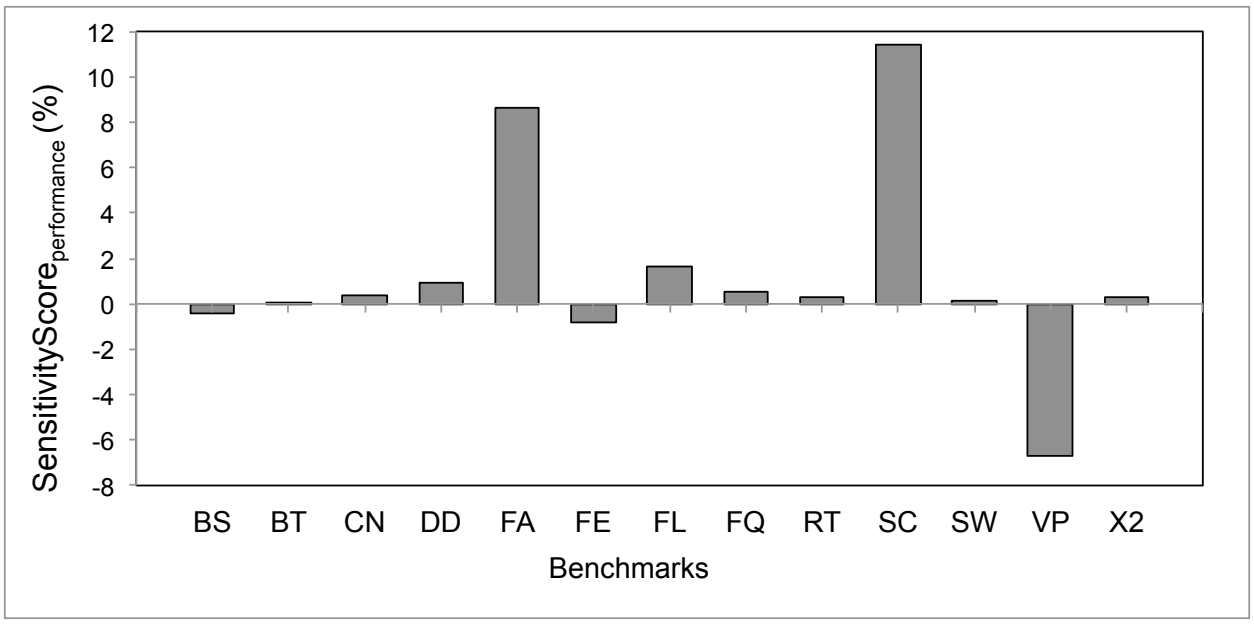


Figure 4.8: Characterization results of the PARSEC benchmarks for intra-application L1-cache contention, represented as SensitivityScore$_{\text{performance}}$

We observe in the figure that all the benchmarks except *blackscholes (BS), ferret (FE)* and *vips (VP)* show performance improvement (positive SensitivityScore$_{\text{performance}}$). When two threads from the same benchmark are placed on the core that uses one L1-cache, data sharing among the threads causes fewer cache misses as they share more than they contend for L1-cache. The fewer number of cache misses reduces the number of cycles to complete execution, improving the application performance. Additionally, as the benchmark's threads share data, when they are placed on the cores that use the same L1-cache, there are up to 99% reduced cache-snooping operations, decreasing the cache coherency protocol's overhead. These benchmarks, showing performance improvements, do not suffer from intra-application contention for L1-cache. As *facesim (FA)* and *streamcluster (SC)* have a large amount of data sharing, they show performance improvements of 8.6% and 11.42%, respectively. *Dedup (DD)* and *fluidanimate (FL)* show performance improvement of 0.9% and 1.6%, respectively. *Canneal (CN), freqmine (FQ), raytrace (RT)* and *x264 (X2)* show very small performance improvements of 0.34%, 0.55%, 0.29%, and 0.28%, respectively. Although *bodytrack (BT)* and *swaptions (SW)* show performance improvement, the magnitude of the

improvement is very close to zero, less than 0.1%. So the data sharing in these benchmarks yields minimal improvements. On the other hand, *ferret* and *vips* have fewer number of sharers compared to other benchmarks [38], causing contention when the threads share L1-cache. This characteristic results in more cache-misses and performance degradation by approximately 1% and 7%, respectively. Because these benchmarks show performance degradation (negative $\text{SensitivityScore}_{\text{performance}}$) when the threads only share L1-cache, they suffer from intra-application contention for L1-caches. *Blackscholes* shows the lowest and almost negligible performance degradation of 0.4% and is not much affected by L1-cache contention.

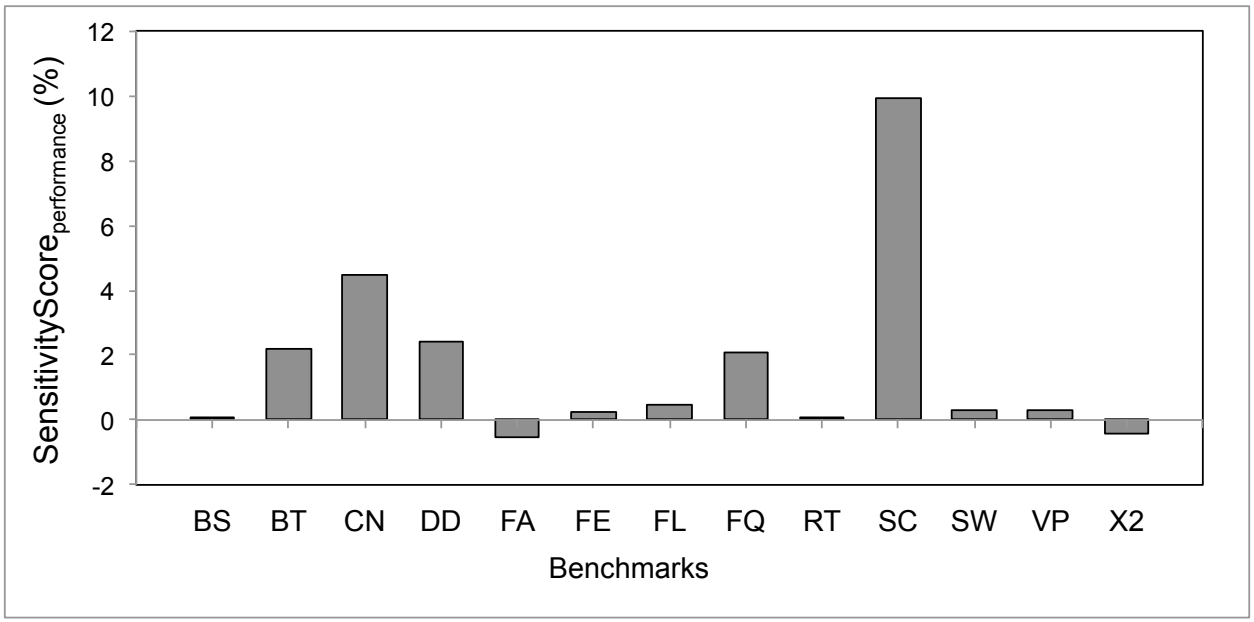


Figure 4.9: Characterization results of the PARSEC benchmarks for intra-application L2-cache contention, represented as $\text{SensitivityScore}_{\text{performance}}$

L2-cache: The results of intra-application contention of the PARSEC benchmarks for L2-caches on Intel-Yorkfield, represented as $\text{SensitivityScore}_{\text{performance}}$, are shown in Figure 4.9. In the figure we observe that *facesim* and *x264* show performance degradation close to 1%, on average and suffer from intra-application contention for L2-caches. The rest of the benchmarks show performance improvements because of sharing and reduced cache coherency traffic and do not suffer from intra-application L2-cache contention. Among these, *canneal, dedup* and

streamcluster's performance improvements are among the highest, respectively 4.5%, 2.4% and almost 10%, respectively. *Bodytrack* and *freqmine* also show performance improvements of approximately 2%. Although *vips* does not show performance improvement due to sharing in L1-caches, it shows slightly better sharing in L2-cache and small performance improvement of 0.28%. *Ferret, fluidanimate* and *swaptions* show small performance improvements of 0.27%, 0.49%, and 0.28%, respectively. *Blackscholes* and *raytrace* have negligible performance improvements, approximately 0.03% and 0.02% respectively, showing very small amount of sharing.

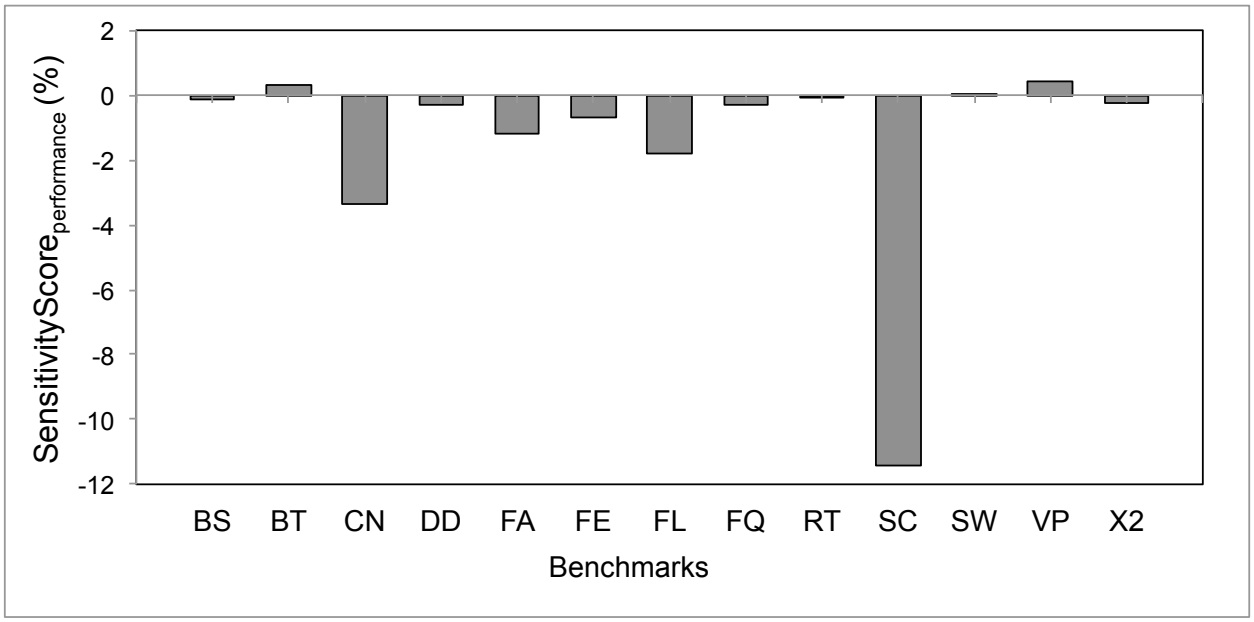


Figure 4.10: Characterization results of the PARSEC benchmarks for intra-application FSB contention, represented as SensitivityScore$_{\text{performance}}$

Front Side Bus: The results of intra-application contention of the PARSEC benchmarks for the FSB on Intel-Harpertown, represented as SensitivityScore$_{\text{performance}}$, are shown in Figure 4.10. We observe from the graph that the performances of most of the benchmarks, except *bodytrack* and *vips*, degrade when we map the threads to use one FSB. Because there is performance degradation for the reduced bus bandwidth, we conclude that there is intra-application contention for the FSB among the threads of these benchmarks. *Streamcluster* suffers the most performance degradation (nearly 12%), thereby showing the highest intra-

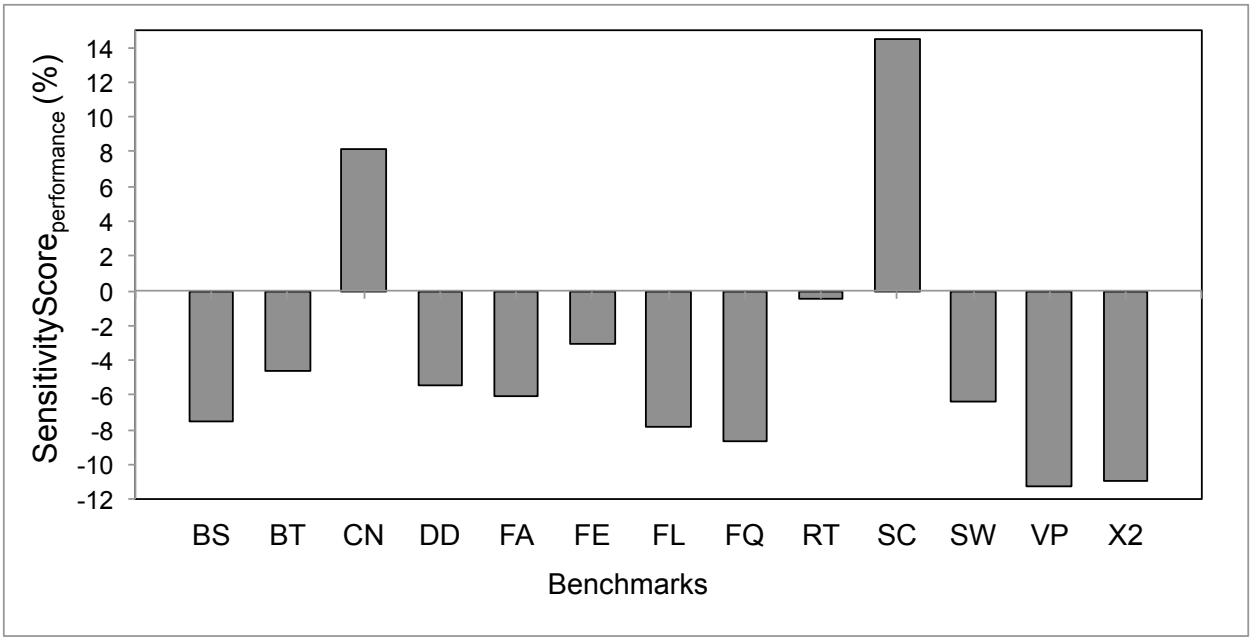


Figure 4.11: Characterization results of the PARSEC benchmarks for intra-application on-chip memory controller contention, represented as $\mathbf{SensitivityScore_{performance}}$

application contention for the FSB. *Canneal* suffers nearly 4% performance degradation. *Facesim* and *fluidanimate* show nearly 2% performance degradation. The performances of *blackscholes, dedup, ferret* and *freqmine* degrade on average by 1%. The performance degradation of *swaptions* and *raytrace* is negligible, less than 0.05%. In contrast, *vips* and *bodytrack* show very small performance improvements of 0.4% and 0.4% respectively and do not show intra-application contention for the FSB.

Memory Controller: The results of intra-application contention of the PARSEC benchmarks for the on-chip memory controller on Intel-Xeon, represented as $\text{SensitivityScore}_{\text{performance}}$, are shown in Figure 4.11. We observe from the graph that the performances of most benchmarks degrade when the threads are placed on the cores that use the same memory controller, except *canneal* and *streamcluster*. The on-chip memory controller has L3-cache integrated with it, and sharing the same memory controller means sharing the same L3-cache. The performances of *canneal* and *streamcluster* improve, respectively, by more than 8% and 14% in the sharing configuration because of the data-sharing among the threads in the L3-cache.

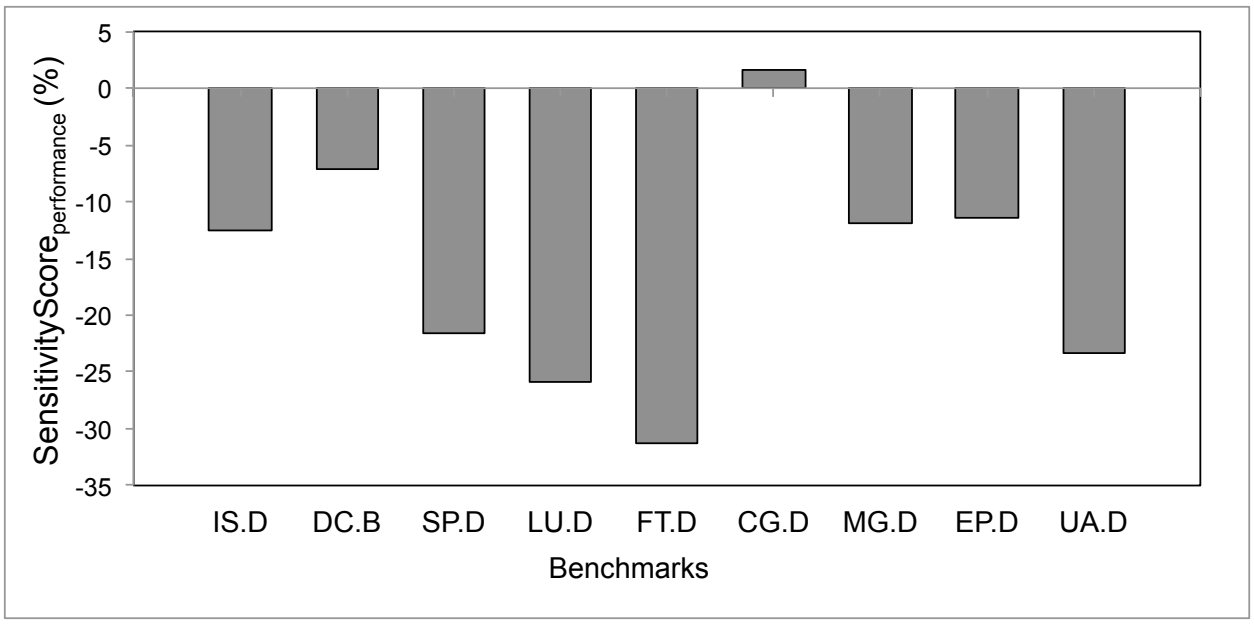


Figure 4.12: Characterization results of the NPB benchmarks for intra-application on-chip memory controller contention, represented as $\mathbf{SensitivityScore_{performance}}$

Except these two applications, the rest of the benchmarks suffer from intra-application contention for the on-chip memory controller and L3-cache. Especially *vips* and *x264* have performance degradation of more than 11% and 10%, respectively.

The $\text{SensitivityScore}_{\text{performance}}$ results of intra-application contention of the NPB benchmarks for the memory controller are shown in Figure 4.12. We observe from the graph that the performances of most benchmarks degrade when the threads are placed on the cores that use the same memory controller and L3-cache, except *CG.D*. *CG.D*'s performance improves because its bandwidth requirement is satisfied by one memory controller, and the threads do not have contention for the combined memory controller and L3-cache. The remaining benchmarks suffer from severe contention for the memory controller and L3-cache, where the performance degradation ranges from 7% to more than 31%. These application threads need more bandwidth during their executions.

L3-cache: The results of intra-application contention of the PARSEC benchmarks for L3-cache on AMD-Opteron platform, represented as $\text{SensitivityScore}_{\text{performance}}$, are shown

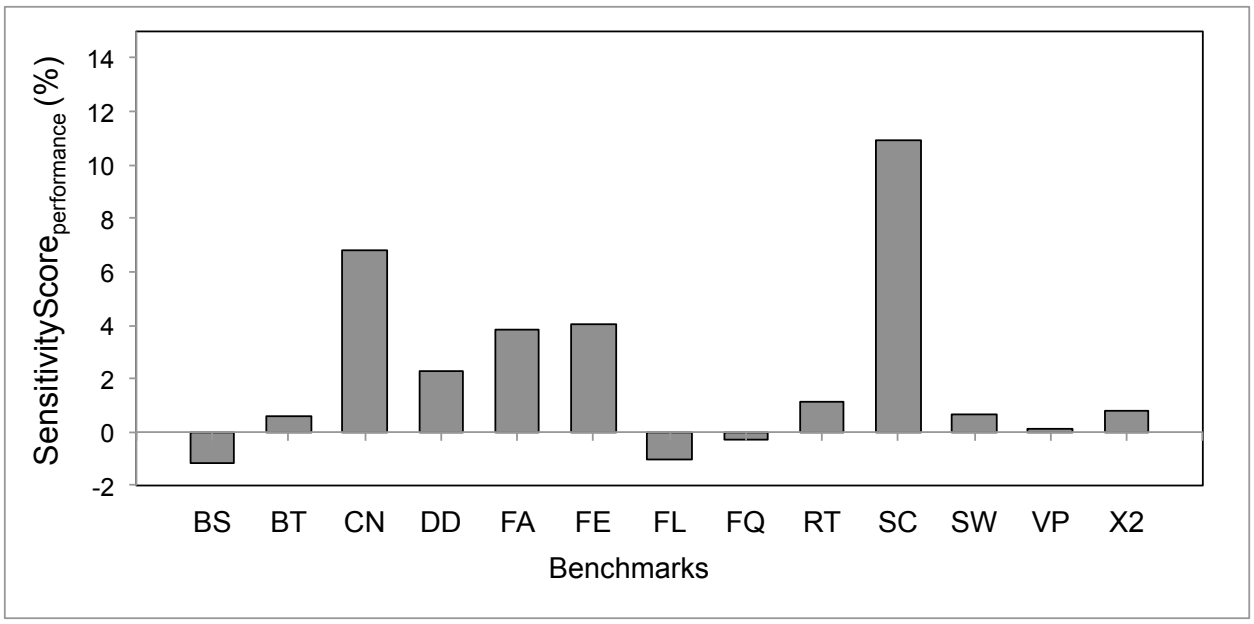


Figure 4.13: Characterization results of the PARSEC benchmarks for intra-application L3-cache contention, represented as SensitivityScore$_{\text{performance}}$

in Figure 4.13. From the graph, we observe that most benchmarks' performances improve in the sharing configuration, which range from roughly 1% to 10%. These performance improvements are because of the sharing behavior in the L3-cache by the sibling threads, especially for *canneal* and *streamcluster*. There are three benchmarks that suffer performance degradation because of L3-cache contention among the sibling threads. These applications are *blackscholes*, *fluidanimate*, and *freqmine*, and they have intra-application L3-cache contention.

The characterization results of intra-application contention of the NPB benchmarks for L3-cache on AMD-Opteron platform, represented as SensitivityScore$_{\text{performance}}$, are shown in Figure 4.14. We observe from the graph that the performances of all benchmarks, except *EP.D*, degrade when the threads are placed on the cores that use the same L3-cache. *EP.D* has a negligible performance improvement of less than 1%. The remaining benchmarks suffer from severe contention for the L3-cache, where the performance degradation ranges from 10% to more than 233%. These applications have very large memory footprint. When they are placed on the cores that use two L3-caches to the cores that use one L3-cache, the difference in total cache space causes the application to contend more and results in a

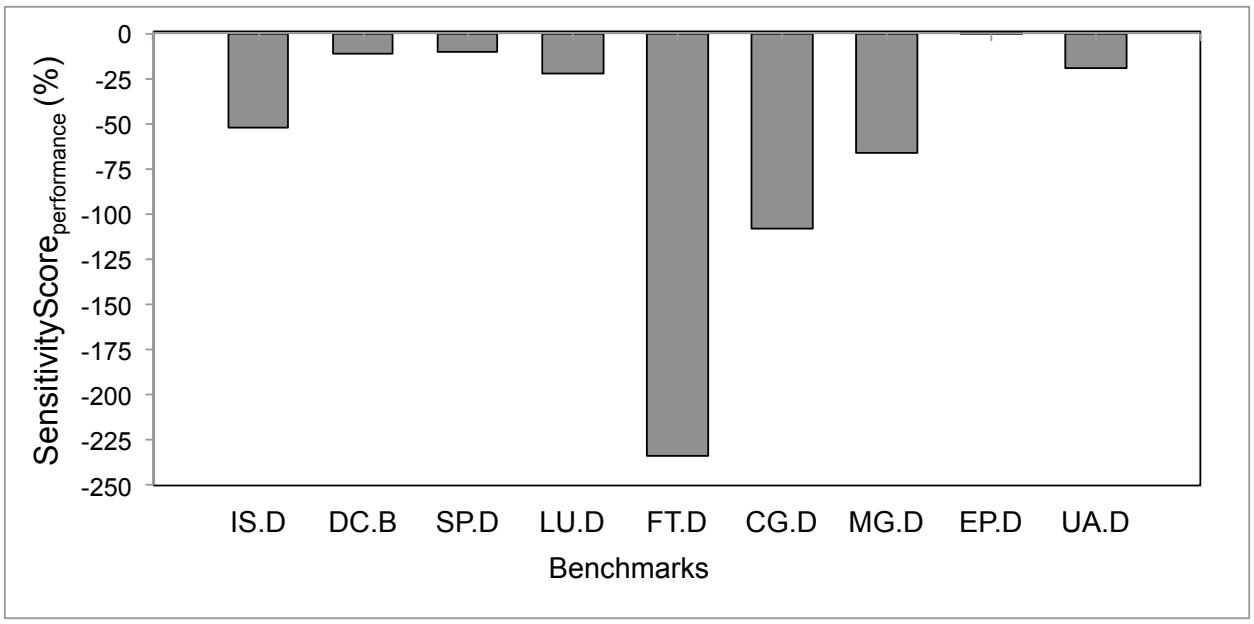


Figure 4.14: Characterization results of the NPB benchmarks for intra-application L3-cache contention, represented as SensitivityScore$_{\text{performance}}$

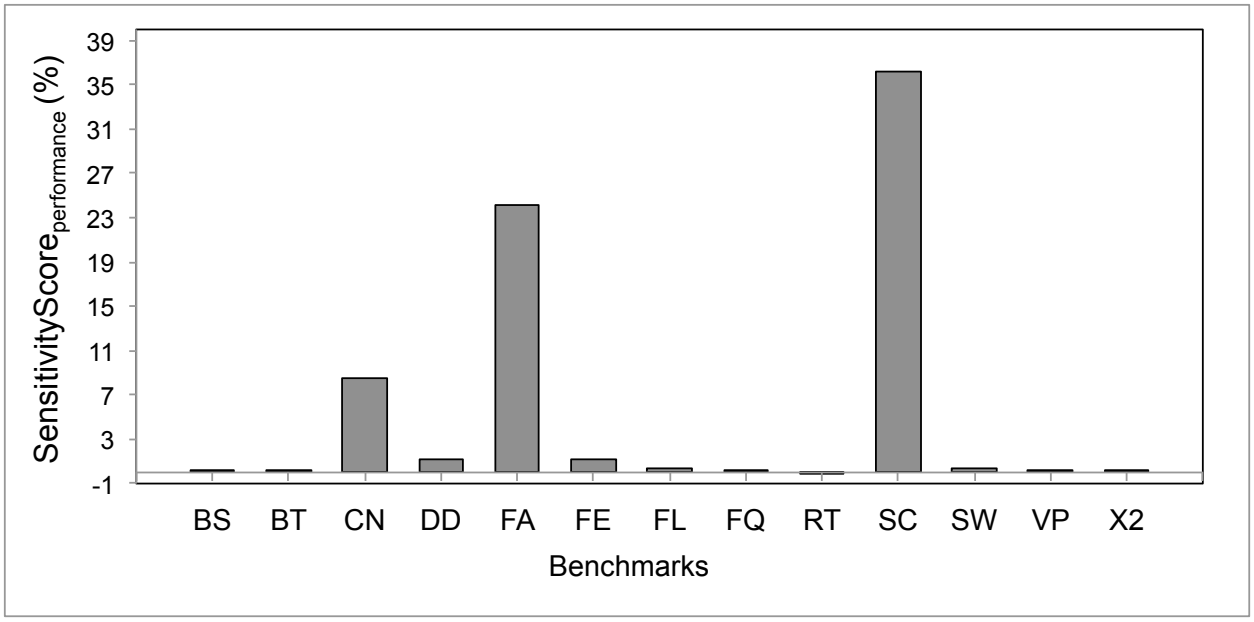


Figure 4.15: Characterization results of the PARSEC benchmarks for intra-application memory socket contention, represented as SensitivityScore$_{\text{performance}}$

significant performance degradation.

Memory Socket: The results of intra-application contention of the PARSEC benchmarks for the memory socket connection on AMD-Opteron platform, represented as SensitivityScore$_{\text{performance}}$,

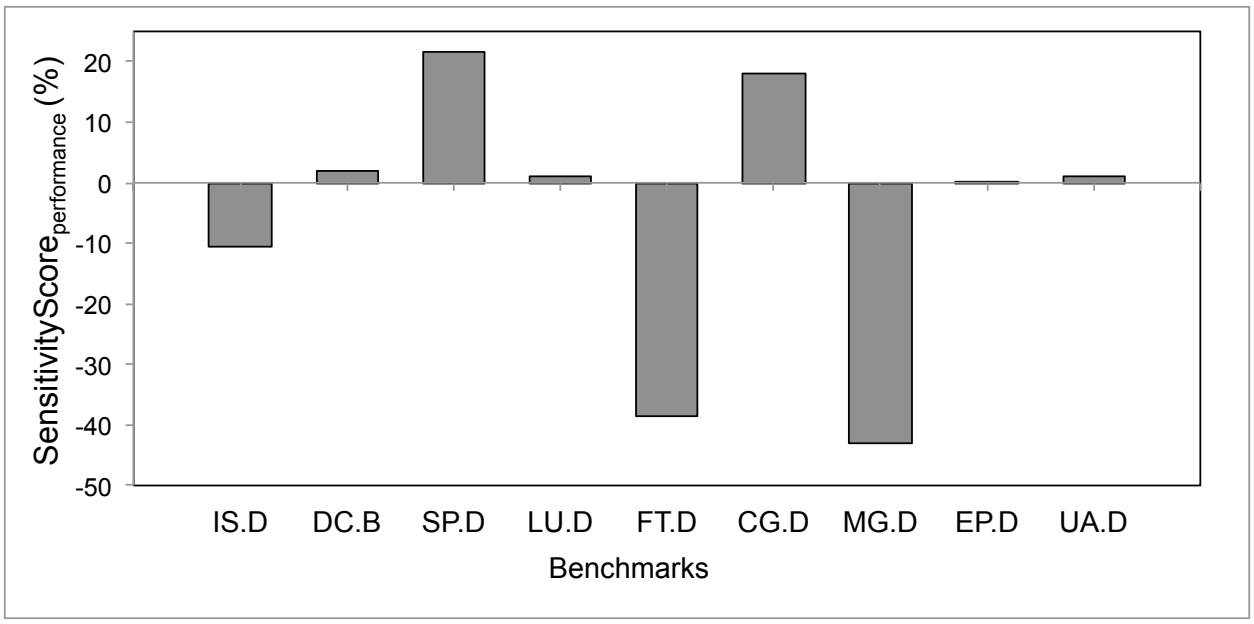


Figure 4.16: Characterization results of the NPB benchmarks for intra-application memory socket contention, represented as SensitivityScore$_{\text{performance}}$

are shown in Figure 4.15. From the figure we observe that most applications have improved performance when the threads share the same memory socket connection. Especially *canneal*, *facesim* and *streamcluster* have performance improvements of more than 8%, 24%, and 36% respectively. When these application threads are spread across to use separate memory sockets, the threads use distributed memory and the latency of remote memory accesses degrades the application performance. When they are placed on the cores that share the same memory socket, they do not use the distributed memory and use only local memory. The local memory access reduces the latency and improves application performance in the sharing configuration than that of the non-sharing configuration. Only one application, *raytrace*, has negligible performance degradation of 0.04% when the threads share the same memory socket, suffering from very low intra-application memory socket contention.

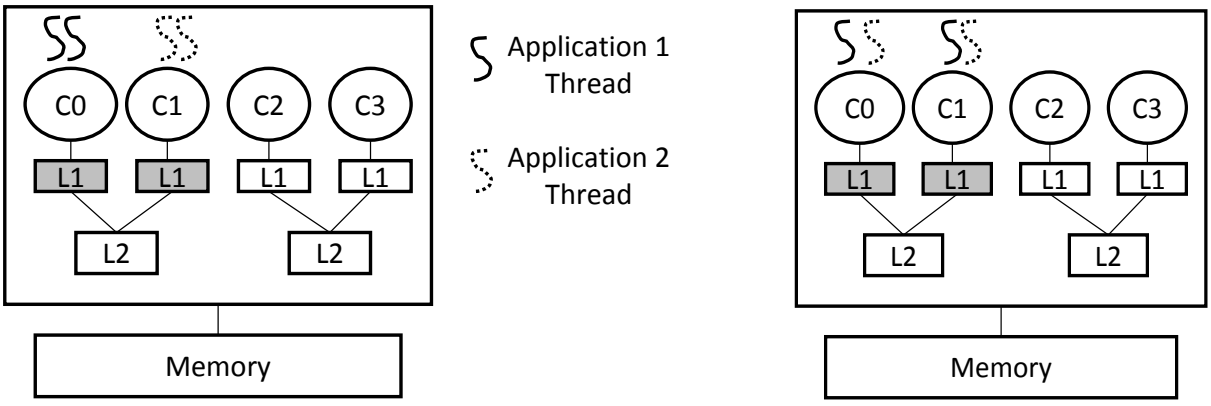
The results of intra-application contention of the NPB benchmarks for the memory socket connection on AMD-Opteron platform, represented as SensitivityScore$_{\text{performance}}$, are shown in Figure 4.16. We observe that most applications have performance improvements when all threads share the same memory socket connection because of reduced access to remote

memory and lower memory latency. Three benchmarks *IS.D*, *FT.D*, and *MG.D* have more memory bandwidth requirements. Therefore, if they are placed on the cores that use one memory socket connection, the sibling threads have intra-application contention for this resource, which leads to performance degradation.

The characterization results of PARSEC and NAS benchmarks based on intra-application contention for the shared targeted resources are used in the mapping phase of the framework. As inter-application contention results are not used in the mapping phase, we show the characterization based on inter-application contention for only the PARSEC benchmarks using Intel-Yorkfield and Intel-Harpertown platforms. The NPB benchmarks can be characterized in the same way as PARSEC, using the characterization configurations described below.

4.4.1.4 Characterization for Inter-application Contention

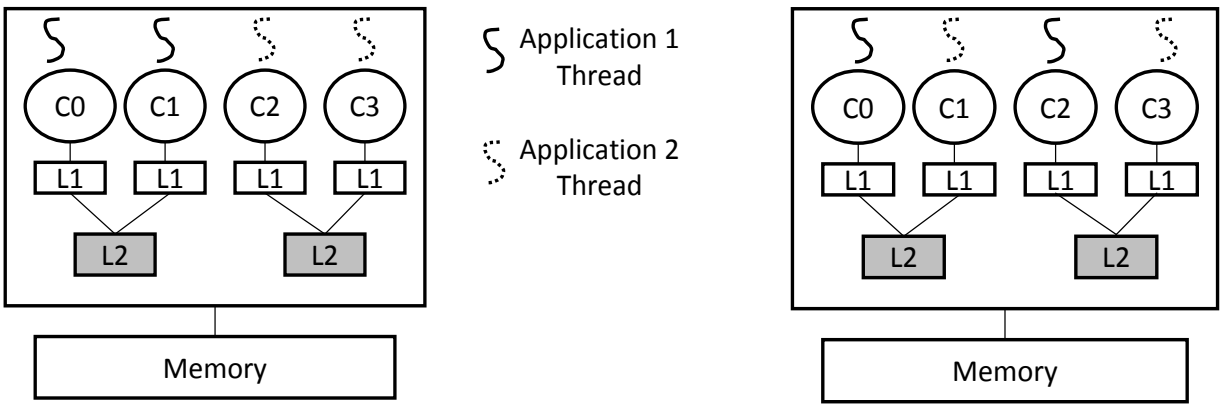
To understand the effect of inter-application contention for a particular resource in the memory hierarchy on application performance, each benchmark is run with another benchmark (co-runner) in two configurations. The resources in the memory hierarchy that are considered in the experiments are: L1-cache, shared L2-cache, FSB. The experiments for each shared resource are described below.



(a) Non-sharing Configuration (b) Sharing Configuration

Figure 4.17: Configurations to characterize a multi-threaded application for inter-application L1-cache contention

L1-cache: To measure the effect of inter-application contention for L1-cache on a multi-threaded application's performance, we run pairs of PARSEC benchmarks each with two threads in two configurations. In the non-sharing configuration, two threads of each benchmark get exclusive L1-cache access. There is no inter-application contention for L1-cache between them because L1-cache is not shared with the co-runner's threads. Two threads of one benchmark are placed on one core, e.g., C0 and two threads of the co-running benchmark are placed on the other core, e.g., C1 (shown in Figure 4.17(a)). In the sharing configuration, two threads from both benchmarks share the L1-caches and there is potential contention for L1-cache among them. Here, two threads from both benchmarks are placed on the two cores that share the same L2-cache, e.g., C0 and C1 (shown in Figure 4.17(b)). As we measure contention only for L1-caches, we keep the effect of L2-cache contention the same using one L2-cache and choose Intel-Yorkfield, having one FSB, to make sure that the contention for the FSB remains unchanged.



(a) Non-sharing Configuration (b) Sharing Configuration

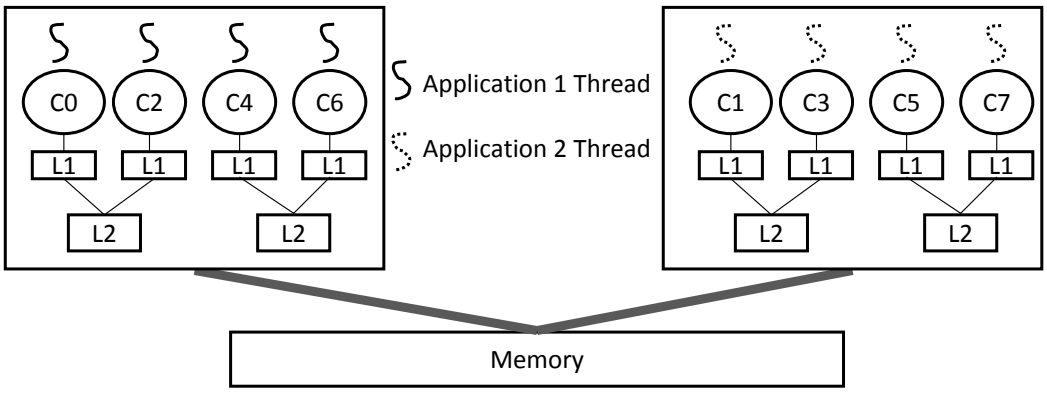
Figure 4.18: Configurations to characterize a multi-threaded application for inter-application L2-cache contention

L2-cache: Similar to L1-caches, to determine the effect of inter-application contention for L2-caches, we run pairs of PARSEC benchmarks each with two threads in two configurations. In the non-sharing configuration, two threads of each benchmark get exclusive L2-cache access. There is no inter-application L2-cache contention among them as L2-cache is not shared with

the co-runner's threads. On Intel-Yorkfield, two threads of one benchmark are placed on two cores, e.g., C0, C1 (shown in Figure 4.18(a)) or C2, C3 and two threads of the co-running benchmark are placed on the remaining cores, e.g., C2, C3 (shown in Figure 4.18(a)) or C0, C1. In the sharing configuration, one thread each from both benchmarks shares the L2-caches and there is potential contention for L2-cache between them. As shown in Figure 4.18(b), one thread from both benchmarks are placed on the two cores that share one L2-cache, e.g., C0, C1 and the second threads from both benchmarks are placed on the remaining two cores, which share the second L2-cache, e.g., C2, C3. Both configurations use the same number of L1-caches and single socket memory connection. So we are able to measure how L2-cache contention affects each benchmark's performance because the only difference between these configurations is how the L2-cache is shared between co-runners.

Front Side Bus: We run two PARSEC benchmarks each with four threads on Intel-Harpertown in two configurations to determine the effect of inter-application contention for FSB. In the non-sharing configuration, each benchmark gets its exclusive FSB access and there is no FSB interference/contention from the co-running benchmark. Four threads from one benchmark are placed on the four cores that share one socket connection to memory, e.g., C0, C2, C4, C6 and four threads from the other benchmark are placed on the remaining four cores that share the second socket connection to memory, e.g., C1, C3, C5, C7 (shown in Figure 4.19(a)). In the sharing configuration, both benchmarks share both FSB and there is potential contention for this resource between them. Here, four threads from one benchmark are placed equally on the four cores that have separate socket connections (separate bus) to memory, e.g., C0, C2, C1 and C3 and the remaining threads on the remaining cores (shown in Figure 4.19(b)). The only difference between these two configurations is how applications share the FSB connection to the memory. As both configurations use the same sized L1- and L2-cache, the contention for L1- and L2-cache remains unchanged, and we are able to determine how separate FSB usage affects the performance of each benchmark.

For the performance analysis for inter-application contention for a particular resource, we



(a) Non-sharing Configuration

(b) Sharing Configuration

Figure 4.19: Configurations to characterize a multi-threaded application for inter-application FSB contention

use Equation 4.2 to calculate the percentage performance difference between the application's performances in the two characterization configurations with each of its co-runners using the following formula:

$$Percent\_Performance\_Difference_i =$$

$$\frac{(SumOfCycles_{non-sharing_i} - SumOfCycles_{sharing_i}) * 100}{SumOfCycles_{non-sharing_i}} \tag{4.2}$$

Here, $SumOfCycles_{non-sharing_i}$ and $SumOfCycles_{sharing_i}$ are the sum of the hardware performance counter, UNHALTED _CORE _CYCLES's, sampling values in the non-sharing and sharing configuration, respectively, with the i-th co-runner, where $i = 1, 2, ..., n$ and $n =$ number of co-runners.

4.4.1.5 Characterization Results and Analyses for Inter-application Contention

The characterization results of the PARSEC benchmarks for inter-application L1-, L2-cache and FSB contention are shown in Figure 4.20- 4.22. In each figure, the X-axis corresponds to each benchmark in alphabetical order. Each column is a percentage stacked graph, where the stacks or segments show the performance results of a benchmark with each of its co-runners in alphabetical order from the bottom to the top. The lighter shade segments represent performance improvement, and darker shade segments represent performance degradation. For example, in Figure 4.20 for *BS*, the first segment from the bottom shows performance improvement with *BT*, the next segment shows performance degradation with *CN* while measuring inter-application L1-cache contention. Similarly, for *BT*, the first and second segment from the bottom shows performance degradation respectively with *BS* and *CN*.

If a particular segment in a benchmark's column is in the lighter shade, it means that the benchmark's performance improves in the sharing configuration. A performance improvement results when the benchmark's threads show lower contention for the resource with its co-runner's threads compared to the contention among its own threads for that resource. For example, in Figure 4.21, FE's (Ferret) performance improves when running with RT (Raytrace) as its co-runner, which means FE's threads do not have much sharing among themselves and have more L2-cache contention among themselves than the contention with the co-running RT's threads. On the other hand, if a particular segment in a benchmark's column is in the darker shade, it means that the benchmark's performance degrades in the sharing configuration and the benchmark's threads suffer from higher contention with its co-runner's threads than the contention among its own threads. For example, in Figure 4.21, FE's performance degrades when running with SC (Streamcluster) as its co-runner, which means FE's threads have more L2-cache contention with the co-running SC's threads than the contention among its own threads.

The number on top of each column (*Absolute Performance Difference Summation (APDS)*) is the sum of the absolute percentage performance differences of each benchmark with each of

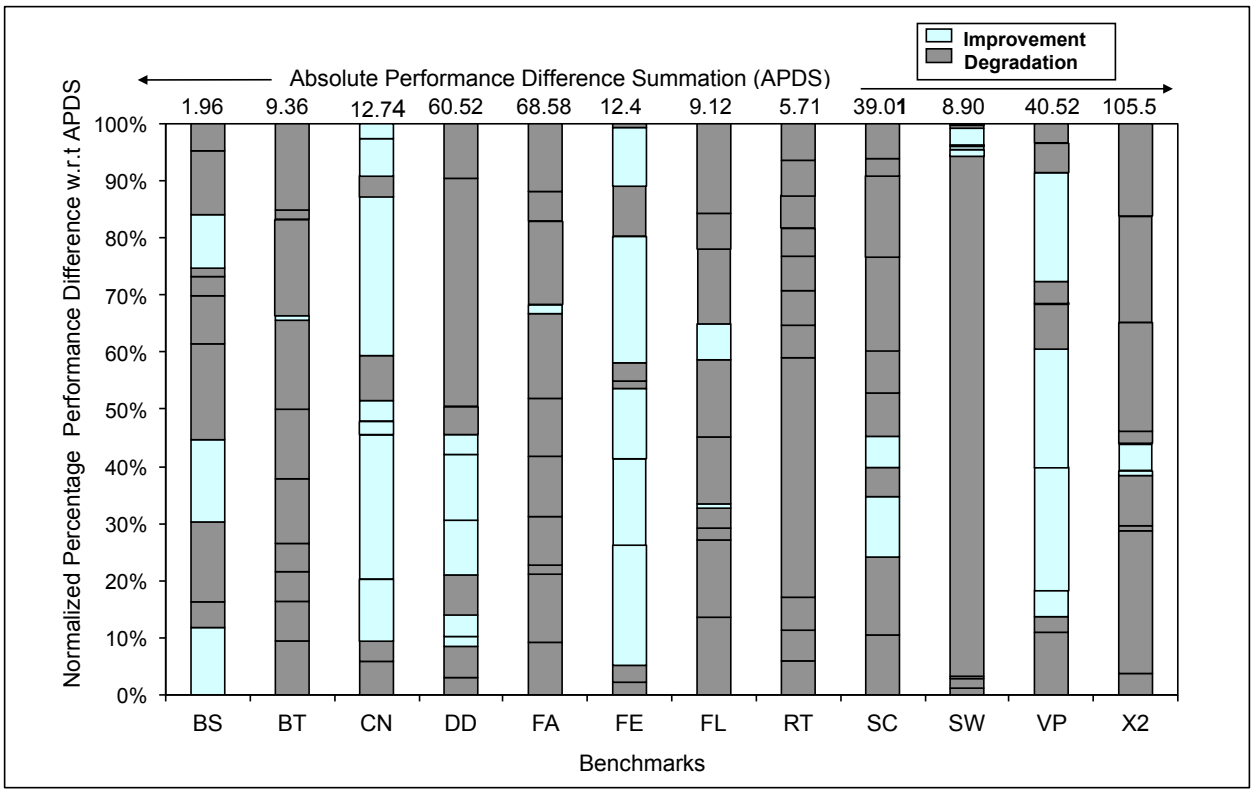


Figure 4.20: Characterization results of the PARSEC benchmarks for inter-application contention for L1-cache

its co-runners. The height of each segment in the columns represents percentage performance difference of a benchmark with one of its co-runners, normalized with respect to this summation to keep the total height of the column at 100%. To get the actual percentage performance difference for a benchmark with any co-runner, we multiply the height of the appropriate segment in the benchmark's column with the APDS value above the column. For example, to get the actual percentage performance improvement for *BS* with *BT* for L1-cache contention, we multiply the height of first segment of first column in Figure 4.20 with 1.96, which is $0.1173 * 1.96 = 0.23\%$.

L1-cache: From the results of inter-application contention of the PARSEC benchmarks for L1-cache (shown in Figure 4.20), we can categorize the benchmarks into three classes. This classification depends on how much they suffer from inter-application contention for L1-cache resulting in performance degradation or number of the darker shaded segments

in each column. The first class includes the benchmarks whose column has most of its segments in the darker shade and shows highest inter-application contention for L1-cache. This class includes *bodytrack, facesim, fluidanimate, raytrace, streamcluster, swaptions* and *x264*. Among these benchmarks, *facesim, streamcluster, x264* and *raytrace, swaptions* show, respectively, the most and least contention as the APDS values are among the highest and lowest of all benchmarks in this class. The next class includes the benchmarks whose columns have almost half of its height in the lighter and the other half in the darker shade. This class includes *blackscholes* and *dedup*. From the magnitude of APDS, we infer that *dedup* has more performance impact for L1-cache contention compared to *blackscholes*. The third category includes the benchmarks whose columns have most segments in the lighter shade. This class includes *canneal, ferret* and *vips*, which suffer more from intra-application than inter-application contention for L1-cache as their performance improve with most of the co-runners. *Vips* suffers the most due to intra-application contention as it has the highest APDS value among these benchmarks (also validated by the Figure 4.8 results).

L2-cache: Figure 4.21 shows the experimental results of the inter-application contention of the PARSEC benchmarks for L2-cache on Intel-Yorkfield. Similar to L1-cache contention results, we can categorize the benchmarks in three classes. In this case, we categorize them based on the APDS values as we observe in the figure that most of the benchmarks have all the column-segments in the darker shade, denoting performance degradation due to inter-application L2-cache contention. The first class includes the benchmarks that have the highest APDS values representing greater impact on the performance. This class includes *canneal, dedup, streamcluster,* and *vips*. All the segments of these benchmarks' columns are in the darker shade showing performance degradation due to high inter-application contention for L2-cache. The next category includes the benchmarks that have lower APDS than that of the previous class. This class includes *bodytrack, facesim, ferret, fluidanimate,* and *x264*. These benchmarks have most column segments in the darker shade showing performance degradation for L2-cache contention except *x264* and *ferret*. *X264* shows more

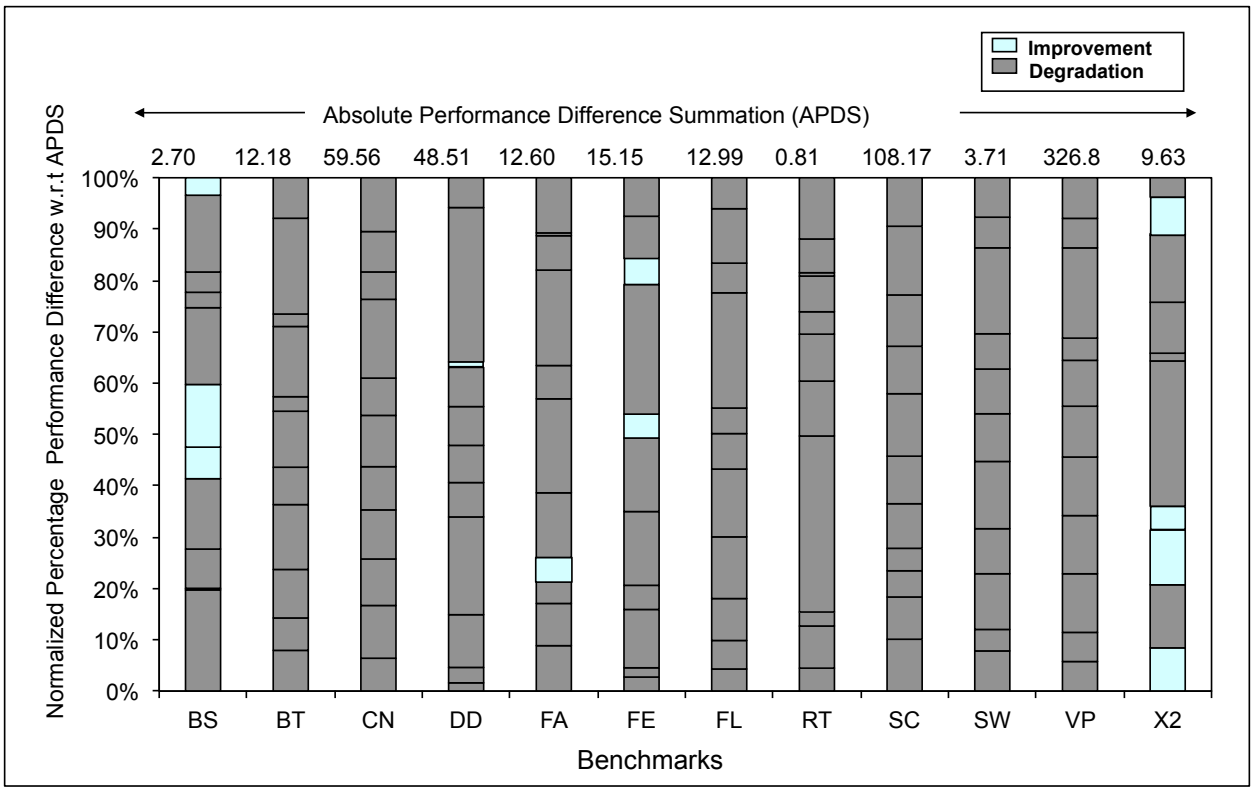


Figure 4.21: Characterization results of the PARSEC benchmarks for inter-application contention for L2-cache

intra-application contention for L2-cache with *blackscholes, canneal, dedup,* and *swaptions* as co-runner. *Ferret* shows more intra-application L2-cache contention with *raytrace* and *swaptions* as co-runner. The last class includes the rest of the benchmarks that show very small APDS values. This class includes *blackscholes, raytrace,* and *swaptions*. For each co-runner, these three benchmarks show on average 0.24%, 0.07% and 0.31% performance differences respectively, which is very small compared to those of the other benchmarks. So we can conclude that these three benchmarks' performances are not much affected by the inter-application L2-cache contention.

Front Side Bus: From the results of inter-application contention of the PARSEC benchmarks for the FSB on Intel-Harpertown (shown in Figure 4.22), we can categorize the benchmarks into three classes. Similar to L1-cache, the classification depends on how much they suffer from inter-application contention for the FSB resulting in performance

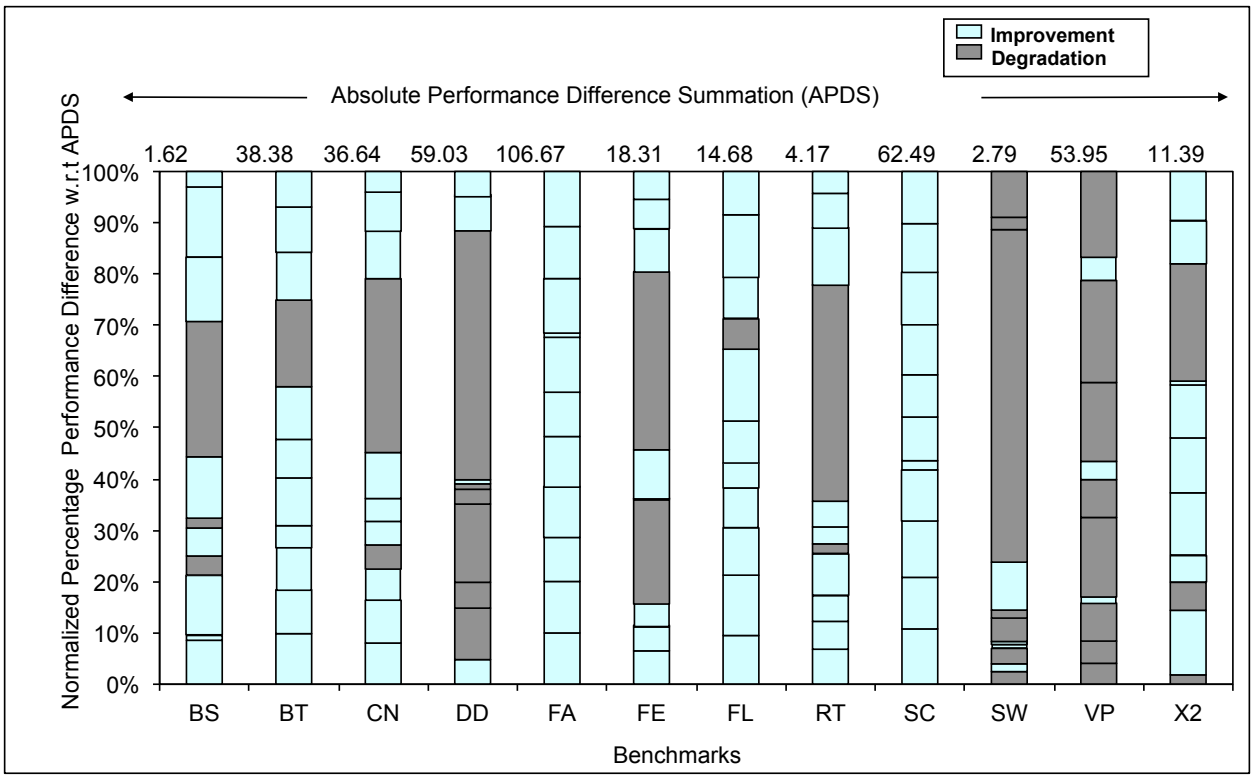


Figure 4.22: Characterization results of the PARSEC benchmarks for inter-application contention for FSB

degradation (i.e., the total length of darker segments in each column). The first class includes the benchmarks *dedup, swaptions* and *vips*, which have the most column area in the darker shade. *Dedup* and *vips* show the highest APDS values in this class and suffer more from inter-application contention for the FSB. The second class includes benchmarks that have both the lighter and darker shaded segments of almost equal length. This class includes *blackscholes, ferret, raytrace* and *x264*. Among these benchmarks, *blackscholes* and *raytrace* have very small APDS values, so their performance is not much affected because of the FSB contention. The third class includes the benchmarks whose columns have most of the segments in the lighter shade. These benchmarks' performances improve because of the increased bandwidth and they have more intra-application than inter-application contention for the FSB. This class includes *bodytrack, facesim, fluidanimate, canneal* and *streamcluster*. Among these benchmarks, *facesim* and *streamcluster* have the highest APDS values, which indicate they have higher intra-application contention for the FSB and is also validated by

Benchmarks	L1-cache	L2-cache	FSB	Memory Controller	L3-cache	Memory Socket
blackscholes	yes	no	yes	yes	yes	no
bodytrack	no	no	no	yes	no	no
canneal	no	no	yes	no	no	no
dedup	no	no	yes	yes	no	no
facesim	no	yes	yes	yes	no	no
ferret	yes	no	yes	yes	no	no
fluidanimate	no	no	yes	yes	yes	no
freqmine	no	no	yes	yes	yes	no
raytrace	no	no	yes	yes	no	yes
streamcluster	no	no	yes	no	no	no
swaptions	no	no	no	yes	no	no
vips	yes	no	no	yes	no	no
x264	no	yes	yes	yes	no	no

Table 4.3: Summary of the intra-application contention results for the PARSEC benchmarks

the results in Figure 4.10. We include *canneal* in this category as for most of its co-runners, it improves performance when it uses increased bandwidth and it also has high APDS value. All benchmarks suffer from inter-application contention for the FSB when they run with *streamcluster* as co-runner in sharing configuration. From this we infer that *streamcluster* has a higher memory requirement for which its co-runners suffer. Only *facesim* does not degrade performance with *streamcluster* as it suffers more due to intra-application contention.

4.4.2 Characterization: Discussion and Summary

Table 4.3 summarizes the characterization results of PARSEC benchmarks based on intra-application contention for the targeted resources on the four experimental platforms. From the table we observe that most PARSEC applications do not suffer from intra-application contention for the shared caches. Only three out of thirteen benchmarks suffer from intra-application contention for L1-, L2- and L3-caches. In particular, *canneal* and *streamcluster* do not suffer from intra-application contention for any cache resource, including private and shared caches. These two applications have data sharing among the sibling threads that

Benchmarks	Memory Controller	L3-cache	Memory Socket
IS.D	yes	yes	yes
DC.B	yes	yes	no
SP.D	yes	yes	no
LU.D	yes	yes	no
FT.D	yes	yes	yes
CG.D	no	yes	no
MG.D	yes	yes	yes
EP.D	yes	no	no
UA.D	yes	yes	no

Table 4.4: Summary of the intra-application contention results for the NPB benchmarks

help the application performances to improve when the threads share the same cache. Most PARSEC applications suffer from FSB and the on-chip memory controller contention among the sibling threads. All benchmarks, except *raytrace*, perform better when the sibling threads use the same memory socket connection, which reduces remote memory access and its latency.

Table 4.4 summarizes the characterization results of the NPB benchmarks for the targeted resources on Intel-Xeon and AMD-Opteron. From the table we observe that *IS.D, FT.D,* and *MG.D* suffer from intra-application contention for all the shared resources on the two platforms including the memory controller, L3-cache, and memory socket. Most benchmarks suffer from intra-application contention for the integrated memory controller with L3-cache on Intel-Xeon and L3-cache on AMD-Opteron. Three out of nine benchmarks suffer from memory socket contention among the sibling threads on AMD-Opteron.

Table 4.5 summarizes the characterization results of PARSEC benchmarks based on inter-application contention for the targeted resources on Intel-Yorkfield and Intel-Harpertown. Analyzing the APDS values in all inter-application contention results, we infer that the performances of *blackscholes, raytrace* and *swaptions* are not affected when the threads share the targeted resource with co-runners, and they do not have inter-application contention for the resources in the memory hierarchy. When a PARSEC application's threads are

Benchmarks	L1-cache	L2-cache	FSB
blackscholes	no	no	no
bodytrack	yes	yes	no
canneal	no	yes	no
dedup	yes	yes	yes
facesim	yes	yes	no
ferret	no	yes	no
fluidanimate	yes	yes	no
raytrace	no	no	no
streamcluster	yes	yes	no
swaptions	no	no	no
vips	no	yes	yes
x264	yes	yes	no

Table 4.5: Summary of the inter-application contention results for the PARSEC benchmarks

mapped to use the same shared cache with co-runners, the application performance degrades compared to when the cache is being only used by the sibling threads. Thus, we can conclude that most PARSEC benchmarks suffer from inter-application L2-cache contention. *X264* suffers the most performance degradation due to inter-application L1-cache contention. *Vips* suffers the most due to inter-application contention for L2-cache. Only *dedup* and *ferret* show inter-application contention for FSB. *Dedup* is the only benchmark that suffers from inter-application contention for all the resources considered in the memory hierarchy on these two platforms.

Tables 4.6 and 4.7 show the $\text{SensitivityScores}_{\text{performance}}$ of the PARSEC and NPB benchmarks, respectively. Positive $\text{SensitivityScore}_{\text{performance}}$ means application performance improves, and negative $\text{SensitivityScore}_{\text{performance}}$ means application performance degrades in the sharing configuration. These $\text{SensitivityScores}_{\text{performance}}$ are used later in the mapping phase of the framework.

4.4.3 Mapping: Experimental Details and Results

To evaluate $\text{ReSense}_{\text{Performance}}$'s effectiveness in mapping the threads of applications from a workload using $\text{ReSensor}_{\text{P}}$, we choose applications from the multi-threaded benchmark suites,

Platform	Intel-Yorkfield	Intel-Harper-town	Intel-Harper-town	Intel-Xeon	AMD-Opteron	AMD-Opteron
Benchmarks	L2-cache	L2-cache	FSB	L3-cache +MC	L3-cache	Memory Socket
blackscholes (BS)	0.0354	−0.0922	−0.1277	−7.5441	−1.1348	0.1289
bodytrack (BT)	2.2210	3.412	0.2922	−4.6291	0.5649	0.0558
canneal (CN)	4.4049	4.6012	−3.3664	8.1034	6.7756	8.5197
dedup (DD)	2.4294	1.7702	−0.2925	−5.4493	2.2727	1.1811
facesim (FA)	−0.5406	−6.6455	−1.1869	−6.0272	3.8579	24.148
ferret (FE)	0.2665	−0.4081	−0.6787	−3.0600	4.0712	1.1431
fluidanimate (FL)	0.4920	1.9047	−1.7727	−7.8056	−1.0396	0.3836
freqmine (FQ)	2.065	−0.4574	−0.3188	−8.6898	−0.2647	0.0623
raytrace (RT)	0.0196	0.7365	−0.0036	−0.4280	1.1278	−0.0463
streamcluster (SC)	9.9298	9.1566	−11.3983	14.4905	10.9319	36.148
swaptions (SW)	0.2762	0.2570	0.0061	−6.3446	0.6461	0.2931
vips (VP)	0.2803	1.1600	0.4082	−11.2819	0.0897	0.1766
x264 (X2)	−0.4192	−0.7398	−0.2531	−10.9900	0.7736	0.0787

Table 4.6: SensitivityScore$_{\text{performance}}$ of the PARSEC benchmarks

Platform	Intel-Xeon	AMD-Opteron	AMD-Opteron
Benchmarks	L3-cache +MC	L3-cache	Memory Socket
IS.D	−12.4783	−52.0424	−10.4446
DC.B	−7.1677	−11.0202	1.9157
SP.D	−21.5190	−10.1729	21.6309
LU.D	−25.8932	−22.3693	1.1384
FT.D	−31.2067	−233.2549	−38.6669
CG.D	1.5424	−107.5569	17.9871
MG.D	−11.9599	−66.5676	−43.0097
EP.D	−11.4738	0.0476	0.2832
UA.D	−23.3012	−19.5132	0.9321

Table 4.7: SensitivityScore$_{\text{performance}}$ of the NPB benchmarks

PARSEC and NAS parallel benchmarks with the same input set described in Section 4.4.1. We choose the same four experimental platforms: Intel-Yorkfield, Intel-Harpertown, Intel-Xeon, and AMD-Opteron. Table 4.1 describes the configurations of the platforms in detail. The selected machines represent a range of different micro-architectures, topologies and types of shared resources in the memory hierarchy and provide evidence of the generality of

$\text{ReSense}_{\text{Performance}}$ and $\text{ReSensor}_{\text{P}}$.

The $\text{ReSense}_{\text{Performance}}$ run-time system is implemented as a user-level virtual execution manager using REEact [99]. We choose this framework because it is customizable, especially designed for CMPs, and has very low (less than 3%) run-time overhead. $\text{ReSense}_{\text{Performance}}$ uses several services provided by REEact to detect the creation and termination of an application thread, including detecting the start and finish of an application and pinning application threads on specific cores.

We compare the experimental results with the native OS, as after an extensive search for similar work, we find it is the only viable option. Most of the previous thread-mapping or scheduling work focus on single-threaded applications, and extensions to accommodate multi-threaded applications are not obvious. The prior research on multi-threaded applications, on the other hand, focuses on optimizing energy, choosing the thread count, minimizing lock contention, or optimal core allocation, goals and techniques which are different than mitigating shared-resource contention and improving application performance by determining the thread-mapping. This work is the first to have management of contention for shared-memory resources for multiple multi-threaded applications from dynamic workloads via thread-mapping as the goal. We believe comparing to the native OS is a fair comparison as recent operating systems, including the one we use, consider an application's cache and memory behavior in scheduling [100].

To evaluate $\text{ReSense}_{\text{Performance}}$'s effectiveness over the native OS, we run the experiments in two configurations. In the first or baseline configuration, we run the workloads under the OS's control where the native OS determines the thread-mapping (called OS-mapping). In the second configuration, we run the workloads under $\text{ReSense}_{\text{Performance}}$'s control, using the mapping determined by $\text{ReSensor}_{\text{P}}$ (called $\text{ReSensor}_{\text{P}}$-mapping). In all experiments, the number of workloads are chosen to assure statistical significance for *t-test* (See Section 4.4.4). The evaluation metrics, *average response time* and *throughput* are computed according to the following equations and the evaluation results are normalized with respect to the native OS.

Here, n is the total number of applications in a workload and *Execution Time* is the average wall-clock execution time of an application.

$$Average\ Response\ Time = \frac{\sum_{i=1}^{n} ExecutionTime_i}{n} \tag{4.3}$$

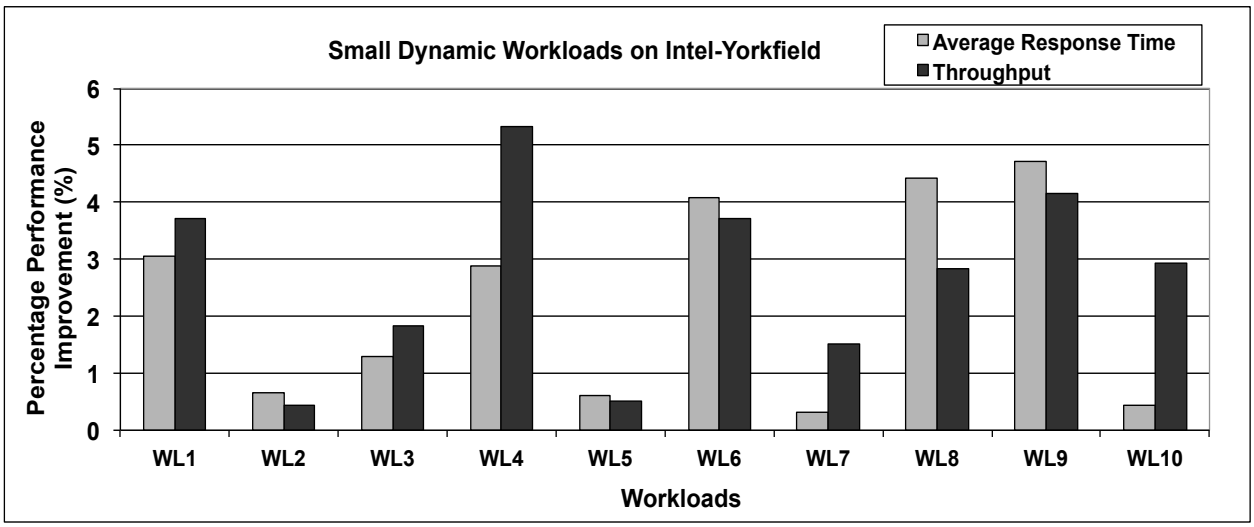
$$Throughput = \sum_{i=1}^{n} \frac{1}{ExecutionTime_i} \tag{4.4}$$

In all our experiments described in the following sections, we configure each benchmark to run with two, four, eight and six threads on Intel-Yorkfield, Intel-Harpertown, Intel-Xeon and AMD-Opteron, respectively. We represent each workload, WL_n that consists of n applications, as $\{ts_1(BM_1,k_1)\ ts_2(BM_2,k_2)\ ...\ ts_n(BM_n,k_n)\}$, which means at time-stamp ts_i, BM_i arrives and executes for k_i iterations. Depending on the size of a workload, the time-stamps are randomly chosen between 0 and 400 seconds, and the benchmarks are randomly selected from the PARSEC and NPB benchmark suites. The number of iterations is randomly chosen between 1 and 10. These parameters for the different-sized workloads are described in the corresponding experiments in detail.

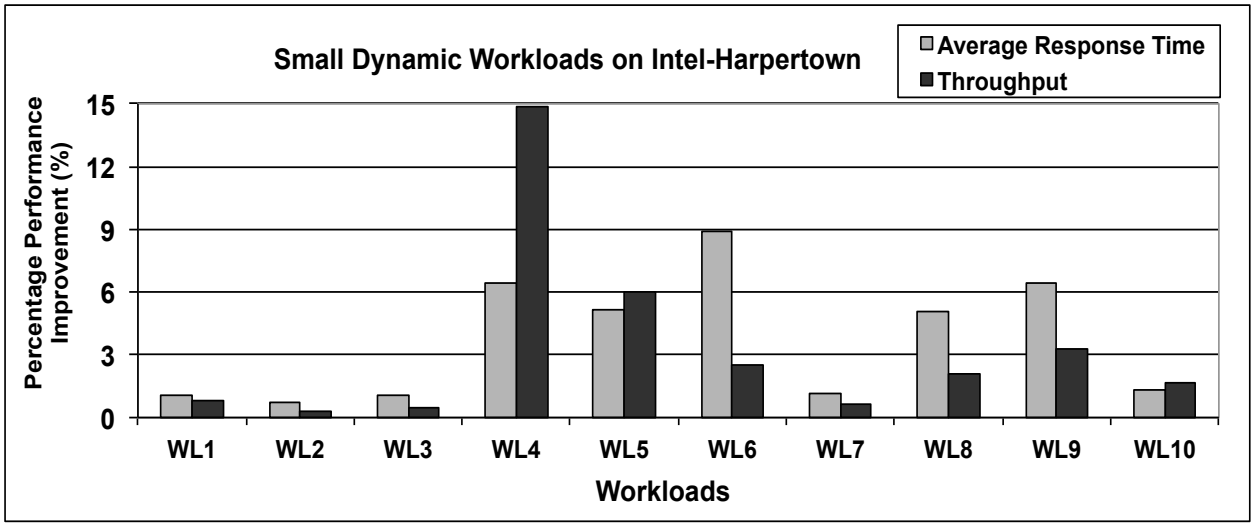
4.4.3.1 Evaluation Results: Small Dynamic Workloads

To evaluate if $\text{ReSense}_{\text{Performance}}$ determines the effective thread-to-core mappings of the multi-threaded applications using $\text{SensitivityScores}_{\text{performance}}$ and $\text{ReSensor}_{\text{P}}$, we first use *Small* dynamic workloads. We randomly select three PARSEC benchmarks. The first two benchmarks start execution simultaneously. After the second benchmark finishes, the third benchmark executes for a random i_3 iterations. We choose to execute the third benchmark to evaluate $\text{ReSense}_{\text{Performance}}$'s effectiveness at dynamically adjusting the mapping based on the new benchmark's $\text{SensitivityScore}_{\text{performance}}$. Each workload has simultaneously executing two or one benchmark at some point in time. The benchmarks and the parameters of the workloads are described in details in Table 4.8.

In Figure 4.23(a) we observe that the average response time and throughput of most



(a) Results on Intel-Yorkfield



(b) Results on Intel-Harpertown

Figure 4.23: Performance results of *Small* Dynamic Workloads, normalized to the native OS ($\text{ReSense}_{\text{Performance}}$ performs better than the OS)

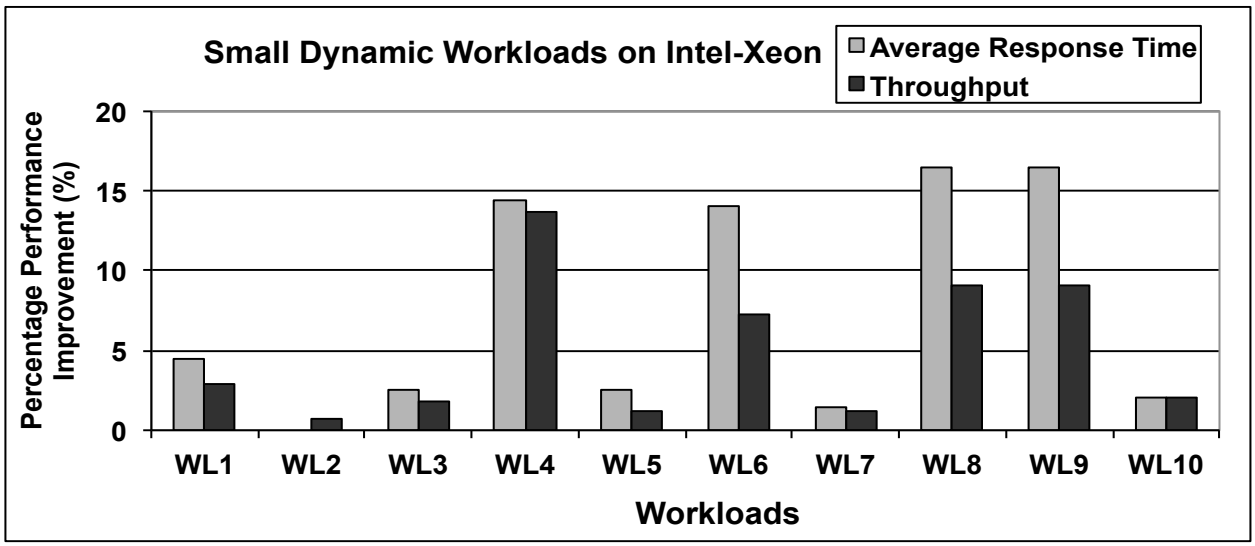
workloads improve by up to 4.75% and 5.32% on Intel-Yorkfield. The improvement indicates that $\text{ReSense}_{\text{Performance}}$ adjusts the benchmarks' thread-mappings dynamically in the presence of a co-runner using $\text{ReSensor}_{\text{P}}$ and the corresponding $\text{SensitivityScores}_{\text{performance}}$. The workloads, *WL2*, *WL3*, *WL5* and *WL7*, consist of at least two benchmarks that are not L2-cache sensitive, and thus the performance difference between the OS and $\text{ReSense}_{\text{Performance}}$ is small.

Workloads	Parameters of *Small* Workloads	Parameters of *Medium* Workloads
WL1	{(*CN*,10)(*FL*,4)(*SW*,8)}	{(*FA*,5)(*FL*,7) 117 (*SP*,9) 322 (*DD*,5)}
WL2	{(*FL*,9)(*BS*,6)(*BT*,2)}	{(*SC*,8)(*LU*,5) 289 (*IS*,7) 196 (*BS*,6)}
WL3	{(*SW*,3)(*CN*,10)(*FL*,8)}	{(*EP*,2)(*SW*,3) 228 (*UA*,9) 211 (*DD*,5)}
WL4	{(*RT*,4)(*SC*,1)(*CN*,3)}	{(*MG*,4)(*RT*,10) 242 (*DC*,7) 275 (*BT*,5)}
WL5	{(*BS*,7)(*FL*,7)(*FA*,2)}	{(*SC*,8)(*BS*,2) 113 (*IS*,9) 139 (*DC*,10)}
WL6	{(*FA*,1)(*BS*,3)(*SC*,3)}	{(*FE*,7)(*DC*,6) 227 (*CN*,3) 284 (*BT*,10)}
WL7	{(*CN*,4)(*FA*,1)(*BS*,1)}	{(*SC*,9)(*FE*,7) 257 (*FQ*,3) 147 (*IS*,6)}
WL8	{(*CN*,2)(*SW*,9)(*SC*,5)}	{(*UA*,4)(*SW*,10) 133 (*CG*,6) 146 (*VP*,7)}
WL9	{(*SC*,9)(*FQ*,3)(*SW*,3)}	{(*SP*,3)(*FQ*,6) 138 (*SC*,8) 155 (*CN*,10)}
WL10	{(*FQ*,5)(*CN*,4)(*RT*,2)}	{(*CN*,5)(*FA*, 9) 157 (*IS*,4) 167 (*FL*,9)}

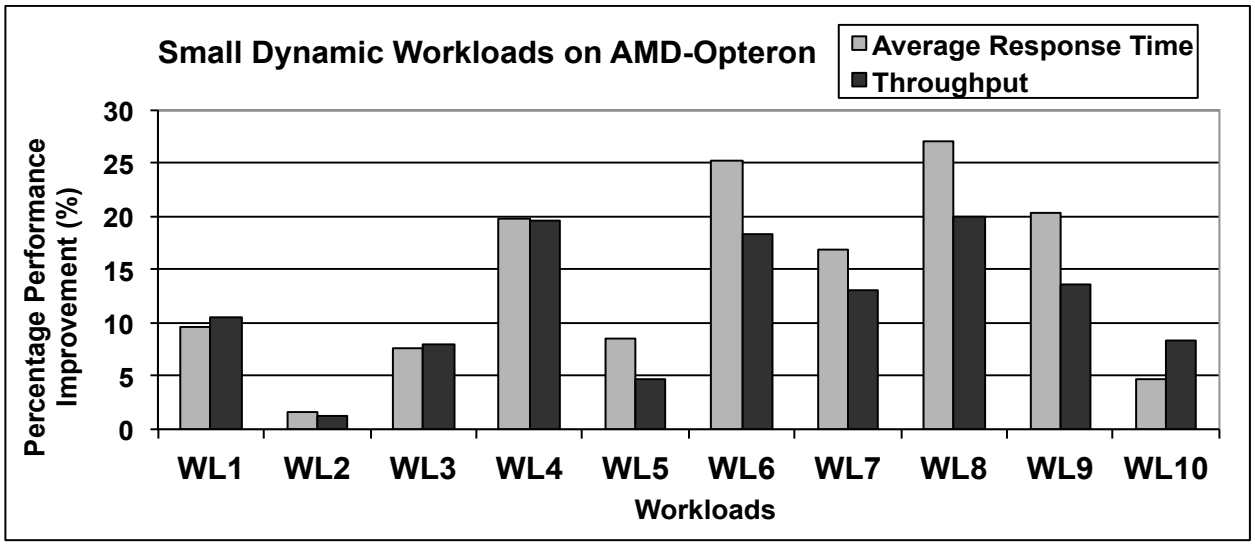
Table 4.8: *Small* and *Medium* Dynamic Workload Set

In Figure 4.23(b) we observe that the average response time and throughput improve by up to 8.89% and 14.88% for most workloads on Intel-Harpertown, especially those containing *SC*, *FA*, *FL* and *CN*. These benchmarks have the highest $\text{SensitivityScores}_{\text{performance}}$ for FSB and are more memory-intensive than the other benchmarks. *SC*, *FL* and *CN* also have data sharing in the L2-cache. Because $\text{ReSensor}_{\text{P}}$ considers the benchmarks' $\text{SensitivityScore}_{\text{performance}}$ for both L2-cache and FSB, $\text{ReSense}_{\text{Performance}}$ maps the threads to use both FSB's bandwidth and the same cache, resulting in response time and throughput improvements. As the benchmarks in other workloads do not have high $\text{SensitivityScores}_{\text{performance}}$ for FSB, their performance differences between OS and $\text{ReSense}_{\text{Performance}}$ are small.

The improvement of both average response time and throughput by up to 16.52% and 13.70% (Figure 4.24(a)) and by up to 27.03% and 19.97% (Figure 4.24(b)), indicates that $\text{ReSensor}_{\text{P}}$ effectively adjusts the thread-mappings depending on the benchmarks' $\text{SensitivityScores}_{\text{performance}}$ and the underlying platform's resource topology. Comparing Figure 4.23 and Figure 4.24, we observe that the performance improvements on the more powerful machines (Intel-Xeon and AMD-Opteron) are much higher than that of the less powerful machines (Intel-Yorkfield and Intel-Harpertown). This discrepancy between the performance gains of the more and less powerful machine is caused by the benchmarks, which



(a) Results on Intel-Xeon



(b) Results on AMD-Opteron

Figure 4.24: Performance results of *Small* Dynamic Workloads, normalized to the native OS (ReSense$_{\text{Performance}}$ performs better than the OS)

are more sensitive to the shared resources on the more powerful machine.

To summarize, by utilizing an application's SensitivityScore$_{\text{performance}}$, ReSense$_{\text{Performance}}$ effectively uses ReSensor$_{\text{P}}$ to map threads from dynamic pairs of multi-threaded applications and improves response time and throughput.

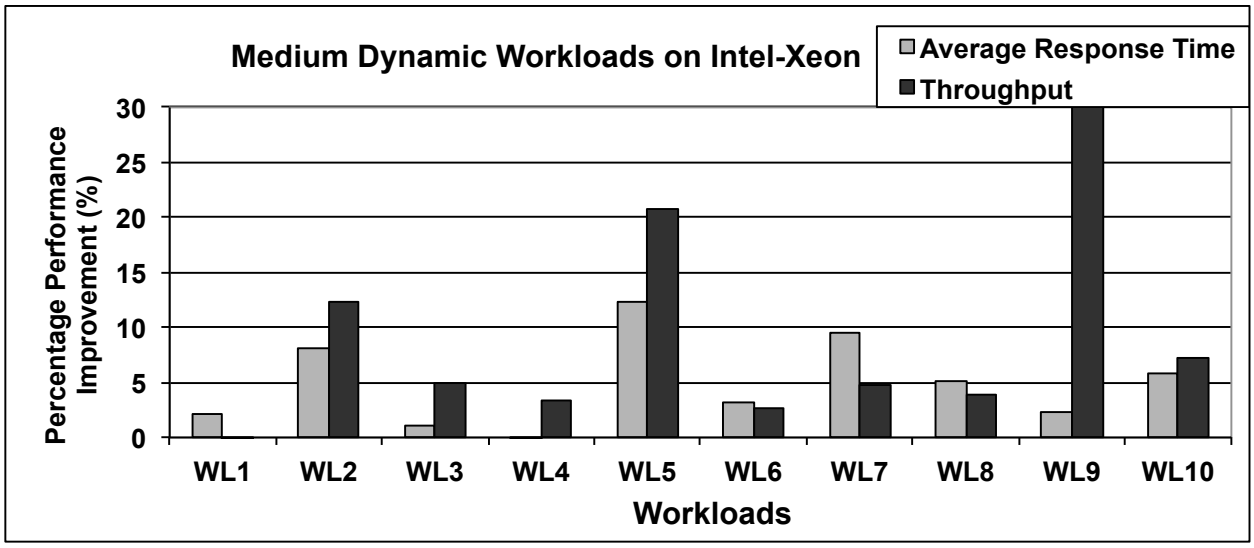
4.4.3.2 Evaluation Results: Medium Dynamic Workloads

To demonstrate that $\text{ReSense}_{\text{Performance}}$ effectively maps threads from workloads consisting of multi-threaded applications, we run experiments with *Medium* workloads. Each workload consists of four randomly selected benchmarks from both PARSEC and NPB to have a diverse set of applications. Two benchmarks start simultaneous execution at the beginning and the third and fourth benchmark arrive after random intervals. A benchmark in the workload continues to execute and re-execute for a number of times. Thus, even when the third and fourth benchmarks arrive and execute, the first and second benchmarks are still executing. If any benchmark finishes execution, it restarts immediately without any intermediate delay if its number of iterations is not over. Therefore, on average more than 50% of the time, there are four simultaneously executing multi-threaded applications in the system for *Medium* workloads. The benchmarks and the parameters of the workloads are described in Table 4.8.

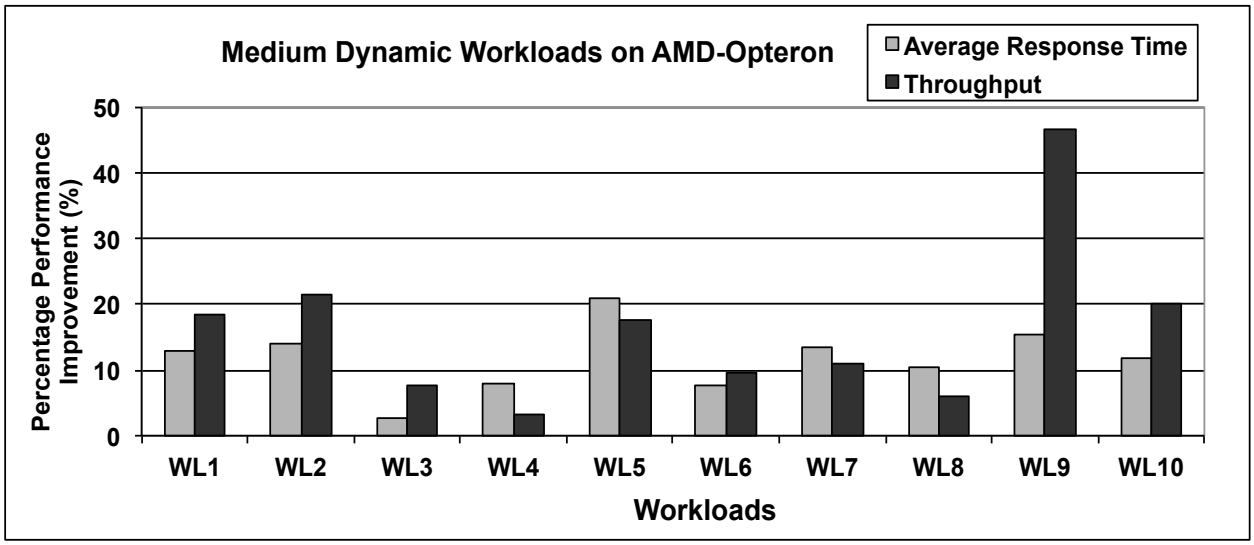
In Figure 4.25, we observe that $\text{ReSense}_{\text{Performance}}$ improves the average response time and throughput by up to 12.38% and 30% on Intel-Xeon and 20.89% and 46.56% on AMD-Opteron, over the native OS. From the improvements in both metrics for every workload, we conclude that $\text{ReSense}_{\text{Performance}}$ effectively maps multi-threaded applications from very diverse dynamic workloads and dynamically adjusts the thread-mappings using $\text{ReSensor}_{\text{P}}$ as the benchmarks arrive and execute non-deterministically.

On Intel-Xeon, all the benchmarks in $WL1$, $WL3$, $WL4$, and $WL8$ have negative $\text{SensitivityScores}_{\text{performance}}$ for (L3-cache+MC). Therefore, $\text{ReSense}_{\text{Performance}}$ maps the sibling threads on separate L3-caches to reduce the cache contention among threads. Under OS-mapping, the sibling threads are randomly mapped on the cores using separate L3-caches and the mapping determined by $\text{ReSensor}_{\text{P}}$ and OS is similar. Therefore, the performance difference between $\text{ReSense}_{\text{Performance}}$ and the OS is small.

To summarize, $\text{ReSense}_{\text{Performance}}$ improves the workload's average response time and throughput by dynamically adjusting the thread-mappings of the multi-threaded applications in the presence of multiple dynamic co-runners using $\text{ReSensor}_{\text{P}}$ and $\text{SensitivityScore}_{\text{performance}}$



(a) Results on Intel-Xeon



(b) Results on AMD-Opteron

Figure 4.25: Performance results for *Medium* Dynamic Workloads, normalized to the native OS ($\text{ReSense}_{\text{Performance}}$ performs better than the OS)

of the applications.

4.4.3.3 Evaluation Results: Large Dynamic Workloads

To evaluate $\text{ReSense}_{\text{Performance}}$'s scalability and the capability of handling more multi-threaded applications and threads in a more dynamic environment, we run experiments with *Large* dynamic workloads. The workloads are composed of randomly selected eight benchmarks

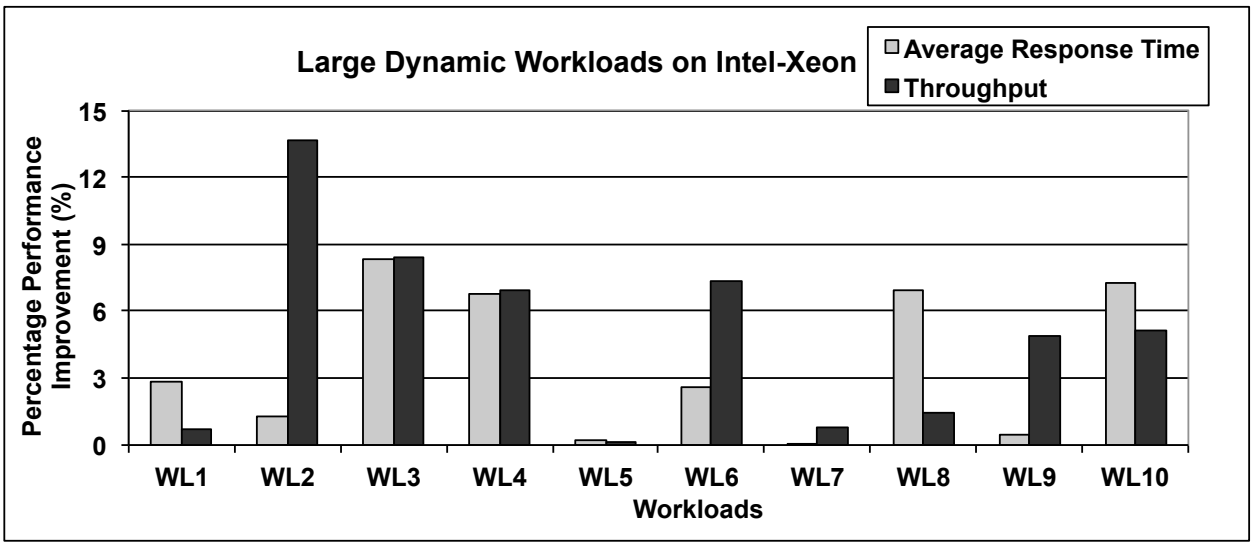
from both PARSEC and NPB to create diversity. Because the arrival time and execution duration of the benchmarks are randomly selected, each workload has one to eight benchmarks simultaneously executing in the system at some point in time. The benchmarks and the parameters of the workloads are described in Table 4.9.

Workload	Parameters
WL1	{12 (BS,3) 34 (DC,7) 50 (CN,4) 68 (FL,6) 85 (RT,5) 101 (IS,3) 114 (FE,5) 141 (SW,6)}
WL2	{25 (SW,6) 28 (SP,7) 89 (SC,6) 90 (DC,5) 123 (FQ,2) 132 (DD,6) 150 (RT,2) 173 (CN,3)}
WL3	{8 (CN,3) 38 (SC,6) 45 (SW,5) 50 (IS,3) 57 (FA,4) 80 (RT,1) 111 (DD,5) 120 (FL,5)}
WL4	{18 (UA,5) 30 (SW,4) 68 (FA,5) 71 (SC,7) 102 (LU,9) 124 (FE,6) 160 (CG,5) 169 (FQ,2)}
WL5	{16 (FQ,8) 12 (FL,3) 20 (EP,2) 28 (BS,9) 40 (CN,5) 51 (FA,10) 60 (DD,2) 72 (UA,4)}
WL6	{5 (IS,7) 56 (RT,8) 85 (LU,8) 109 (CN,5) 111 (EP,9) 119 (BS,1) 127 (FL,9) 131 (VP,8)}
WL7	{84 (FQ,10) 95 (SP,1) 96 (SW,3) 136 (CG,8) 147 (UA,4) 167 (SC,10) 175 (RT,3) 194 (FA,2)}
WL8	{57 (FA,7) 66 (RT,9) 70 (BT,3) 106 (SW,5) 118 (DC,3) 131 (UA,4) 140 (FE,7) 191 (CG,1)}
WL9	{79 (EP,3) 110 (LU,1) 120 (BS,1) 127 (SC,6) 163 (MG,4) 169 (DC,3) 178 (VP,2) 198 (FE,3)}
WL10	{10 (CN,2) 11 (EP,7) 78 (DC,8) 96 (DD,2) 99 (IS,3) 154 (RT,5) 168 (CG,1) 195 (FL,4)}

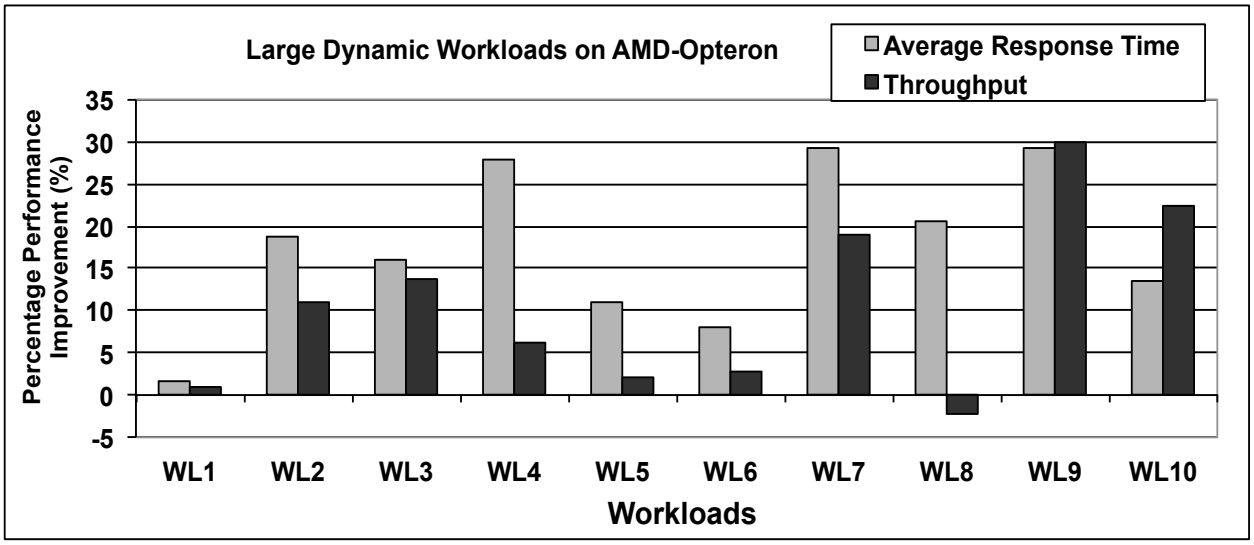
Table 4.9: *Large* Dynamic Workload Set

In Figure 4.26, we observe that $\text{ReSense}_{\text{Performance}}$ improves the average response time and throughput of the workloads up to 8.29% and 13.65%, respectively, on Intel-Xeon and 29.34% and 29.86%, respectively, on AMD-Opteron, over the native OS. On Intel-Xeon, all workloads show improvements for both metrics. The mapping decision of $\text{ReSensor}_{\text{P}}$ is similar to the OS-mapping for $WL1$, $WL5$, and $WL7$. Therefore, the performance differences between the OS and $\text{ReSense}_{\text{Performance}}$ are small.

On AMD-Opteron, most workloads have high performance improvements for both metrics. The benchmarks in $WL1$, particularly RT, SW and BS are not very sensitive to contention for the shared resources on this platform. $\text{ReSense}_{\text{Performance}}$ maps IS's threads on separate processors and L3-caches and OS-mapping also maps the threads randomly on any processor. Because both OS and $\text{ReSensor}_{\text{P}}$ map the threads similarly on the cores, the performance difference between the OS and $\text{ReSense}_{\text{Performance}}$ for $WL1$ is small. The throughput degrades by 2% for $WL8$. $\text{ReSense}_{\text{Performance}}$ co-locates one of the benchmarks in the workload, FE, with the long-running NAS-benchmarks, causing FE's performance degradation. As FE has a lower execution time, it has a higher impact on the throughput equation. Therefore, even



(a) Results on Intel-Xeon



(b) Results on AMD-Opteron

Figure 4.26: Performance results for *Large* Dynamic Workloads, normalized to the native OS ($\text{ReSense}_{\text{Performance}}$ performs better than the OS)

if the average response time improves for this workload, the throughput does not improve over the native OS.

In summary, $\text{ReSense}_{\text{Performance}}$ effectively manages *large* dynamic workloads, consisting of eight randomly selected benchmarks and improves both average response time and throughput using the thread-mappings determined by $\text{ReSensor}_{\text{P}}$.

Dynamic Workloads	Metrics	Intel-Yorkfield	Intel-Harpertown	Intel-Xeon	AMD-Opteron
Small	Average Response Time	2.25 ± 1.58 p-val= 0.0005	3.73 ± 2.68 p-val= 0.005	14.15±13.4 p-val= 0.005	7.45±6.18 p-val= 0.0005
	Throughput	2.70 ± 1.64 p-val= 0.0005	3.27 ± 3.95 p-val= 0.025	4.89±4.03 p-val= 0.005	11.70±6.39 p-val= 0.0005
Medium	Average Response Time	–	–	4.94 ±4.18 p-val= 0.005	11.70±7.45 p-val= 0.0005
	Throughput	–	–	9.005 ± 9.74 p-val= 0.01	16.17±14.80 p-val= 0.005
Large	Average Response Time	–	–	3.67±3.37 p-val= 0.005	17.60±14.19 p-val= 0.005
	Throughput	–	–	4.95±4.45 p-val= 0.0005	10.58±10.8 p-val= 0.01

Table 4.10: Confidence interval of performance improvements for three workloads

4.4.4 Mapping: Discussion and Statistical Analyses

Because we use workloads having randomly selected benchmarks in our experiments, we perform a significance test for the reported average response time and throughput. We assume the null hypothesis that $\text{ReSense}_{\text{Performance}}$ does not improve the average response time and throughput of the workloads over the native OS. As we perform each experiment in two configurations, using the native OS and $\text{ReSense}_{\text{Performance}}$ run-time, each experiment has two distributions, *OS* and *ReSense*. We perform a *t-test* to compare these distributions to determine if *OS* is better than *ReSense* in terms of average response time and throughput [101]. For each dynamic workload on the experimental platforms, we observe that the null hypothesis is rejected with p-value of at most 0.005 for average response time and 0.025 for throughput. It indicates that the probability of $\text{ReSense}_{\text{Performance}}$ improving a workload's average response time and throughput over the OS is very high, at least 99.5% and 97.5%, respectively.

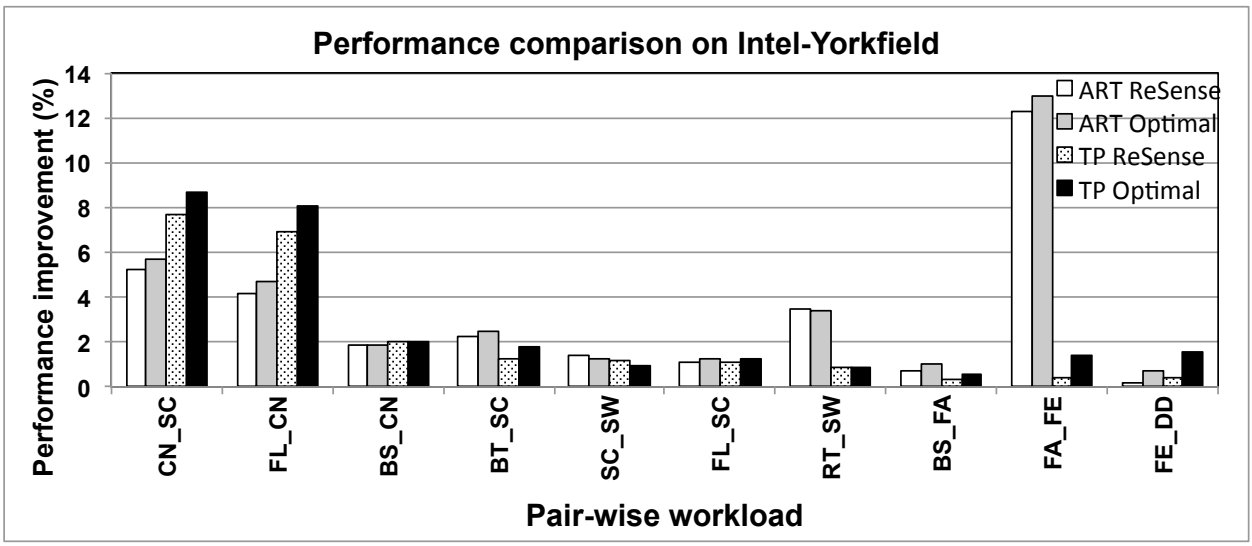
We determine the confidence intervals of the mean improvement of the average response time and throughput provided by $\text{ReSense}_{\text{Performance}}$ over the native OS, shown in Table 4.10. If the confidence interval for any metric is x±y with $p-val = z$, it means that at $(1-z)*100\%$

confidence interval the mean improvement ranges from $x - y$ to $x + y$. From the table, we observe that the lower interval for the average response time is always greater than 0, which indicates $\text{ReSense}_{\text{Performance}}$ always improves the average response times of the three dynamic workload sets on the four platforms using $\text{ReSensor}_{\text{P}}$. For throughput, the lower interval is always greater than 0 except the negligible -0.68% for *Small* on Intel-Harpertown, -0.69% for *Medium* on Intel-Xeon, and -0.22% for *Large* on AMD-Opteron. These negligible negative values are caused by the very small performance degradation of the benchmark that has a smaller execution time than the other applications in the workloads. From the higher interval, we observe that the maximum average response time improvement is 31.79% for *Large* and the maximum throughput improvement is 30.97% for *Medium* on AMD-Opteron, over the native OS.

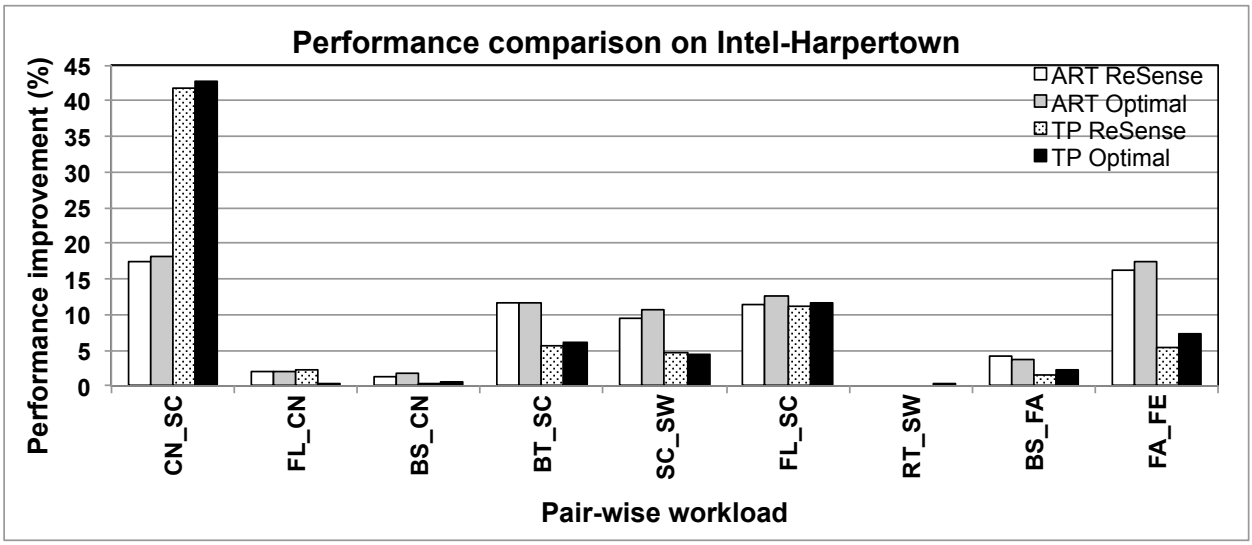
To summarize, the statistical analysis validates $\text{ReSense}_{\text{Performance}}$'s effectiveness to map and improve the performance of multi-threaded applications over the native OS using the thread-mappings determined by $\text{ReSensor}_{\text{P}}$.

4.4.5 Mapping: Performance Comparison with Experimentally Determined Optimal Thread-mapping

To evaluate $\text{ReSensor}_{\text{P}}$'s effectiveness in contention mitigation, we compare application performance obtained from $\text{ReSense}_{\text{Performance}}$- and the experimentally determined optimal thread-mappings. We experimentally determine the optimal performance of a workload by choosing the minimum average response time and maximum throughput of the optimal thread-mapping among all possible thread-to-core-mapping configurations. Throughout this section, when we use "optimal" we mean the experimentally determined optimal. Dynamic workloads have $O(r^{n_1} * r^{n_2} * ... * r^{n_n})$ numbers of different thread-mapping configurations, where r is the number of a particular shared resources on a platform, and n_1, n_2, ..., n_n are the numbers of executing applications at time t_1, t_2, ..., t_n, respectively. As dynamic workloads have such a large number of configurations, it is unfeasible to experimentally



(a) Results on Intel-Yorkfield



(b) Results on Intel-Harpertown

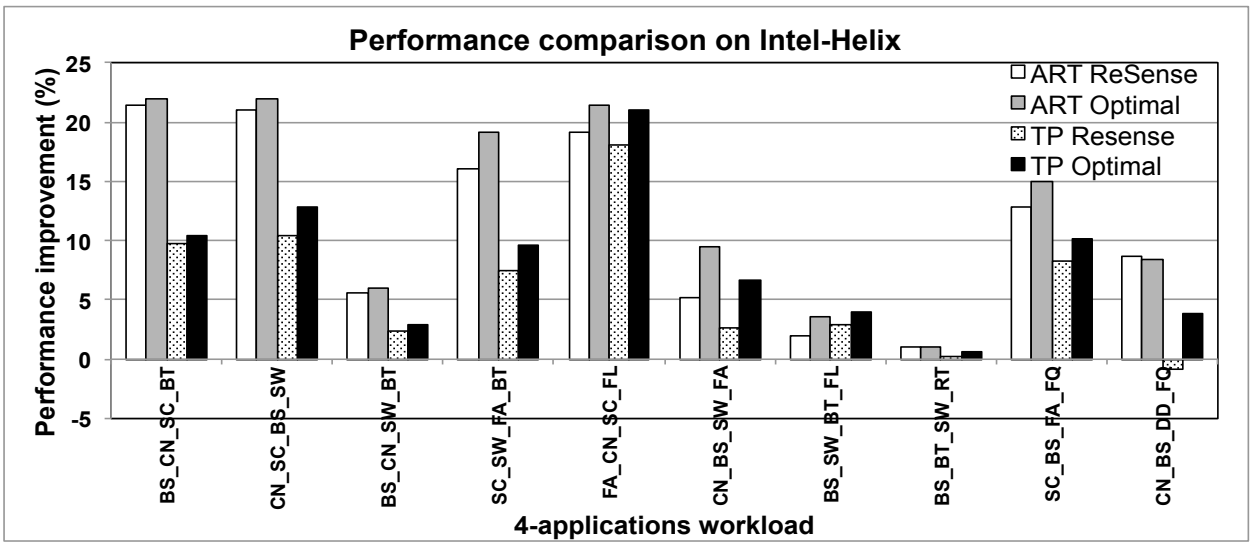
Figure 4.27: Performance comparison between ReSense$_{\text{Performance}}$ and experimentally determined Optimal thread-mapping for pair-wise workloads, normalized to the native OS

determine the optimal performance. Therefore, we choose to use workloads that have all the benchmarks start at the same time and execute for the same number of iterations so that it is feasible to determine the optimal performance.

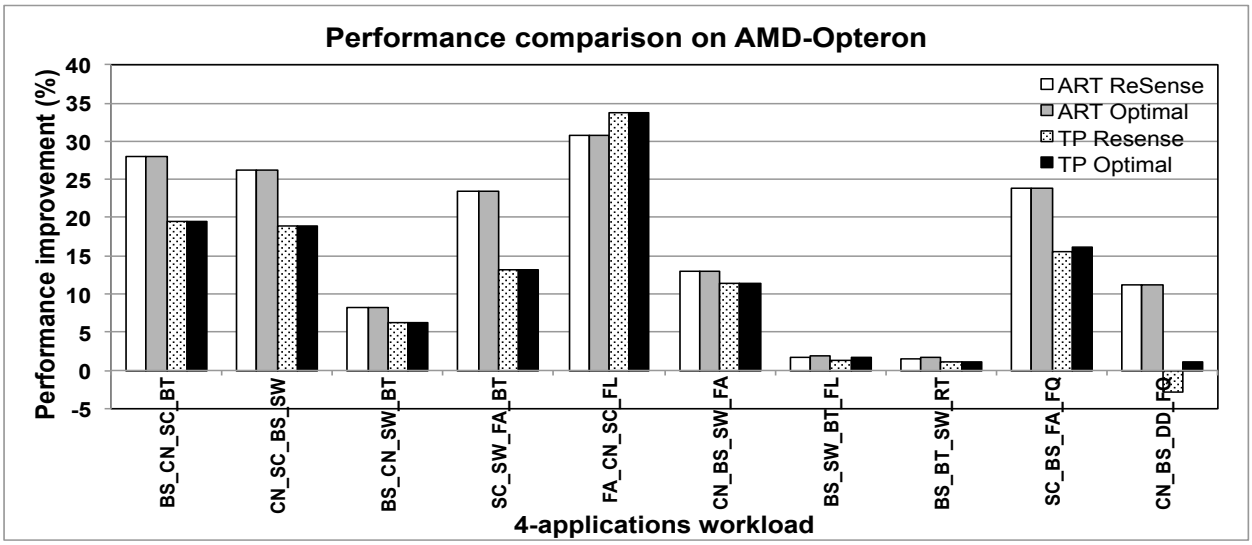
For performance comparison on Intel-Yorkfield and Intel-Harpertown, we run experiments with randomly selected ten pair-wise workloads from the PARSEC benchmark suite. We

do not include the NAS benchmarks in the random selections because those benchmarks have execution times that are too long to finish the experiments for all possible thread-mapping configurations. Figure 4.27 shows the experimental results relative to the native OS, where the X-axis labels the initials of the benchmarks used as the workload. For each workload, the first two bars show average response times (ART) for $\text{ReSense}_{\text{Performance}}$ and experimentally determined optimal mappings, and the last two bars show throughput (TP) for $\text{ReSense}_{\text{Performance}}$ and experimentally determined optimal mapping. In both Figure 4.27(a) and 4.27(b), we observe that $\text{ReSense}_{\text{Performance}}$'s performance improvements are very close to that of the optimal. The average ART difference between $\text{ReSense}_{\text{Performance}}$ and experimentally determined optimal is 0.27% and 0.18% on Intel-Yorkfield and Intel-Harpertown, respectively. The average TP difference between $\text{ReSense}_{\text{Performance}}$ and experimentally determined optimal is 0.49% and 0.10% on Intel-Yorkfield and Intel-Harpertown, respectively. Both the ART and TP differences between $\text{ReSense}_{\text{Performance}}$ and optimal are negligible on both machines, and we conclude that $\text{ReSense}_{\text{Performance}}$ always ensures near-optimal performance on these machines using the thread-mappings determined by $\text{ReSensor}_{\text{P}}$.

For performance comparison on Intel-Xeon and AMD-Opteron, we run experiments with randomly selected ten 4-application workloads from the PARSEC benchmark suite. Figure 4.28 shows the experimental results relative to the native OS. In both Figure 4.28(a) and 4.28(b), we observe that $\text{ReSense}_{\text{Performance}}$'s performance improvements are very close to that of the experimentally determined optimal. The average ART difference between $\text{ReSense}_{\text{Performance}}$ and experimentally determined optimal is 1.49% and 0.05% on Intel-Xeon and AMD-Opteron, respectively. For $CN\_BS\_DD\_FQ$, $\text{ReSense}_{\text{Performance}}$ degrades the throughput by 2% on both machines. Because of the short execution time of the benchmark DD, even if $\text{ReSense}_{\text{Performance}}$ degrades DD's performance by 1 second, it caused the overall throughput degradation because of throughput's definition. The average TP difference between $\text{ReSense}_{\text{Performance}}$ and experimentally determined optimal is 2.08% and 0.49% on Intel-Xeon and AMD-Opteron, respectively. Both the ART and TP differences between



(a) Results on Intel-Xeon



(b) Results on AMD-Opteron

Figure 4.28: Performance comparison between ReSense$_{\text{Performance}}$ and experimentally determined Optimal thread-mapping for 4-applications Workloads, normalized to the native OS

ReSense$_{\text{Performance}}$ and experimentally determined optimal are very small on both machines. Therefore, we can conclude that ReSense$_{\text{Performance}}$ performs very competitively with the optimal mapping using the thread-mappings determined by ReSensor$_{\text{P}}$.

The improvements over the native OS by ReSense$_{\text{Performance}}$ are mainly due to the characterization and use of these characterizations by ReSensor$_{\text{P}}$. If ReSensor$_{\text{P}}$ does not consider

Thread-mappings	2-application workload		4-application workload	
	Avg. Response Time	Throughput	Avg. Response Time	Throughput
C_+	-1.70%	5.57%	14.1%	8.28%
C_-	0.51%	-3.77%	-2.5%	-6.51%
$\text{ReSense}_{\text{Performance}}$	5.07%	7.90%	16.77%	11.82%

Table 4.11: Average performance improvements (positive values) or degradation (negative values) over the native OS, for thread-mappings using fixed positive, fixed negative and characterization-based $\text{SensitivityScores}_{\text{performance}}$

any $\text{SensitivityScore}_{\text{performance}}$ ($\text{SensitivityScores}_{\text{performance}}$ are 0) to determine the thread-mappings, the performance of the workload is the same as the native OS. To further isolate the benefits of the $\text{SensitivityScores}_{\text{performance}}$, we explore the performance of workloads when the $\text{SensitivityScore}_{\text{performance}}$ of each application is set to the same magnitude with positive or negative sign and run the $\text{ReSensor}_{\text{P}}$ algorithm for these two cases. Table 4.11 summarizes these experimental results for the 2-application and 4-application workloads used in Section 4.4.5 on Intel-Harpertown and AMD-Opteron, respectively, relative to the native OS. The rows C_+ and C_- show the performance results of the thread-mappings determined using the fixed positive and negative $\text{SensitivityScore}_{\text{performance}}$ (same magnitude with positive or negative sign), respectively, for all applications in the workloads. The row $\text{ReSense}_{\text{Performance}}$ shows the performance results of the thread-mappings determined by $\text{ReSensor}_{\text{P}}$ using an application's $\text{SensitivityScore}_{\text{performance}}$ from the characterization. From the table we observe that the C_+ thread-mapping degrades application performance for the 2-application workloads, and the C_- thread-mapping degrades application performance for both 2-application and 4-application workloads. Therefore, thread-mappings determined using a fixed positive or negative $\text{SensitivityScore}_{\text{performance}}$ do not ensure application performance improvements.

In contrast, with the computed $\text{SensitivityScore}_{\text{performance}}$ from the characterization, for the 2-application workloads $\text{ReSense}_{\text{Performance}}$ improves the workloads' both average response time by 5.07% and throughput by 7.9% on average, and for the 4-application workloads $\text{ReSense}_{\text{Performance}}$ improves both the workloads' average response time by 16.77% and through-

put by 11.82% on average, relative to the native OS. Therefore, these results show that the computed $\text{SensitivityScore}_{\text{performance}}$ is essential for the workload's improved performance.

4.5 Summary

In this chapter, we addressed the challenges of mitigating shared resource contention in the memory hierarchy caused by multi-threaded applications in modern multi- and many-core machines using ReSense_Performance, a performance instance of the ReSense framework.

In the characterization phase, we instantiated the general methodology of the ReSense framework to characterize a multi-threaded application for both intra- and inter-application contention for the shared resources in the memory hierarchy using performance as the characterization metric. To demonstrate the methodology, we characterized the applications in the widely used PARSEC and NPB benchmark suites for shared-memory resource contention on four different multicore platforms. The characterization revealed several interesting aspects of the benchmark suites. Two of the thirteen PARSEC benchmarks exhibited no intra-application contention for the cache resources at any level of the memory hierarchy. Nine PARSEC benchmarks exhibited inter-application contention for the L2-cache. Contention for the front-side bus and memory controller was a major factor with most the benchmarks and degraded application performance by more than 11%. All benchmarks, except one, performed better when the sibling threads used the same memory socket connection, which reduced remote memory access and its latency. On the other hand, two of the nine NPB benchmarks suffered from intra-application contention for all the shared resources on the platforms including the memory controller, L3-cache, and memory socket. Three out of nine NPB benchmarks suffered from memory socket contention among the sibling threads. Each application's characteristics for a particular shared resource was represented as $\text{SensitivityScore}_{\text{performance}}$, which was determined offline as the application ran solely using the methodology developed for intra-application contention.

In the mapping phase, the ReSense run-time system was instantiated as $\text{ReSense}_{\text{Performance}}$.

The $\text{ReSense}_{\text{Performance}}$ run-time system employed the $\text{ReSensor}_{\text{P}}$ algorithm to map application threads from the input dynamic workloads. $\text{ReSensor}_{\text{P}}$ dynamically determined the thread-mappings of the multi-threaded applications in a workload in the presence of any number of co-runners using each application's $\text{SensitivityScore}_{\text{performance}}$ for a particular platform. The algorithm optimized the objective function of this instance, which was to minimize the workload's average response time and maximize throughput. $\text{ReSense}_{\text{Performance}}$ did not require the application's source code modifications. To determine $\text{ReSense}_{\text{Performance}}$'s effectiveness, $\text{SensitivityScores}_{\text{performance}}$ were determined for 22 benchmarks from PARSEC-2.1 and NPB-OMP-3.3 for the shared resources in the memory hierarchy on four different platforms. Using three different sized dynamic workloads composed of randomly selected two, four and eight co-running benchmarks with randomly selected start times, $\text{ReSense}_{\text{Performance}}$ was able to improve the average response time of the three workloads by up to 27.03%, 20.89%, and 29.34% and throughput by up to 19.97%, 46.56%, and 29.86% respectively, over the native OS using $\text{ReSensor}_{\text{P}}$ on real hardware. By estimating and comparing $\text{ReSensor}_{\text{P}}$'s effectiveness with the optimal thread-mapping for two different workloads, we found that the maximum average difference with the experimentally determined optimal performance was 1.49% for average response time and 2.08% for throughput.

From the results of both characterization and mapping phase, we conclude that the ReSense framework was effectively used to mitigate contention for the shared resources in the memory hierarchy on both multi- and many-core architectures.

Chapter 5

Using ReSense for Reliability

This chapter describes how the ReSense framework is used to develop the reliability instance, ReSense_Reliability, for improving reliability on modern multicore architectures. ReSense_Reliability targets minimizing the effect of soft errors in shared caches on a given CMP platform.

5.1 Introduction

Soft errors induced by alpha particles from packaging and atmospheric neutrons are a significant source of transient errors in modern microprocessors [15, 2]. These transient faults arise from high energetic particles that generate electron-hole pairs when passing through any semiconductor device. These electron-hole pairs cause the transistor nodes to collect charges. When these charges are sufficiently accumulated, they may invert the state of a logic device and create logical faults. These phenomena of state inversions in logic devices are also known as *bit-flips*. Because this type of fault does not result into a permanent failure of a device, these are known as *soft* or *transient* errors [15].

Because of technology scaling, the size of the transistors decreases and the total number of bits on the multicore platforms increases. As a result, the occurrence rate of these soft errors increases because of the higher number of smaller transistors on the system [12, 2].

Therefore, as multicore machines continue to scale up to many-core machines, soft errors pose a significant risk for multicore processors and any systems-on-chip. The soft error failure rate continues to grow significantly especially for exa-scale computing [102]. The growing rate of soft errors increases the probability of higher bit-flips that potentially cause applications to execute incorrectly. The incorrect executions of the applications result in wrong outputs and visible errors, which reduces the reliability and dependability of the targeted system. Therefore, it is crucial to develop techniques to address soft errors and minimize its effect on application execution so that reliable execution can be ensured on CMP machines.

Research efforts have proposed both hardware and software techniques to reduce the effect of soft errors. These techniques can be further classified into two categories: (a) error detection and correction, and (b) error prevention. The techniques that detect and correct errors have a high overhead in terms of performance, area, cost, and power. Such high overhead makes these approaches impractical and unsuitable for a large number of cores on a targeted multicore platform. Therefore, in this research, we focus on error prevention using software techniques to reduce the probability of soft errors affecting application execution.

Software techniques for error prevention include compilation to reduce an application's susceptibility to soft errors [40, 19, 56, 92, 41]. Instruction scheduling is applied to make an application more resilient towards soft errors [94, 95]. These compiling techniques are static approaches to reduce the effect of soft errors on application execution. However, the soft errors affect an application's dynamically during its execution because of its transient nature. Therefore, a software technique that dynamically adjusts an application execution can be more effective in reducing the impact of soft errors, and is orthogonal to the compiling techniques.

Soft errors can affect an application's execution by causing bit-flips in the application's data as it occupies any hardware component during its execution. Such bit-flips in the resources result in wrong application outputs and visible errors to the users. As different applications occupy and use the hardware resources in different ways, the sensitivity to soft

errors in the resources on a particular platform is highly application dependent [13]. For example, consider an application that frequently reads the content of a cache-line, X, during its entire execution. If the content of the cache-line X is corrupted because of soft errors at some point, the application's execution is affected when it reads the same cache-line during its next cache access.

On the other hand, if the data being used by the application occupy a resource for a long period of time, it increases the probability of application execution being affected by soft errors. In the previous example, even when the cache-line X is accessed infrequently by the application, if the duration of X residing in the cache is very long, the probability of its being affected by soft errors increases. Therefore, the probability of an application's execution being affected by the bit-flips from soft errors in any resource, which is defined as an application's *vulnerability* to soft errors [103], depends on the application's usage and occupancy behaviors of that resource.

The effects of soft errors and its severity on application execution also depend on the execution platform and the underlying resources. For example, if the platform has a large cache, then it is more susceptible to soft errors because more die area is exposed to the energetic particles, which increases the probability of potential bit-flips. Therefore, the execution platform's underlying resources can make the application execution and output more sensitive to the effect of soft errors.

The effect of soft errors on a multi-threaded application can be severe for several reasons on a CMP. A multicore platform has multiple copies of the micro-architectural resources, e.g., re-order buffers on each processor core and memory resources, e.g., private and shared caches. A multi-threaded application typically executes with multiple threads of execution that can occupy multiple resources at the same time. Occupying multiple resources increases both the frequency of resource access and the occupancy duration in the resources. This high occupancy increases the probability of bit-flips corrupting the application execution and makes the application more vulnerable to soft errors. The resources on a multicore machine

are heavily occupied when a workload with multiple multi-threaded applications is executed. Depending on the applications that constitute a workload, the level of applications' resiliency to soft errors depends on the resource usage and occupancy behavior of the individual applications. To minimize the probability of soft errors affecting any application's execution, the behaviors that increase vulnerability should be controlled from the application-level. Therefore, in this research, we focus on designing a dynamic, lightweight, and application-level software technique that leverages a multi-threaded application's characteristics to reduce its vulnerability to soft errors.

There are several challenges in minimizing application vulnerability via dynamic application-level software measures. First, the transient errors can occur anytime during an application's execution, and the probability of these error occurrences need to be represented as a metric to quantify its effect. It is important to accurately determine this metric by the software technique so that it can effectively take measures to reduce the probability of soft errors affecting the application execution. Second, soft errors cause bit-flips in the hardware micro-architectural and memory resources that are used by an application during its execution [19]. Therefore, the application-level technique should effectively control the application's occupancy duration in these resources so that its susceptibility to soft errors remains low. Third, being dynamic and transient in nature, soft errors affect different applications in various ways. Therefore, an effective software technique must be able to control the occupancy duration of the resources when any combination of applications executes so that this soft-error effect remains at the minimum level.

To address these challenges using an application-level error prevention technique to reduce the effect of soft errors, we need to understand how application behavior of resource usage and occupancy, along with the underlying resource parameters on the platform, affect the probability of visible errors when the execution is corrupted by soft errors. Once we understand the application behavior that influences its vulnerability to soft errors on a particular platform, we can design and develop an effective application-level error prevention technique, which

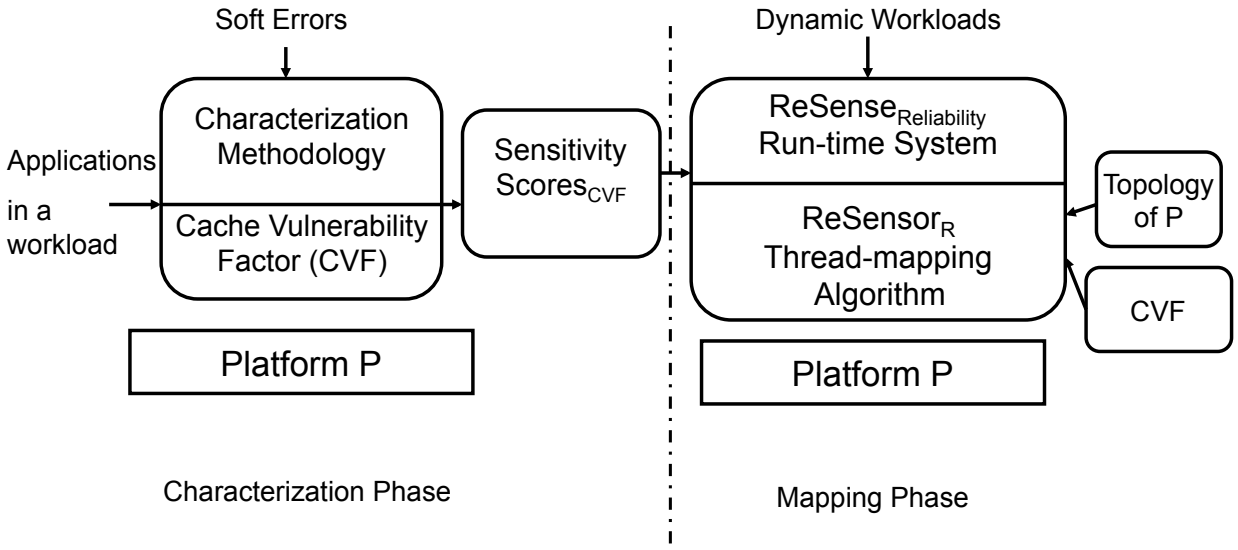


Figure 5.1: Components of the ReSense_Reliability Instance

can minimize the probability that an application output is impacted by soft errors and improve the overall reliability. These insights motivate the creation of ReSense_Reliability, a *reliability instance* of the ReSense framework to minimize the effect of soft errors on the multi-threaded applications in a workload for multicore architectures.

Soft errors can affect an application's output because of the bit-flips created by high energetic particles in both micro-architectural and memory resources. The micro-architectural resources include the re-order buffer, reservation station, and load-store unit, and the memory resources include private and shared caches. Between these two types of resources, it is extremely rare to observe any visible error during application execution because of bit-flips in the micro-architectural resources even under high-flux beam-test condition [104]. On the other hand, as caches occupy a significant die area in the machine's hardware, visible errors because of bit-flips in caches are very frequent [105, 106]. In particular, L2-caches are more vulnerable than L1-data and L1-instruction caches [107]. Therefore, in this work, we focus on the effect of soft errors that occur in the memory resources, particularly shared caches.

Figure 5.1 shows the components of ReSense_Reliability. An application's inherent characteristics and execution behaviors determine its usage and occupancy duration in the

cache resources, which impacts its vulnerability to soft errors. The characterization phase of ReSense_Reliability instantiates the general methodology of the ReSense framework to characterize a multi-threaded application based on its resource occupancy duration in the shared caches. It uses cache vulnerability factor as the characterization metric and calculates $\text{SensitivityScores}_{\text{CVF}}$ for the applications in a workload. Each $\text{SensitivityScore}_{\text{CVF}}$ represents the vulnerability characteristics of a multi-threaded application. These characterizations reveal several insights about the applications' behaviors and help determine the effective mapping to reduce the overall cache vulnerability for a workload.

In the online mapping phase of ReSense_Reliability, we instantiate the run-time system of the ReSense framework as $\text{ReSense}_{\text{Reliability}}$. The $\text{ReSense}_{\text{Reliability}}$ run-time system dynamically manages the mappings of the application threads from a given dynamic workload by employing a thread-mapping algorithm, $\text{ReSensor}_{\text{R}}$ and the pre-determined $\text{SensitivityScores}_{\text{CVF}}$. The $\text{ReSensor}_{\text{R}}$ algorithm determines the thread-mappings of the multi-threaded applications in a workload using the $\text{SensitivityScores}_{\text{CVF}}$ of the applications. The algorithm optimizes the objective function of this instance, which is to minimize the overall cache vulnerability factor to reduce the effect soft errors in caches on a workload's execution.

We assume that the shared caches include redundant bits, e.g., parity bits, error-correcting codes (ECC). When the application execution is exposed to high energetic particles and beams, these redundant bits in the hardware can correct some soft errors depending on the error-correcting capability of the codes. For example, SECDED ECCs can correct a single-bit error and detect a double-bit error. Such error correcting techniques are commonly used to protect shared caches [15]. By reducing the overall cache vulnerability factor, $\text{ReSensor}_{\text{R}}$ reduces the probability of uncorrectable double-bit errors. Furthermore, if there is a performance penalty or overhead for correcting single-bit errors, $\text{ReSensor}_{\text{R}}$ reduces the probability of occurring the penalty.

Minimizing cache vulnerability also leads to reduced failure-in-time (FIT) rate. Utilizing a thread-mapping algorithm demonstrates the use of an application-level technique to

reduce applications' vulnerability to soft errors. The thread-mapping algorithm reduces the probability of visible errors;however, it does not guarantee a 100% error protection.

The outline of this chapter is as follows: Section 5.2 describes the characterization phase of the instance, which includes the characterization methodology and metric to determine a multi-threaded application's vulnerability characteristics for shared caches. Section 5.3 describes the mapping phase, which includes $\text{ReSense}_{\text{Reliability}}$ system and $\text{ReSensor}_{\text{R}}$ algorithm that dynamically map threads from a workload by utilizing the offline pre-determined characterizations of the applications in the workload to improve reliability. Section 5.4.1 describes the experiments performed to characterize the multi-threaded PARSEC benchmarks for vulnerability to soft errors in the shared caches and presents the detailed characterization results and analyses. Section 5.4.3 describes the experimental methodology and evaluation metrics for the mapping phase. Section 5.4.4 discusses the mapping results and analyzes them statistically. Section 5.5 concludes the chapter.

5.2 Characterization for Vulnerability to Soft Errors

In the characterization phase of the ReSense_Reliability instance, we instantiate the general methodology of the ReSense framework to characterize a multi-threaded application based on its vulnerable resource occupancy in shared caches.

5.2.1 Background

According to the general methodology of the ReSense framework, we need a metric to characterize an application based on its resource occupancy and vulnerability. This characterization metric represents an application's characteristics with respect to how much its execution is susceptible to soft errors.

The *architectural vulnerability factor* (AVF) is a well-studied metric to quantify the architectural masking of raw soft errors in any processor structure [103]. AVF represents the probability with which a fault in a processor structure will result in a visible error during the execution of an application. AVF can be calculated as the percentage of time the

structure contains architecturally correct execution (ACE) bits (i.e., the bits that affect the final application output). AVF can be calculated for both micro-architectural structures in the processor and memory resources. For a storage cell in memory resources, AVF is the percentage of cycles that this cell contains ACE bits [103].

The *ACE lifetime* is defined as the percentage of cycles during which a resource contains ACE bits. It represents the lifetime during which an application is vulnerable or susceptible to soft errors. Any bit-flips because of soft errors during the ACE lifetime can result into incorrect application outputs.

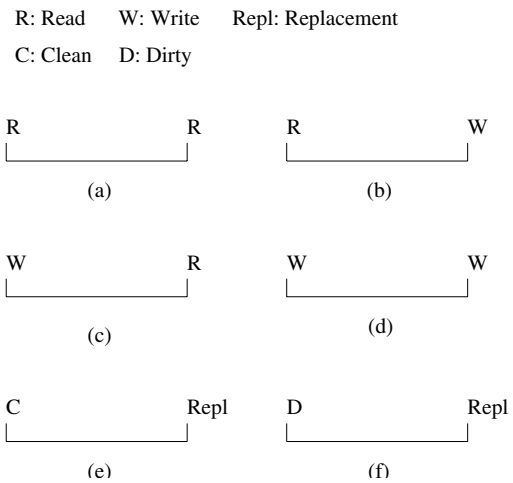


Figure 5.2: Access patterns (intervals) of cache lines (a) Read-Read pattern (b) Read-Write pattern (c) Write-Read pattern (d) Write-Write Pattern (e) Clean-Replacement pattern (f) Dirty-Replacement pattern [1]

The *cache vulnerability factor* (CVF) is the AVF for cache memory resources. CVF represents the probability that a soft error in cache memories can propagate to the processor or other memory hierarchy resources and result in a possible visible error [1]. CVF can be calculated for a cache by performing ACE lifetime analyses for each cache block or cache-line. The *total lifetime* of a cache block is defined as the interval between the time when the cache block is brought into the cache because of a cache miss and the time when this block is replaced or evicted by another cache block, determined by a cache replacement policy. A cache block can be in either a "clean" or "dirty" state depending on whether it has been modified

by the application or not. As the processor is executing the application and performing read-write operations on the cache blocks, we can divide the stream of cache accesses to a cache-line into six different patterns or lifetime intervals for a write-back cache, as shown in Figure 5.2. Among these patterns, bit-flips or soft errors generated during the write-write (W-W) and read-write (R-W) intervals are corrected by the latter write operations, and soft errors occurred between the clean and replacement (C-Repl) interval are discarded at the time of replacement. On the other hand, the bit-flips because of soft errors that occur in the cache-lines between the read-read (R-R) and write-read (W-R) intervals can be loaded to the processor, and the bit-flips during the dirty-replacement (D-Repl) intervals can affect the lower-level memory hierarchy resources after writing back. Therefore, the R-R, W-R, and D-Repl lifetime intervals contribute to the ACE lifetime and CVF of a write-back cache. Any bit-flip during these time intervals in caches can propagate to the processor or other components in the system and result in incorrect application execution and output. The CVF is defined as the fraction of the average time the application is susceptible to soft errors during its execution with respect to its total execution time, and is calculated using the following equation [1]:

$$CVF = \frac{\sum_{i=1}^{n} VulnerableLifetime(block_i)}{\sum_{i=1}^{n} TotalLifetime(block_i)} \tag{5.1}$$

$$= \frac{\sum_{i=1}^{n} Lifetime_{R-R,\ W-R,\ D-Repl}(block_i)}{\sum_{i=1}^{n} TotalLifetime(block_i)} \tag{5.2}$$

Here, $block_i$ is any cache block that is loaded into the cache as the application executes and n is the total number of cache blocks or cache-lines. $VulnerableLifetime$ represents the total time interval that $block_i$ is susceptible to soft errors and $TotalLifetime$ represents the total lifetime of $block_i$. $Lifetime_{R-R,W-R,D-Repl}$ represents the summation of vulnerable lifetimes of $block_i$ for a write-back cache (Equation 5.2), which consists of W-R, R-R, and D-Repl time intervals.

5.2.2 Characterization Metric

The described CVF model is designed for a uniprocessor machine [1]. In this research, as we address cache reliability in a multicore architecture that typically has multiple private and shared caches, we adjust this model to consider multiple caches. When there are multiple caches in multicore machines, the MESI protocol is usually implemented in the hardware to maintain the cache coherency [108]. Therefore, we modify the original CVF model to account for the MESI protocol. In the MESI protocol, when an application executes, each cache-line's status gets updated, which is one of the four states: *modified* (M), *exclusive* (E), *shared* (S), and *invalid* (I). For each cache-line, we maintain a time-stamp that gets updated whenever its status gets changed during application execution. For all the state changes for a cache-line that go to the *modified* and *invalid* state, the difference between the current time-stamp and the time-stamp when it was last updated is *not* considered to contribute to the ACE lifetime. This is because any bit-flips in the cache-line because of a soft error during that time interval is overwritten, and thus cannot change the output of the execution. These state changes fall into the (b) and (d) access pattern shown in Figure 5.2.

On the other hand, the time intervals when the cache-line's state changes from the *shared*, *modified*, and *exclusive* state to the *shared* and *exclusive* state are considered to contribute to ACE lifetime because any soft errors during these intervals may cause a visible error. These state changes fall into the (a) and (c) access pattern shown in Figure 5.2, which contributes to the ACE lifetime. If any dirty cache block is being replaced by a new cache block for a write-back cache, the write-back duration until the dirty cache block is written back to the resources at the lower level of the memory hierarchy, is considered as ACE lifetime. We calculate the CVF for the caches on a multicore platform using Equation 5.3.

$$CVF_{mc} = \frac{\sum_{i=1}^{n} Lifetime_{before-after}(block_i)}{\sum_{i=1}^{n} TotalLifetime(block_i)} \tag{5.3}$$

$$= \frac{\sum_{i=1}^{n} Lifetime_{S-S,S-E,E-E,E-S,M-E,M-S,M-I}(block_i)}{\sum_{i=1}^{n} TotalLifetime(block_i)} \tag{5.4}$$

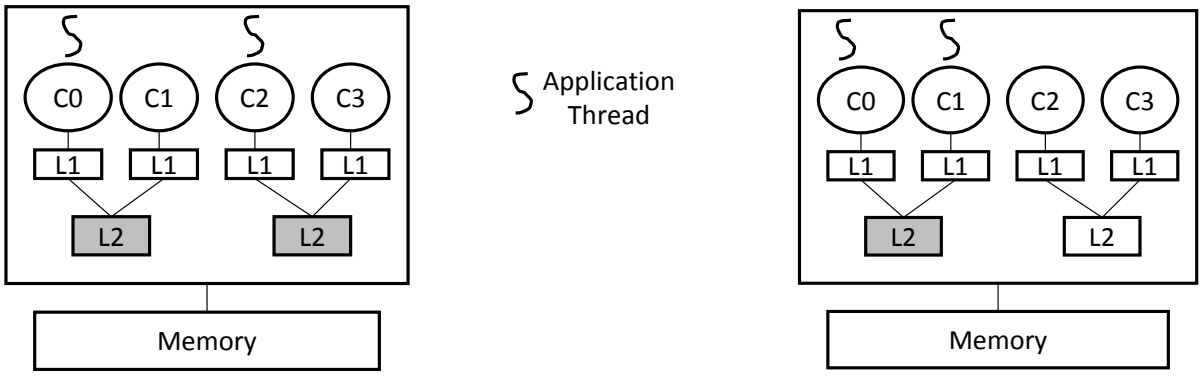
Here, $Lifetime_{before-after}$ represents the vulnerable ACE lifetime of $block_i$ for a cache with MESI protocol, which includes the time intervals between the cache-line state changes from *before* to *after*. The ACE patterns of the cache-line's state changes include S-S, S-E, E-S, E-E, M-S, M-E, and M-I. Among these, the cache-line status change patterns of S-S, S-E, E-S, and E-E represent R-R cache access pattern, and cache-line status patterns of M-S and M-E represent the W-R cache access pattern in Figure 5.2. Lastly, M-I represents the D-Repl access pattern. The summation of the time intervals between these state changes are considered as ACE lifetime, and contribute to an application's vulnerability to soft errors in caches on multicore architectures (Equation 5.4).

For ReSense_Reliability, to characterize a multi-threaded application for its resource occupancy behavior in the shared caches on a multicore platform, we use CVF_{mc} as the characterization metric.

5.2.3 Characterization Methodology

To characterize a multi-threaded application based on its resource occupancy duration, we vary an application's resource occupancy and determine how this occupancy variance impacts its venerability to soft errors. Because a multi-threaded application can be configured to create multiple threads of execution, each thread can use a separate targeted resource. Therefore, a multi-threaded application's resource occupancy can be varied by controlling the number of targeted resources its threads use.

We instantiate the general characterization methodology of the ReSense framework to characterize a multi-threaded application based on its occupancy duration behavior for a targeted resource. We run each multi-threaded application with at least two threads in two characterization configurations. In the first or *non-sharing* configuration, application threads are placed on the cores that use multiple targeted resources, where each thread use a separate targeted resource. In the second or *sharing* configuration, the threads are placed on the cores that use the same targeted resource. The placement of the threads



(a) Non-sharing Configuration (b) Sharing Configuration

Figure 5.3: Configurations to characterize a multi-threaded application for vulnerability to soft errors in shared L2-cache

with respect to the other resources, i.e., the number of other resources the application uses, remains the same across these two configurations. Because the threads are placed on the cores that use different numbers of targeted resources in these two configurations, the application's occupancy duration and usage in the resource are varied during its execution. This variance has an impact on the vulnerable (ACE lifetime) and total lifetime, which affects the application's vulnerability to soft errors in the targeted shared resource.

According to the general characterization methodology described in Section 3.2.2, the experimental platform should have multiple numbers of the targeted resource so that the characterization configurations can be applied.

Figure 5.3 shows the characterization configurations for a multi-threaded application to determine its vulnerability behavior to soft errors in a shared L2-cache. In the non-sharing configuration, application threads are placed on the cores that use separate L2-caches (core C0 and C2), and in the sharing configuration, application threads are placed on the cores that use the same L2-cache (core C0 and C1). In both configurations, the application thread use the same number of L1-caches. The only difference between these configurations is the number of L2-caches used by the application threads, which creates variation in the L2-cache occupancy by the application. CVF_{mc} is determined in both configurations and used to

calculate $\text{SensitivityScore}_{\text{CVF}}$.

5.2.4 $\text{SensitivityScore}_{\text{CVF}}$: Sensitivity Scores for Reliability

As a multi-threaded application's vulnerability characteristics vary depending on the underlying platform and cache configurations, the offline characteristics for different resources is determined *only once* for a particular cache configuration on the targeted platform. The sensitivity score of a multi-threaded application is represented as a $\text{SensitivityScore}_{\text{CVF}}$ for each shared cache on a particular CMP. A multi-threaded application's $\text{SensitivityScore}_{\text{CVF}}$ is calculated using Equation 5.5.

$$SensitivityScore_{CVF} = \frac{(CVF_{mc-non-sharing} - CVF_{mc-sharing}) * 100}{CVF_{mc-non-sharing}} \tag{5.5}$$

Here, $CVF_{mc-non-sharing}$ is the average CVF_{mc} of the shared caches in the non-sharing configuration, and $CVF_{mc-sharing}$ is the CVF_{mc} of the same shared cache in the sharing configuration.

$\text{SensitivityScore}_{\text{CVF}}$ has both a sign and magnitude. The positive sign of the $\text{SensitivityScore}_{\text{CVF}}$ indicates that an application's CVF_{mc} decreases and reliability improves in the sharing configuration, when the threads share the same cache. It means this configuration reduces or eliminates the vulnerable intervals described in Figure 5.2, which causes the overall reduction in the ACE lifetime and consequently CVF_{mc} of the cache. On the other hand, the negative sign of the $\text{SensitivityScore}_{\text{CVF}}$ indicates that the application's vulnerability in caches increases as the threads share the same cache. This means this configuration increases the ACE lifetime of the individual cache block during the application's execution and causes the overall CVF_{mc} to increase. The magnitude of $\text{SensitivityScore}_{\text{CVF}}$ represents how much the CVF_{mc} varies across configurations. Therefore, the sensitivity score for reliability represents how the cache usage affects an application's cache vulnerability and helps design the $\text{ReSensor}_{\text{R}}$ algorithm in the mapping phase.

If the targeted platform has multiple levels of shared caches, then each multi-threaded application is characterized for the shared caches at each level. We store each application's $\text{SensitivityScore}_{\text{CVF}}$ for the shared caches at each level in the $SV_{vulnerability}$ vector. For example, if the targeted platform has two levels of shared caches, then the sensitivity vector, $SV_{vulnerability}$, is a two-element vector for each multi-threaded application in a workload. This vector is used as an input to the thread-mapping algorithm.

5.3 Mapping Co-located Multi-threaded Applications for Reliability

We describe the mapping phase of ReSense framework for the reliability instance, which includes the $\text{ReSensor}_{\text{R}}$ algorithm and the $\text{ReSense}_{\text{Reliability}}$ run-time system.

5.3.1 The $\text{ReSensor}_{\text{R}}$ Thread-mapping Algorithm

Because a $\text{SensitivityScore}_{\text{CVF}}$ represents an application's shared-cache vulnerability to soft errors, the algorithm uses these scores to determine the thread-mappings for any number of multi-threaded applications in a given workload and optimizes the objective function, i.e., minimizes the cache vulnerability factors.

Depending on the cache miss rate characteristics of the co-running applications in the given workload and the application's vulnerability characteristics as $\text{SensitivityScore}_{\text{CVF}}$, there can be two cases:

Special Case: If an application has a higher miss rate when its threads share the same cache, these application threads can cause the CVF_{mc} to decrease by frequently updating cache-lines with the fetched data blocks. If the workload has one or more applications whose high miss rate causes the CVF_{mc} to decrease, then the other applications in the workload are mapped to share the same cache with these cache-intensive application threads as co-runners. Here, the cache-intensive application's cache usage and high miss rate cause the shared cache vulnerability to reduce.

General Case: If none of the applications in the workload has high enough cache misses that lower the overall shared CVF_{mc}, then the algorithm considers the sign and magnitude of the $\text{SensitivityScore}_{\text{CVF}}$ for the applications in the workload and maps them by choosing the mapping configuration that lowers the ACE lifetime according to their characterization.

Similar to the $\text{ReSensor}_{\text{Generic}}$ thread-mapping algorithm of the framework, depending on the number of applications in the given workload and the number of targeted resources, there can be two scenarios. Here, by targeted resources we mean the shared caches. Both cases described in the previous paragraphs are applicable in these two scenarios. We describe these scenarios and the intuition behind the algorithms in the following paragraphs:

Scenario 1: There are the same or more targeted shared resources on the targeted platform than the total number of applications in the given workload. In this scenario, if the workload has applications that have high cache miss rates causing low CVF_{mc}, then the applications are mapped to use the same targeted shared cache, according to **Special Case**. Otherwise, the mapping should be done according to **General Case**.

Scenario 2: There are a fewer targeted shared resources on the targeted platform than the total number of applications in the given workload. In this scenario, as the number of applications is higher, the applications are mapped such that threads from one application share the same targeted resource with the threads from a different application. Intuitively, this mapping causes most of the cache access patterns to *not* contribute to ACE lifetime because threads from different applications have different access patterns and can cause potential cache misses. These cache misses can replace the content of the cache-lines and lower the CVF_{mc}. Cache vulnerability can be further reduced by grouping the applications more intelligently based on their cache characteristics. If there is any application that satisfies the condition described in **Special Case**, then this application's threads are spread across the shared caches and mapped to use the same shared cache with the other applications in the workload. The high cache misses of the cache-intensive co-running application would lower the CVF_{mc} by reducing the ACE lifetimes of the cache blocks. On the other hand, if

the workload does not have any application satisfying the condition for **Special Case**, then the thread-mappings are followed for **General Case**.

Considering these scenarios and cases, we instantiate the $\text{ReSensor}_{\text{R}}$ algorithm from the $\text{ReSensor}_{\text{Generic}}$ algorithm to determine the thread-mappings of multi-threaded applications using their cache vulnerability characterizations, $\text{SensitivityScores}_{\text{CVF}}$.

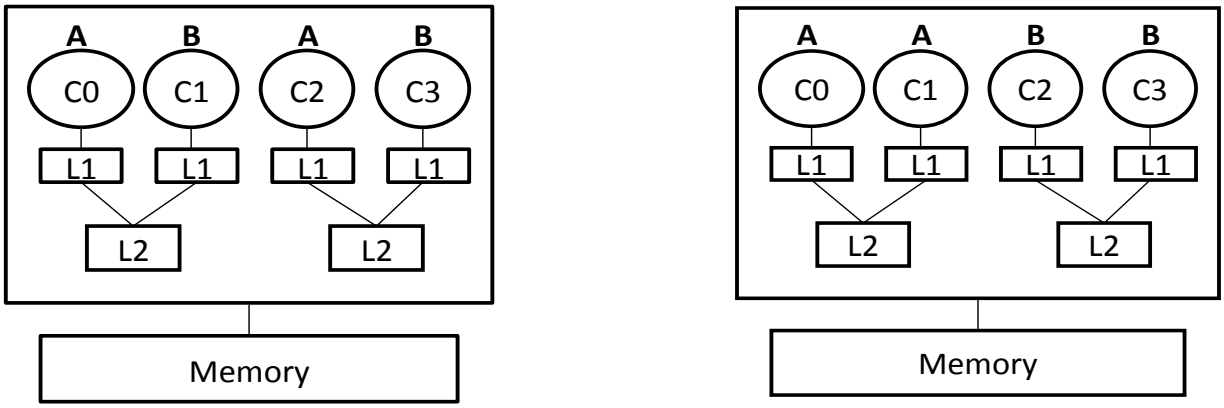
Algorithm 4 gives the $\text{ReSensor}_{\text{R}}$ algorithm, which maps the application threads from the input workload WL using the applications' $\text{SensitivityScores}_{\text{CVF}}$. Platform P can have multiple levels of shared caches, and the vulnerability characterizations of the applications in WL for the caches at each level are stored in $SV_{vulnerability}$. The algorithm stores the total number of multi-threaded applications and the applications in $nApps$ and $[Apps]$ variable (line 2, 3), respectively.

The algorithm determines the thread-mappings of each application considering the resources at the bottom of the memory hierarchy to the top (line 5), i.e., from L3-caches to L2-caches. At each level of the memory hierarchy, the algorithm identifies the shared cache R at that level and the total number of R (line 6 and 7). It computes two arrays: the set of cores that share or use the same R, $[C_+]$, and the set of cores that do not share the same R, $[C_-]$ (lines 8, 9). These two arrays are later used to determine the cores on which the application threads are mapped. The $[VulApps]$ variable is initialized to be empty in the beginning of the loop iteration (line 4). The algorithm computes the set $[VulApps]$ for handling **Special Case**, which stores the applications that have significantly higher miss rates in shared caches causing its CVF_{mc} to decrease in the sharing configuration (line 10). It updates the $[Apps]$ variable to save the rest of the applications (line 11). The algorithm then collects the $\text{SensitivityScores}_{\text{CVF}}$ of the applications in the updated $[Apps]$ for resource R in $[SS_V]$ (line 13). It then sorts the $[SS_V]$ array according to the magnitude of the $\text{SensitivityScore}_{\text{CVF}}$ in descending order so that the application with the most sensitivity is prioritized during mapping (line 14).

Now depending on the values of NR and $nApps$, there are two scenarios described earlier.

Algorithm 4 The ReSensor$_{\mathrm{R}}$ Algorithm: Mapping application threads to minimize overall vulnerability to soft errors in shared caches

INPUT: Workload WL, Topology of the experimental platform P, Sensitivity vector $SV_{vulnerability}$ of the applications in WL on P
$nApps \leftarrow$ total number of multi-threaded applications in WL
$[Apps] \leftarrow$ set of all multi-threaded applications in WL
$[VulApps] \leftarrow \emptyset$
for each level MHL in the memory hierarchy of P **do**
  $R \leftarrow$ shared cache at MHL
  $NR \leftarrow$ number of R at MHL
  $[C_+] \leftarrow$ set of cores that use or share the same R on P
  $[C_-] \leftarrow$ set of cores that do not use or share the same R on P
  $[VulApps] \leftarrow$ set of cache-intensive multi-threaded applications, whose higher cache miss rate in R causes lower CVF_{mc}
  $[Apps] \leftarrow [Apps] - [VulApps]$ / * Compute set difference */
  $nVulApps \leftarrow$ number of applications in $[VulApps]$
  $[SS_V] \leftarrow SV_{vulnerability}[R]$ of the applications in $[Apps]$
  sort $[SS_V]$ array in descending order of the magnitude of the SensitivityScore$_{\mathrm{CVF}}$ and re-arrange $[Apps]$ accordingly
  if $NR >= nApps$ **then**
    /* Scenario 1: equal or more shared resources than the number of applications */
    if $[VulApps] != \emptyset$ **then** /* Special Case */
      n $\leftarrow$ maximum ($nVulApps$, $nApps - nVulApps$)
      for (i = 0 ; i < n ; i++) **do**
        if $[C_+]$ has available core(s) **then**
          map $Apps[i]$-threads and $VulApps[i]$-threads on the available cores from $[C_+]$
        else
          map $Apps[i]$-threads and $VulApps[i]$-threads on any core on P
        end if
      end for
    else if $[VulApps] == \emptyset$ **then** /* General Case */
      for (i = 0 ; i < $nApps$; i++) **do**
        if $SS_V[i] > 0$ AND $[C_+]$ has available core(s) **then**
          map $Apps[i]$-threads on the available cores from $[C_+]$
        else if $SS_V[i] < 0$ AND $[C_-]$ has available core(s) **then**
          map $Apps[i]$-threads on the available cores from $[C_-]$
        else
          /* $[C_+]$ or $[C_-]$ does not have available core(s) */
          map $Apps[i]$-threads on any core on P
        end if
      end for
    end if



(a) Scenario 1: Special Case

(b) Scenario 1: General Case

Figure 5.4: Mapping decision for the two cases in Scenario 1

Scenario 1: This scenario handles the mapping when the number of resources is equal or more than the number of applications (line 15 - 37). There are two cases to consider depending on if [$VulApps$] variable is empty or not. For **Special Case**, when [$VulApps$] is not empty, application threads from [$VulApps$] are mapped on the cores from [C_+] to share the same cache with the application threads from [$Apps$] (line 21). Sharing the same cache with the cache-intensive application, which has the important characteristics that more cache misses decrease its CVF_{mc}, is effective in minimizing the overall cache vulnerability by reducing the ACE lifetime of the cache-lines.

For example, consider a workload that has two multi-threaded applications, A and B each with two threads, and these applications are to be mapped on the quad-core platform (e.g., Intel-Yorkfield), shown in Figure 5.4. Here, both $nApps$ and NR equal 2, and [$Apps$] = [A, B], [C_+]=[{C0,C1}, {C2,C3}], and [C_-]=[{C0,C2}, {C1,C3}]. Let us assume that A is a cache-intensive application, whose CVF_{mc} decreases because of high cache miss rate. A is saved in [$VulApps$] and [$Apps$] is updated to contain only B. As [$VulApps$] array is non-empty, the threads from the applications in [$VulApps$] are mapped with the threads from the applications in [$Apps$] on the cores from [C_+] to share the same L2-cache. The final mapping is shown in Figure 5.4(a).

For **General Case**, where [$VulApps$] is empty, the algorithm considers the applications'

SensitivityScores$_{\text{CVF}}$, which are stored in $[SS_V]$, to determine the thread-mappings (line 26 - 36). The $[SS_V]$ array is sorted according to the magnitudes of SensitivityScores$_{\text{CVF}}$ in descending order, so that the most sensitive application for vulnerability is prioritized. The algorithm then determines the mappings of the application threads according to the sign of its SensitivityScore$_{\text{CVF}}$. If the application has a positive SensitivityScore$_{\text{CVF}}$, then the threads are mapped on the cores from $[C_+]$ (line 28 - 29). This mapping helps lowering the CVF_{mc} for those applications, where sharing the same cache lowers its vulnerable lifetime interval. If the application has negative SensitivityScore$_{\text{CVF}}$, then the threads are mapped on the cores from $[C_-]$ (line 30 - 31). This mapping helps to lower the CVF_{mc} for those applications, where using separate caches lowers its D-Repl lifetime interval. If there is no core left in $[C_+]$ or $[C_-]$ array, the threads can be mapped on any available core (line 34).

For the same workload and platform described above, let us now assume that none of the applications is cache-intensive with lower CVF_{mc}, and the conditions for **Special Case** is not satisfied. Assume that the SensitivityScore$_{\text{CVF}}$ for application A and B are $-a_R$ and $-b_R$, respectively, and $|a_R| > |b_R|$. Here, the algorithm first considers application A because it has a higher magnitude. It maps the threads on the cores from $[C_-]$ because its SensitivityScore$_{\text{CVF}}$'s sign is negative. Then it maps the second application, B's threads on the remaining cores from $[C_-]$ considering its negative SensitivityScore$_{\text{CVF}}$. The final mapping of the application threads is the same as shown in Figure 5.4(a).

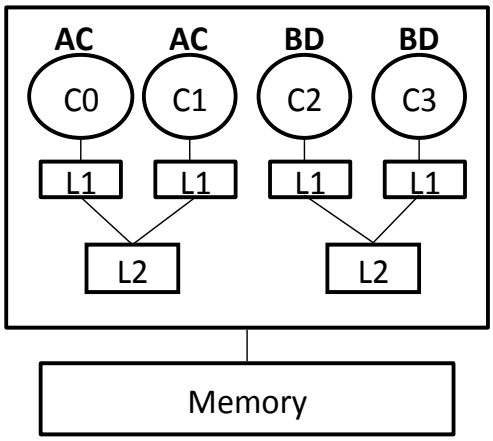
When the signs of the SensitivityScore$_{\text{CVF}}$ for two applications are different, the algorithm already considers the most-sensitive application first and the final mapping does not impact the less-sensitive application's vulnerability significantly. For example, let us assume that that the SensitivityScore$_{\text{CVF}}$ for application A and B are $+a_R$ and $-b_R$, respectively and $|a_R| > |b_R|$. Because A has a higher magnitude of SensitivityScore$_{\text{CVF}}$, the algorithm maps A's threads onto the cores from $[C_+]$ considering its positive sign of the sensitivity score. It then maps the B's thread on the available cores from $[C_-]$. Here, application A has a higher sensitivity to sharing L2-cache and higher probability of lowering the vulnerable interval than

the alternate mapping. Therefore, the algorithm chooses the mapping configuration, shown in Figure 5.4(b), to minimize the overall cache vulnerability.

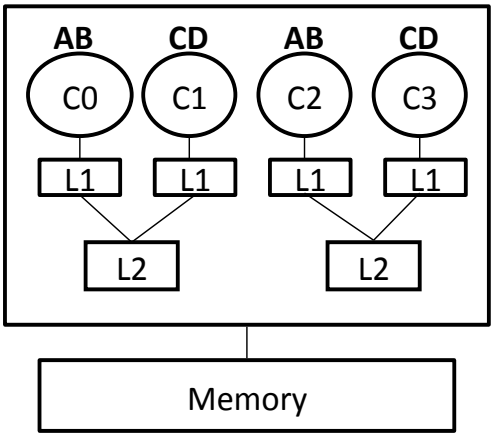
Algorithm 4 The $\text{ReSensor}_{\text{R}}$ Algorithm: Continued

```
38:   else
39:     /* Scenario 2: fewer shared resources than the number of applications */
40:     if [VulApps] ! = ∅ then /* Special Case */
41:       n ← maximum (nVulApps, nApps − nVulApps)
42:       for ( i = 0 ; i < n ; i++ ) do
43:         if [C+] has available core(s) then
44:           map Apps[i]-threads and VulApps[i]-threads on the available cores from [C+]
45:         else
46:           map Apps[i]-threads and VulApps[i]-threads on any core on P
47:         end if
48:       end for
49:     else if [VulApps] == ∅ then /* General Case */
50:       for ( i = 0 ; i < nApps / 2 ; i++ ) do
51:         if SSV[i] > 0 AND [C+] has available core(s) then
52:           map Apps[i]- and Apps[nApps − i − 1]-threads on the available [C+]-cores
53:         else if SSV[i] < 0 AND [C−] has available core(s) then
54:           map Apps[i]- and Apps[nApps − i − 1]-threads on the available [C−]-cores
55:         else
56:           /* [C+] or [C−] does not have available core(s) */
57:           map Apps[i]- and Apps[nApps − i − 1]-threads on any core on P
58:         end if
59:       end for
60:     end if
61:   end if
62: end for
```

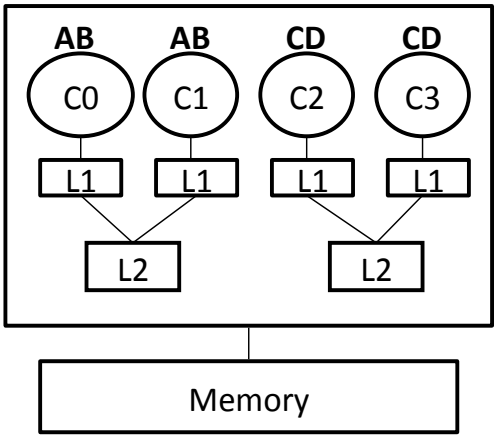
Scenario 2: This scenario handles the mapping when the number of resources is less than the number of applications in a workload (line 39 - 61). Similar to the previous scenario, there can be two cases depending on the emptiness of $[VulApps]$. If $[VulApps]$ has applications that satisfy the condition of **Special Case** (lower CVF_{mc} for higher cache miss rate), then the mapping algorithm pairs the applications from $[VulApps]$ and the remaining applications from $[Apps]$ to share the same cache (line 41 - 48). This mapping causes the cache-intensive application to increase the cache miss rates in the shared caches, which reduces the ACE lifetime of the cache blocks and consequently, decrease the overall CVF_{mc}.



(a) Scenario 2: Special Case



(b) Scenario 2: Special Case



(c) Scenario 2: General Case

Figure 5.5: Mapping decision for the two cases in Scenario 2

For example, consider a workload that has four multi-threaded applications, A, B, C, and D each with two threads, and these applications are to be mapped on the same quad-core platform, shown in Figure 5.4. Here, $nApps$ equals 4 and NR equals 2, and $[Apps] = [A, B, C, D]$. Let us assume that A and B are the cache-intensive applications, whose CVF_{mc} decreases because of high miss rate in the shared caches. $[VulApps]$ contains application A and B and $[Apps]$ is updated to contain application C and D. As $[VulApps]$ is non-empty, the algorithm maps A's threads with C and B's threads with D to use the same shared caches. The final mapping is shown in Figure 5.5(a). If there is only one application (assume A) that satisfies the condition of **Special Case**, then A's threads are spread across the

shared caches so that CVF_{mc}s of both L2-caches are lowered. The rest of the applications are spread across the two shared cache such that their threads use the same shared L2-cache with A's thread. The final mapping for this case is shown in Figure 5.5(b).

For **General Case**, where $[VulApps]$ is empty, the algorithm considers each application's $\text{SensitivityScore}_{\text{CVF}}$, which is stored in $[SS_V]$, to determine the thread-mappings (line 50 - 59). As the $[SS_V]$ array is already sorted according to the magnitude of the $\text{SensitivityScore}_{\text{CVF}}$ in descending order, the algorithm considers the applications with complementary behavior together and prioritizes the more sensitive application while mapping. The algorithm selects the most-sensitive and least-sensitive applications and then determines the mappings of the application threads according to the sign of the most-sensitive application's $\text{SensitivityScore}_{\text{CVF}}$, which is similar to how $\text{ReSensor}_{\text{Generic}}$ maps application threads in this scenario. If the more sensitive application has a positive $\text{SensitivityScore}_{\text{CVF}}$, then the threads of the applications are mapped on the cores from $[C_+]$. The algorithm selects this mapping to lower the shared cache's vulnerable lifetime interval. If the more sensitive application has a negative $\text{SensitivityScore}_{\text{CVF}}$, then the threads are mapped on the cores from $[C_-]$.

Let us consider the same four applications [A, B, C, D] to be mapped on a quad-core machine. Here, we assume that none of the applications satisfies the condition of **Special Case**. Let us assume, $\text{SensitivityScores}_{\text{CVF}}$ of these four applications after sorting are $[+a_R, -c_R, +d_R, -b_R]$, where $|a_R| > |c_R| > |d_R| > |b_R|$. For this workload, the algorithm considers the most-sensitive application A and the least-sensitive application B and maps them to share the same L2-cache (on $[C_+]$-cores) because of the positive sign of A's $\text{SensitivityScore}_{\text{CVF}}$. Then it considers the next most-sensitive application C and D and maps the threads on the remaining cores. The final mapping for this case is shown in Figure 5.5(c).

The algorithm terminates when there are no applications left in the workload whose thread-mappings are not determined.

5.3.2 TheReSense$_{\text{Reliability}}$ Run-time System

The ReSense$_{\text{Reliability}}$ run-time system is instantiated from the ReSense framework. The ReSense$_{\text{Reliability}}$ run-time system detects any thread/application creation and termination, and employs the ReSensor$_{\text{R}}$ thread-mapping algorithm to adjust the application mappings of the input workload. The run-time system takes SensitivityScores$_{\text{CVF}}$ of the applications in a workload as input from the characterization phase and passes these scores and the set of applications whose thread-mappings need to be determined as parameters to ReSensor$_{\text{R}}$. The ReSensor$_{\text{R}}$ algorithm determines the thread-mappings of the applications in the passed workload from ReSense$_{\text{Reliability}}$ using the offline pre-computed SensitivityScores$_{\text{CVF}}$. It optimizes the objective function of this instance, which is to minimize the shared cache vulnerability to soft errors.

5.4 Evaluation of the ReSense_Reliability Instance

In this section, we describe the experimental results of the characterization and mapping phase for the ReSense_Reliability instance of the framework. In the characterization phase, we characterize the PARSEC benchmarks using the methodology described in Section 5.2.3. We present the characterization results in Section 5.4.1 and summary in Section 5.4.2. We present the evaluation results of the mapping phase in Section 5.4.3.

Since we need in-depth and detailed information about the lifetime of individual cache-lines, we use a simulation infrastructure for both the characterization and mapping phase of ReSense_Reliability. For our experiments, we use Simics, which is a timing-accurate full system simulator [109]. Simics is capable of running a full Linux operating system. Therefore, it is used to map application threads to the cores according to the characterization methodology using the *set-affinity* system call.

The targeted platform that we use in our experiments is a four-core machine. We could simulate a multicore machine with more than four cores. However, for the limitations of the simulator capability and its execution time, we choose the targeted platform to have a small

Parameters	Values
Platform	four-core x86 processor (Tango)
Operating System	Linux Kernel version 2.6.15
Cache block size	64 bytes
Cache replacement	Least-recently-used (LRU) policy
Cache coherency protocol	MESI protocol
L1-cache	32KB/core 2-way set associative
L1-cache write policy	write-back, write-allocate
L1-cache latency	1 cycle
L2-cache	4MB 4-way set associative, shared
L2-cache write policy	write-back, write-allocate
L2-cache latency	4 cycle
Memory latency	200 cycles

Table 5.1: Simics configuration for the targeted experimental platform

number of cores, which is typical for a desktop machine. The characterization and mapping phases would work for any cache or system configuration. We choose similar architecture and resource topology as Intel-Yorkfield, described in Figure 4.3(a). The targeted machine has 32KB of L1-D and L1-I cache, which are private for each core. There are two L2-caches, each shared by a pair of cores. The L1-D and L2-caches maintain coherency by the MESI protocol. Both L2-caches are connected to the memory. The details of the simulation configuration are shown in Table 5.1.

We implement the CVF model [1] using the *g-cache* module of the simulator, which supports the MESI protocol in maintaining coherent caches for the targeted multicore machine. We calculate the CVF_{mc} for shared L2-cache using Equation 5.3.

We use the multi-threaded applications from the PARSEC benchmark suite [38]. We choose to use PARSEC benchmark suite because it has a representative set of applications with diverse cache access behaviors that we can analyze for cache vulnerability. Among the thirteen benchmarks in the suite, we characterize eight applications. We select the applications that create worker threads in the beginning of the application execution so that we can analyze CVF_{mc} for both caches on the targeted platform. We do not use *blackscholes* and *canneal*, because these two benchmarks have only the main thread during a significant

portion of their execution duration. We do not use *dedup* as the targeted platform did not have enough memory to execute the benchmark. We do not use *vips* and *ferret* as these applications required a special library, which is not included in the operating system image of the simulator. For all our experiments, we use the largest input set of each benchmark, *simlarge*, for the evaluation using a simulation environment.

5.4.1 Characterization: Experimental Details and Results

We characterize a multi-threaded application for its resource occupancy behavior in the shared caches on the targeted experimental platform according to the methodology presented in Section 5.2.3. As the targeted platform has two cores that share the same shared cache, we configure each benchmark to use two threads so that we can vary the number of resources the application uses.

We run each PARSEC benchmark in two characterization configurations. In the first or non-sharing configuration, we place the application threads on the cores such that each thread uses a separate L2-cache. The application threads are placed on core C0 and C2 (shown in Figure 5.3(a)) and use two L2-caches. In the non-sharing configuration, we place the application threads on the cores such that both threads use the same L2-cache (shown in Figure 5.3(b)). Both configurations place the application threads to use the same number of L1-caches (in this case, 2). The only difference between these configurations is how the threads are placed on the cores with respect to the shared L2-cache. Since these two characterization configurations vary the number of L2-caches the application uses, there is a variation in the resource occupancy created by the application, which affects the application's vulnerability to L2-cache. We measure the application's average CVF_{mc} in both configurations for the L2-cache and calculate its $\text{SensitivityScore}_{\text{CVF}}$ using Equation 5.5.

5.4.1.1 Characterization Results and Analyses

In this section, we present and analyze the experimental results of the PARSEC benchmarks for their cache vulnerability characterizations. The eight benchmarks used in the experiments are:

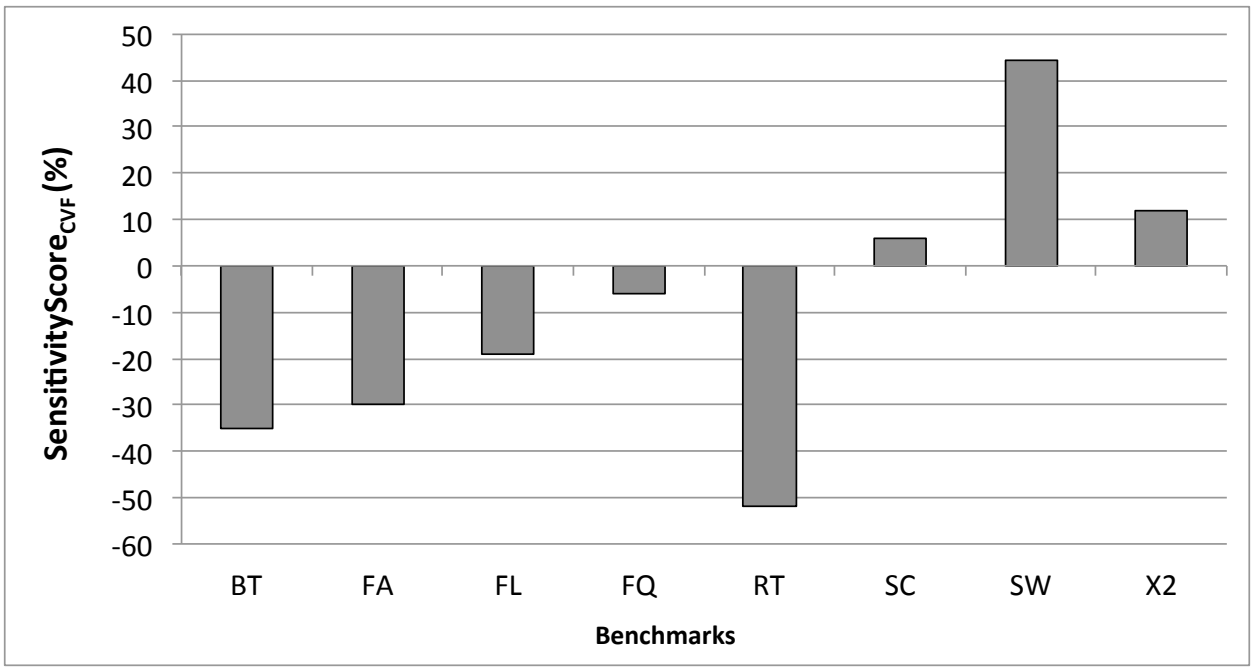


Figure 5.6: Experimental results of vulnerability characterizations of the PARSEC benchmarks for a write-back shared L2-cache, represented as SensitivityScore$_{\mathbf{CVF}}$. Negative values mean increased CVF_{mc}, positive values mean decreased CVF_{mc} in the *sharing* configuration.

bodytrack (BT), *facesim* (FA), *fluidanimate* (FL), *freqmine* (FQ), *raytrace* (RT), *streamcluster* (SC), *swaptions* (SW), and *x264* (X2).

Figure 5.6 shows the SensitivityScore$_{\text{CVF}}$ of the selected PARSEC benchmarks for a write-back, write-allocate shared L2-cache. From the figure we observe that the thread-mapping configurations with respect to sharing the L2-cache has significant impact on the CVF_{mc}, where the difference in magnitude ranges from 5% to more than 50%. Therefore, we can conclude that we can utilize thread-mapping, an application-level technique, in order to control an application's vulnerability to soft errors in caches.

The sharing configuration decreases L2-cache vulnerability for *streamcluster*, *swaptions*, and *x264* (positive SensitivityScore$_{\text{CVF}}$). Figure 5.7 shows the plot of the samples collected during the execution of *streamcluster* on the simulated target platform, when the application threads share the same L2-cache (sharing configuration). The X-axis of the plot shows the different sample instances during *streamcluster*'s execution, and the Y-axis shows the

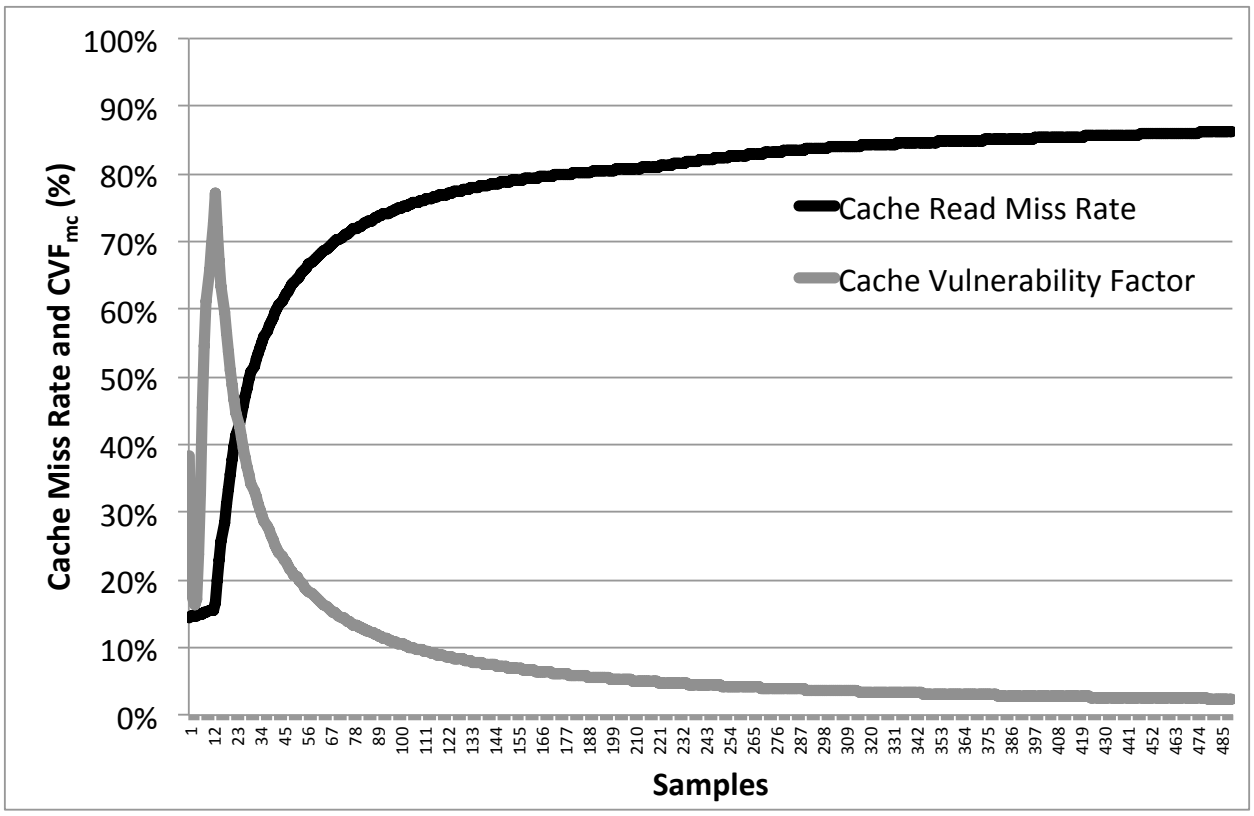


Figure 5.7: Plot of cache miss rate and CVF_{mc} samples during the execution of *streamcluster*

percentage value of the cache read miss rate and CVF_{mc}. The black line shows the plot of the cache miss rate, and the grey line shows the plot and variations in CVF_{mc} values. From this figure, we observe that as the cache read miss rate increases, the CVF_{mc} for the shared cache decreases. The high cache miss rate is caused by the streaming behavior of *streamcluster* and each cache-line being sequentially accessed by the worker threads. When these threads share the same cache, the threads undergo more cache misses than the non-sharing configuration and replace the existing cache blocks with new cache blocks. As the cache blocks get replaced with the new blocks, the timestamp associated with each cache-line gets updated for a cache-line "write" operation and most of these cache access pattern becomes W-W, which does *not* contribute to the ACE lifetime of the shared cache. Therefore, increasing the cache miss rate by creating cache contention reduces the CVF_{mc} of the shared cache. For *x264*, CVF_{mc} reduces in L2-cache sharing configuration and we observed about 12% increase in

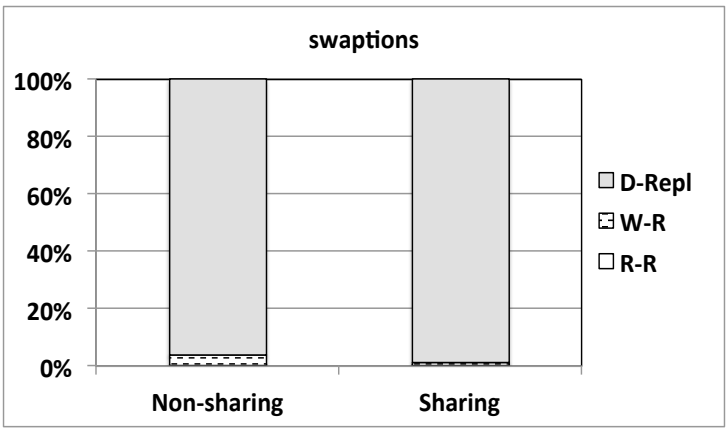


Figure 5.8: Percentage ACE lifetime intervals during the execution of *swaptions*

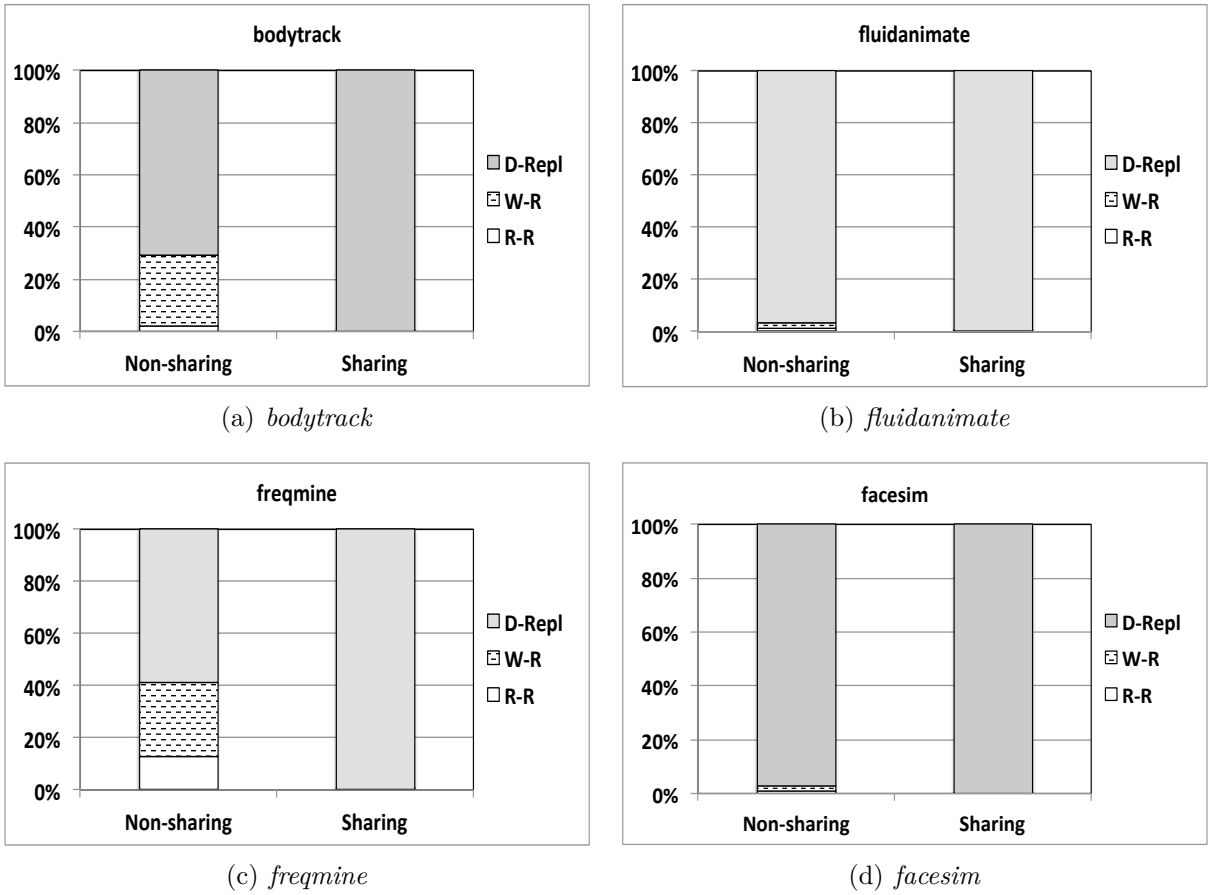
cache read miss rate, compared to the non-sharing configuration. This behavior can also be explained in the same way. As *x264* threads share the same cache, they contend for the cache-lines among themselves. Because of such intra-application contention, the cache blocks brought into the cache-line replaces the old cache blocks, creating large number of W-W intervals, which decreases the ACE lifetime, consequently CVF_{mc} of the L2-cache.

For *swaptions*, both CVF_{mc} and cache miss rate decrease in the sharing configuration when the sibling threads share the same cache. However, the magnitude of cache miss rate do not change significantly. The cache miss rate decreases from 10% to 8%, whereas the CVF_{mc} decreases from 38% to 20%. To further analyze how L2-cache sharing configuration changes *swaptions*' CVF_{mc} for L2-cache, we collect the lifetime data of various ACE intervals that are defined in Figure 5.2. Figure 5.8 shows the stacked graph of *swaptions*' lifetime breakdown for the three ACE cache-line access pattern for a write-back cache. From the figure, we observe that the non-sharing configuration has W-R lifetime interval, which is much smaller in the lifetime breakdown for the sharing configuration. This happens because in the non-sharing configuration, the *swaptions* threads access twice as many cache blocks than that of the sharing configuration. Therefore, some cache-lines in the non-sharing configuration have data that have a longer ACE occupancy. In contrast, the cache-lines get replaced or invalidated by the threads because of more accesses in the same cache for the sharing configuration, which lowers the W-R interval and ACE occupancy duration. Therefore, *swaptions* has a lower

CVF_{mc} for L2-caches in the sharing configuration than the non-sharing configuration.

On the other hand, CVF_{mc} increases in the sharing configuration for rest of the benchmarks. Among these benchmarks, *bodytrack* has a decreased cache miss rate in the sharing configuration. Lower cache miss rate means the threads have a higher hit rate in the cache, which can be caused by the data sharing behavior between the *bodytrack* threads. Because of the sharing behavior, the application threads keep using the same cache blocks. From Figure 5.9(a), we observe that the D-Repl interval is much increased in the sharing configuration, which means *bodytrack* threads have shared write operations. The time interval between the write operation and cache-line replacement contributes to the D-Repl lifetime, consequently to the ACE lifetime. In addition, as the cache miss rate is lower, the cache-line statuses are *not* frequently changed to the W-state. This phenomenon reduces the interval between the current state changes to W-state and consequently increases the ACE lifetime. Therefore, both lower cache miss rates and shared write operations increase *bodytrack*'s ACE lifetime, in turn CVF_{mc}, in the sharing configuration.

Figure 5.9 shows the stacked intervals of ACE lifetime for *fluidanimate* and *freqmine*. In both figures we observe that the W-R and R-R ACE intervals are much reduced when the application threads are mapped in the sharing configuration, especially for *freqmine*. Most of the ACE lifetime consists of the D-Repl interval in both characterization configurations, which means these applications have frequent write operations in the cache-lines that eventually get replaced. Both *fluidanimate* and *freqmine* have increased cache miss rates of 8% and 1%, respectively in the sharing configuration. The increased cache miss rates cause the dirty cache block to be evicted so that the requested cache block can be placed in the cache according to the cache replacement policy. The replaced dirty cache blocks need to be written and updated in the resources lower at the memory hierarchy so that the memory is consistent with the application execution. The latency of the write-back operations contribute to the ACE lifetime because any bit-flips in the cache block before it is written back to the lower-level resources can result in wrong application output when the same cache block is read and



(a) *bodytrack* (b) *fluidanimate*

(c) *freqmine* (d) *facesim*

Figure 5.9: Percentage ACE lifetime intervals during the execution of the applications

used by the application later in the execution. Thus, the long-latency write-back operation causes the CVF_{mc} of L2-cache for these applications to increase in the sharing configuration. Similar explanation is also applicable to *facesim* (lifetime data shown in Figure 5.9(d)). The remaining benchmark, *Raytrace*, has cache miss rate increase from 16% to 18%, which is very small. We can conclude that *raytrace*'s threads in the sharing configuration increases its vulnerability to soft errors in L2-cache by prolonging the ACE residency time of the cache-lines.

5.4.2 Characterization: Discussion and Summary

From the characterizations of the eight PARSEC benchmarks for cache vulnerability, we obtain several insights. Depending on how application threads are placed on the cores to share

Benchmark	Configuration to reduce CVF_{mc}	Reason for the reduced CVF_{mc}
streamcluster (SC)	Sharing	Increased miss-rate
swaptions (SW)	Sharing	Reduced ACE lifetime
x264 (X2)	Sharing	Increased miss-rate
bodytrack (BT)	Non-sharing	Increased miss-rate
facesim (FA)	Non-sharing	Reduced write-back
fluidanimate (FL)	Non-sharing	Reduced write-back
freqmine (FQ)	Non-sharing	Reduced write-back
raytrace (RT)	Non-sharing	Reduced ACE lifetime

Table 5.2: Characterization summary of the PARSEC benchmarks

the same cache or not, the CVF_{mc}s of different applications range from 4% to 78%, and for a particular application, the difference of CVF_{mc} across different characterization configurations ranges from 5% to more than 50%. Therefore, we conclude that thread-mapping technique can be used as an approach to understand the ACE behavior of a multi-threaded application because the technique varies an application's occupancy behavior in the shared caches.

Table 5.2 summarizes the findings of the characterization phase of ReSense_Reliability for the PARSEC applications. It describes the characterization configuration that reduces each application's L2-cache vulnerability and the reason for the reduction. From the characterization, we observed that three PARSEC benchmarks' cache vulnerabilities reduce as the threads share the same L2-cache in the sharing configuration. Among these three applications, *streamcluster* and *x264* have increased cache miss rates in the sharing configuration, which causes most of the cache-line access pattern to result in W-W intervals and decreases the ACE lifetime of the shared cache. *Swaptions*' cache miss rate decreases in the sharing configuration, which reduces both W-R and D-Repl lifetime intervals and causes its vulnerability to decrease.

For rest of the PARSEC benchmarks, the L2-cache vulnerability is reduced when its threads are mapped in the non-sharing configuration. Among these applications, *bodytrack* has decreased L2-cache miss rate in the sharing configuration, which causes its threads to have a higher hit rate in the cache increasing the ACE lifetime. To reduce cache vulnerability of this application, its threads should be mapped in the non-sharing configuration so that the

Benchmark	$\text{SensitivityScore}_{\text{CVF}}$
	L2-cache
bodytrack (BT)	- 34.99
facesim (FA)	- 29.91
fluidanimate (FL)	- 18.89
freqmine (FQ)	- 5.66
raytrace (RT)	- 51.74
streamcluster (SC)	+ 6.03
swaptions (SW)	+ 44.44
x264 (X2)	+ 11.86

Table 5.3: $\text{SensitivityScore}_{\text{CVF}}$ of the PARSEC benchmarks

cache hit rate and ACE lifetime are reduced. Three other applications, *facesim*, *fluidanimate*, and *freqmine*, have increased CVF_{mc} in the sharing configuration because of the long-latency and frequent write-back operations of the dirty cache-lines, which increase the ACE D-Repl interval. To reduce the duration of this ACE lifetime, these applications should be mapped in the non-sharing configuration so that the frequency and duration of the write-back operations are decreased. The CVF_{mc} of *raytrace* is decreased in the non-sharing configuration because of the overall reduction in its ACE lifetime.

From these observations and insights about application characteristics of cache resource usage, we can conclude that thread-mapping can be used to reduce CVF_{mc} of the shared caches. In particular, by choosing co-runner(s) that decrease cache vulnerability by increasing the miss rates in the shared caches and allowing applications to use multiple caches to decrease the duration of frequent write-back operations, the mapping algorithm reduces the applications' vulnerability from a workload to soft errors in shared caches.

We summarize the $\text{SensitivityScore}_{\text{CVF}}$ for the eight PARSEC applications in Table 5.3, which are used as input in the mapping phase. Positive sign represents reduction of CVF_{mc} and negative sign represents increase of CVF_{mc} in the sharing configuration.

5.4.3 Mapping: Experimental Details and Results

To evaluate $\text{ReSense}_{\text{Reliability}}$'s effectiveness in minimizing application's vulnerability to soft error in caches, we use workloads consisting of multi-threaded applications from PARSEC benchmark suite. To design workloads for evaluating the mapping phase, we use the same eight applications with the *simlarge* input set, used in Section 5.4.1. As the evaluation platform, we use the same Simics configuration, described in Table 5.1 used in the characterization phase.

For evaluation, we use several pairs of PARSEC applications as workloads, where both applications start execution simultaneously. We choose three types of pairs: one benchmarks with a positive and one benchmark with a negative $\text{SensitivityScore}_{\text{CVF}}$, two applications with positive $\text{SensitivityScore}_{\text{CVF}}$, and two benchmarks with negative $\text{SensitivityScore}_{\text{CVF}}$. From each category, we select the benchmarks with the execution times such that the simulation are finished within reasonable amount of time.

$\text{ReSense}_{\text{Reliability}}$ can be implemented as a virtual execution manager, similarly as the $\text{ReSense}_{\text{Performance}}$ run-time system. However, as we do experiments using a full-system simulation infrastructure, we execute each pair such that both applications run simultaneously only once in order to ensure the applications run to completion in a reasonable amount of simulation time. For this one iteration, we manually set the core affinity of the applications using the thread-mappings provided by the $\text{ReSensor}_{\text{R}}$ algorithm. This limitation is not because of the run-time system or the algorithm, but for using the Simics simulator.

To evaluate the effectiveness of $\text{ReSense}_{\text{Reliability}}$, we use CVF_{mc} as the evaluation metric. For the general case when the targeted platform has multiple shared caches, we use the following equation, Equation 5.6, to calculate the evaluation metric CVF_{mc-avg}.

$$CVF_{mc-avg} = \frac{\sum_{i=1}^{n} CVF_{mc-SharedCache_i}}{n} \tag{5.6}$$

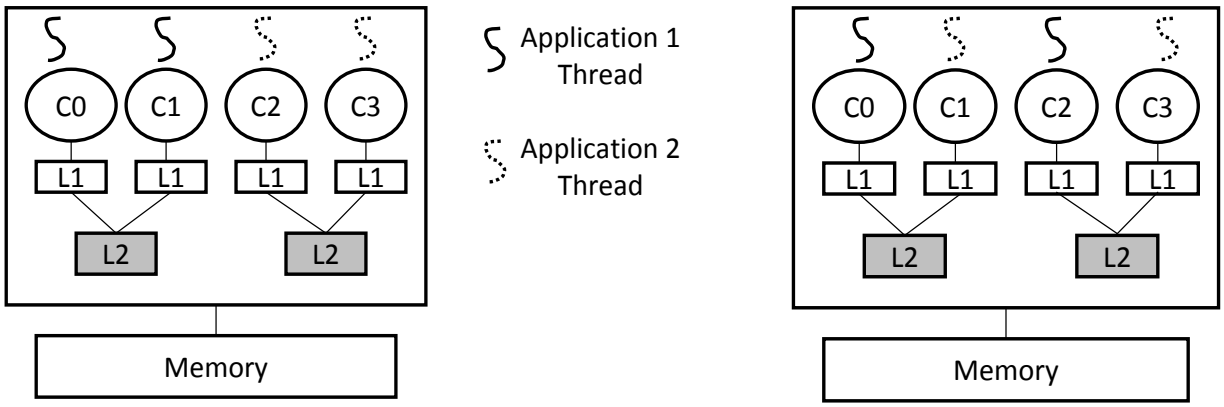
Here, n is the total number of shared caches on the experimental platform and $CVF_{mc-SharedCache_i}$ is the cache vulnerability factor of the i-the shared cache.

Most of the techniques for reducing the application vulnerability are architecture-level approaches. As we use an application-level technique to reduce cache vulnerability, we choose a technique that is also implemented in the application-level for comparison. Therefore, we compare our evaluation results with that of the native operating system (OS), which is used as a baseline. We run each application workload pair described in the previous paragraphs in two configurations. In the baseline configuration, we run the application under the operating system's control where the native OS determines the thread-mappings of the application threads. In the second configuration, we run each application pair under $\text{ReSense}_{\text{Reliability}}$'s control, where the mapping is determined by the $\text{ReSensor}_{\text{R}}$ algorithm using $\text{SensitivityScores}_{\text{CVF}}$. In both configurations, we calculate CVF_{mc-avg} for the shared caches to present the mapping results.

5.4.3.1 Evaluation Results

Figure 5.11 shows the experimental results when pairs of PARSEC applications are mapped using $\text{ReSense}_{\text{Reliability}}$ run-time system in $\text{ReSensor}_{\text{R}}$-mapping. The Y-axis shows CVF_{mc-avg} of the shared L2-caches on the experimental platform, for both baseline (OS-mapping) and the mapping determined by the $\text{ReSensor}_{\text{R}}$ algorithm. The X-axis shows the application pairs used in the experiments. With two applications each with two threads, there are two mapping configurations for the application threads, which are shown in Figure 5.10.

From the figure we observe that for all the application pairs, CVF_{mc-avg} reduces up to 70% over the native OS mapping, when the application threads are controlled by $\text{ReSense}_{\text{Reliability}}$ run-time system and mapped using $\text{ReSensor}_{\text{R}}$. The first pair of *streamcluster* and *swaptions* are the applications that have both positive $\text{SensitivityScore}_{\text{CVF}}$. According to the algorithm, $[VulApps]$ contains *streamcluster* because its CVF_{mc} decreases as its cache miss rate increases (Figure 5.7). Therefore, **Special Case** of the algorithm is chosen by the run-time system, and it maps the threads from different applications to share the same shared cache (Figure 5.10(b)).



(a) Non-sharing Configuration (b) Space-sharing Configuration

Figure 5.10: Configurations for mapping two multi-threaded applications, each with two threads

This mapping configuration allows the application threads to *space-share* the targeted resource. The *SC*-threads increase the cache miss rates in both shared caches, which reduces the ACE lifetime of the cache-lines and consequently CVF_{mc-avg} over the native OS by 31%.

The next six workloads consist of one application that has a positive $\text{SensitivityScore}_{\text{CVF}}$ and one application that has a negative $\text{SensitivityScore}_{\text{CVF}}$. The first pair with *streamcluster* and *bodytrack* application satisfies the condition for **Special Case** of the algorithm. The application threads are mapped in the space-sharing configuration, which reduces the ACE lifetime because of the cache misses from the *SC*-threads. The next three application pairs have *streamcluster* as a co-runner, which satisfies the **Special Case** requirement. Also the other applications in the pairs, *facesim, fluidanimate*, and *freqmine* have higher CVF_{mc} when their sibling threads share the same cache. It means CVF_{mc} is lower for these applications when their sibling threads are spread across the shared caches. These applications also have higher magnitudes of the $\text{SensitivityScore}_{\text{CVF}}$ than that of *streamcluster*, which means the application threads are mapped according to the negative sign of the $\text{SensitivityScore}_{\text{CVF}}$ of these applications. For these reasons, the space-sharing mapping configuration is chosen by the algorithm where the application thread can share the same cache with *SC*-threads. This mapping results in CVF_{mc-avg} reductions of *SC_FA*, *SC_FL*, *SC_FQ* pairs by 12%, 70%,

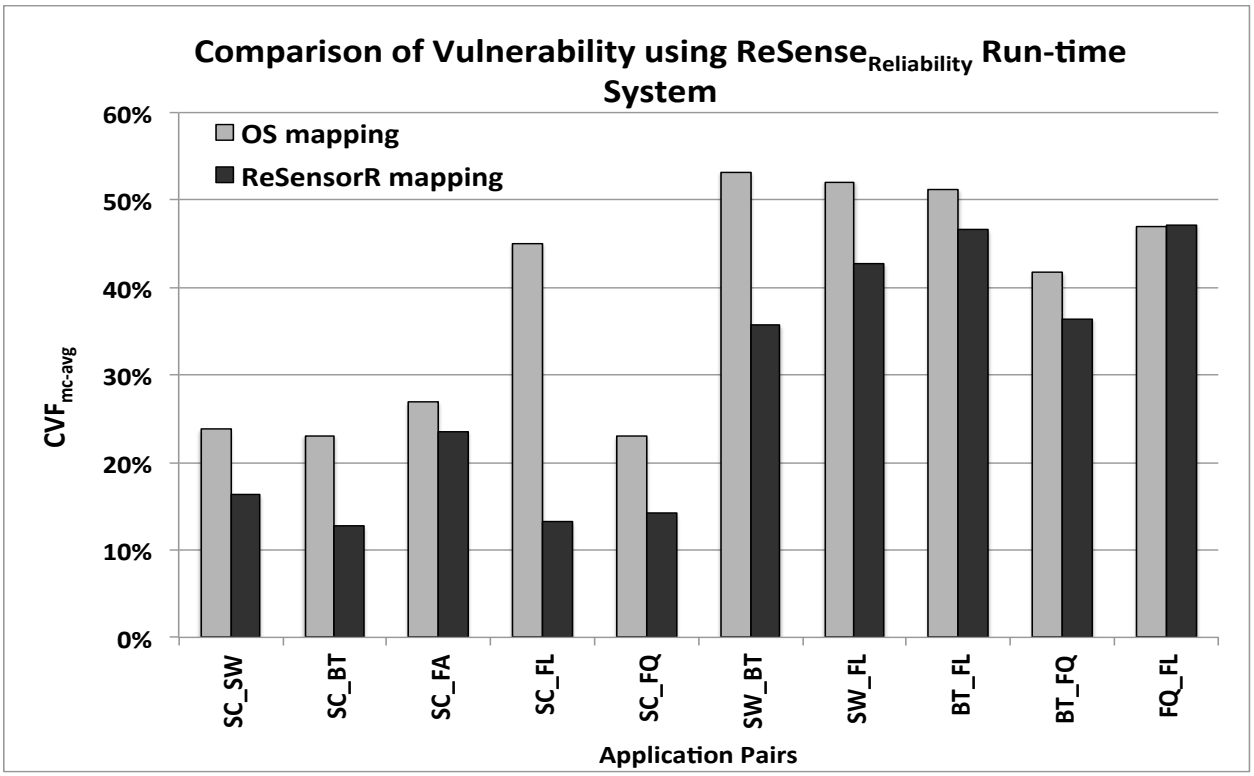


Figure 5.11: Experimental results showing average CVF_{mc} of the shared caches for pairs of PARSEC application for ReSensor$_{\mathrm{R}}$ and OS mapping (Lower bar is better)

and 38% respectively, over the native OS mapping.

For the pairs with *swaptions* as the co-running application, **General Case** of the algorithm is applied as [$VulApps$] array is empty, and the mappings are determined by comparing the magnitudes of the SensitivityScores$_{\mathrm{CVF}}$. For the BT_SW pair, *swaptions*' CVF_{mc} reduces when its threads share the same L2-cache. On the other hand, *bodytrack*'s CVF_{mc} increases when its threads share the same cache. Therefore, in this case, the ReSensor$_{\mathrm{R}}$ algorithm needs to prioritize the applications to determine the mapping that lowers CVF_{mc-avg} compared to the alternate mappings. Comparing the magnitudes of the SensitivityScores$_{\mathrm{CVF}}$, *swaptions* has a higher sensitivity score than *bodytrack*. Therefore, ReSensor$_{\mathrm{R}}$ algorithm prioritizes *swaptions* for determining the mapping that reduces CVF_{mc-avg} and chooses the non-sharing mapping (Figure 5.10(a)). This mapping increases the CVF_{mc} of the cache shared by *bodytrack* by 8%, but decreases the CVF_{mc} of the cache shared by *swaptions* by 67%. The reduction of

CVF_{mc-avg} over the native OS is 33%. This mapping results in a lower CVF_{mc-avg} than the alternate space-sharing mapping, which lowers the CVF_{mc-avg} over the native OS by 27%.

Similar explanation and observation is applicable for *FL_SW* pair. *Fluidanimate* has write operations in the cache-lines that eventually get replaced because of increased cache miss rate in the sharing configuration (Figure 5.3(b)). The latency of the write-back operations contributes to the ACE lifetime of the cache-lines and the CVF_{mc}s of the shared caches increase in the sharing mapping configuration. Therefore, the threads from *fluidanimate* should be mapped on the cores that use two separate caches with the threads from *swaptions* according to the space-sharing mapping (Figure 5.10(b)). However, this mapping increases the CVF_{mc} of the cache shared by *swaptions* by 80%. Therefore, this pair is mapped by comparing the magnitude of the $\text{SensitivityScore}_{\text{CVF}}$ of the applications in the workload. $\text{ReSensor}_{\text{R}}$ algorithm chooses to map these application threads in the non-sharing configuration by prioritizing the more-sensitive application *swaptions* and reduces the CVF_{mc-avg} over the native OS by 18%. Therefore, we can conclude that $\text{ReSensor}_{\text{R}}$ algorithm chooses the mapping that minimizes CVF_{mc-avg} among the possible mapping configurations for the application pairs with *swaptions* as co-runner.

For the remaining pairs, $[VulApps]$ array is empty, and the pairs consist of the applications that both have negative $\text{SensitivityScores}_{\text{CVF}}$. For these pairs, both applications' characteristics of resource occupancy cause the CVF_{mc} of the shared L2-caches to reduce when the threads are spread across the shared caches. For the pairs with *bodytrack*, when the threads from *bodytrack* share the same L2-cache in the non-sharing configuration, the CVF_{mc} of that cache increases because of the increased vulnerable D-Repl lifetime. The other applications in the pair, *fluidanimate* and *freqmine* causes the shared cache's CVF_{mc} to increase by frequently modifying the cache-lines, which increase the write-back latency contributing to the ACE lifetime. For these inherent characteristics of the applications, $\text{ReSensor}_{\text{R}}$ algorithm chooses to map the application threads in the space-sharing configuration to reduce the ACE lifetime of the cache-lines. As a result of this mapping, for *BT_FL* and *BT_FQ* pairs, CVF_{mc-avg}

reduces by 9% and 13% respectively, over the native OS.

For the last pair, $\text{ReSensor}_{\text{R}}$ maps the application threads in the space-sharing configuration because both applications have negative $\text{SensitivityScores}_{\text{CVF}}$. However, this mapping increases CVF_{mc-avg} of the shared caches by 0.19% over the native OS, which is very small. The alternate non-sharing mapping configuration increases the CVF_{mc-avg} by almost 23% over the native OS, which is much higher than 0.19%. Therefore, the space-sharing configuration for *FQ_FL* pair is better for reducing CVF_{mc-avg}.

To summarize, we can conclude from the experimental results that $\text{ReSense}_{\text{Reliability}}$ run-time system effectively uses $\text{ReSensor}_{\text{R}}$ to reduce the CVF_{mc-avg} of the shared caches using the offline vulnerability characteristics of the applications in a workload over the native OS mapping. This reduction in cache vulnerability factors results in a proportional reduction of the overall FIT rate and improvement of the mean-time-to-failure (MTTF) metric for the shared caches.

5.4.4 Mapping: Discussion and Statistical Analyses

We perform an analysis on the experimental results presented in the previous section to determine its statistical significance. We assume the null hypothesis that $\text{ReSense}_{\text{Reliability}}$ run-time system does not reduce the average cache vulnerability, CVF_{mc-avg}, for the workloads over the native OS. As we perform each experiment in two configurations, using the native OS and $\text{ReSense}_{\text{Reliability}}$ run-time, each experiment has two distributions, *OS* and *ReSense*. We perform a paired *t-test* to compare these distributions to determine if *OS* is better than *ReSense* for the CVF_{mc-avg} metric [101]. For the workload with application pairs, the null hypothesis is rejected with a p-value of 0.005. It indicates that the probability of $\text{ReSense}_{\text{Reliability}}$ and $\text{ReSensor}_{\text{R}}$ reducing the workload's average cache vulnerability to soft errors over the native OS is 99.5%, which is very high. In addition, we determine the confidence interval of the average reduction of CVF_{mc-avg} by $\text{ReSense}_{\text{Reliability}}$ and $\text{ReSensor}_{\text{R}}$ over the native OS. This average reduction ranges from 2% to 52%. Because the reduction is

positive, it means $\text{ReSense}_{\text{Reliability}}$ always reduces the application average cache vulnerability using $\text{ReSensor}_{\text{R}}$, and this reduction of cache vulnerability can be as high as 52% over the native OS.

5.5 Summary

In this chapter, we addressed the challenges of minimizing the effect of soft errors in the memory resources on modern multicore architectures using ReSense_Reliability, a reliability instance of the ReSense framework.

The characterization phase of ReSense_Reliability instantiated the general methodology of the ReSense framework to characterize a multi-threaded application based on its resource occupancy duration in the shared caches. It used cache vulnerability factor as the characterization metric and calculated $\text{SensitivityScores}_{\text{CVF}}$ for the applications in a workload. Each $\text{SensitivityScore}_{\text{CVF}}$ represented the vulnerability characteristics of a multi-threaded application.

Using the methodology, we characterized multi-threaded applications from the PARSEC benchmark suite for a write-back, write-allocate shared L2-cache. The characterization revealed several important insights. Different thread-mapping configurations caused the cache vulnerability of different benchmarks to range from 4% up to 78%. Three benchmarks had decreased cache vulnerability and five benchmarks had increased cache vulnerability as the threads shared the same L2-cache. From the characterization results, we had two observations regarding when an application's CVF_{mc} decreased. The first observation is when application threads had a high cache read miss rate, it caused the cache-lines to be frequently updated with the fetched data blocks, which increased non-vulnerable W-W interval of its total lifetime and decreased the CVF_{mc}. The second observation is when the application threads had frequent write operations, it resulted into modified cache-lines. If there was a cache miss and a modified cache-line needed to replaced, the write-back operation of the dirty cache-line contributed to the D-Repl lifetime and increased the ACE lifetime of the

cache block and consequently CVF_{mc}. These two observations and insights about application resource usage were used to reduce CVF_{mc} of the shared caches. In particular, by increasing the miss rate significantly in shared caches by choosing cache-intensive co-runner(s) and allowing applications to use multiple caches to decrease the duration of frequent write-back operations, the mapping algorithm could decrease an application's vulnerability to soft errors in a shared cache.

In the online mapping phase of ReSense_Reliability, the run-time system of the ReSense framework was instantiated as $\text{ReSense}_{\text{Reliability}}$. The $\text{ReSense}_{\text{Reliability}}$ run-time system dynamically managed the mappings of the application threads from a given workload by employing a thread-mapping algorithm, $\text{ReSensor}_{\text{R}}$ and the pre-determined $\text{SensitivityScores}_{\text{CVF}}$. The $\text{ReSensor}_{\text{R}}$ algorithm determined the thread-mappings of the multi-threaded applications in a workload using the $\text{SensitivityScores}_{\text{CVF}}$ of the applications. The algorithm optimized the objective function of this instance, which was to minimize the overall cache vulnerability factor to reduce the effect soft errors in caches on a workload's execution. $\text{ReSense}_{\text{Reliability}}$'s effectiveness was evaluated using ten workloads that consisted of pairs of PARSEC multi-threaded applications. The evaluation revealed that $\text{ReSense}_{\text{Reliability}}$ effectively used applications' $\text{SensitivityScore}_{\text{CVF}}$ and $\text{ReSensor}_{\text{R}}$ mapping algorithm and reduced the overall cache vulnerability by up to 70% over the native operating system. The statistical analyses revealed that the probability of $\text{ReSense}_{\text{Reliability}}$ and $\text{ReSensor}_{\text{R}}$ reducing the workload's average cache vulnerability to soft errors over the native OS was very high, 99.5%, and the reduction of average cache vulnerability could be as high as 52%.

From the analyses results of both characterization and mapping phase, we conclude that the ReSense framework was effectively used to minimize application vulnerability to soft errors in shared cache on a multicore architecture. This reduction in cache vulnerability can result in a proportional reduction of the overall FIT rate and improvement of the MTTF for the shared caches. Utilizing thread-mapping technique to reduce the effect of soft errors demonstrated a novel approach and addressed the challenges of improving application reliability using an

application-level technique.

Chapter 6

Using ReSense for Performance and Reliability Integration

In this chapter, we present a preliminary exploration of the usage of the ReSense framework to address two targeted problems for shared caches: resource contention and soft errors. The chapter describes how ReSense can be used to develop an integrated instance, ReSense_Integration, which addresses these two targeted problems and determines a trade-off between application performance and reliability improvements on modern multi-core platforms.

6.1 Introduction

The thread-mapping configurations determined by the $\text{ReSensor}_{\text{P}}$ algorithm (presented in Chapter 4) that improves an application's performance may not improve its reliability. A cache-intensive multi-threaded application contends for the shared cache when its threads share the same cache. Because the threads contend for the cache, they replace the contents of the cache-lines very frequently and reduce the cache-lines' vulnerable occupancy duration. This behavior reduces the application's susceptibility to soft errors in the shared caches. However, this mapping degrades the application's performance because it causes more cache misses and

creates contention in the cache. To mitigate cache contention, the application's threads are mapped to use separate caches, which leads to application performance improvement. However, this mapping can increase the vulnerable occupancy duration of the cache-lines because the application threads less frequently replace their contents. This increased occupancy duration of the cache-lines can increase the shared cache's vulnerability to soft errors. Similarly, the thread-mapping configurations determined by the $\text{ReSensor}_{\text{R}}$ algorithm (presented in Chapter 5) that improves an application's reliability may not improve its performance. Therefore, there is a trade-off between application performance and reliability improvements.

The thread-mapping that determines a trade-off between the performance and reliability improvements, should utilize an application's sensitivity to the targeted resources. This sensitivity can vary with respect to its contention and vulnerability characterizations for a targeted resource, which is determined by its contentiousness and resource occupancy behaviors. The trade-off between performance and reliability improvements should be determined based on the preference between the contention and vulnerability characterizations of the applications in a workload. If the contentious behavior of the multi-threaded applications in a workload are preferred more than its occupancy and vulnerability to soft errors in the targeted resource, the trade-off thread-mapping should more target mitigating resource contention. If the applications' occupancy and vulnerability behaviors to soft errors are preferred more than the contentiousness in the targeted resource, the trade-off thread-mapping should more target reducing the application vulnerability to soft errors.

To determine an application's behaviors for contention and vulnerability to soft errors in a targeted resource, a characterization technique can be used. This characterization technique should determine how a multi-threaded application's performance and vulnerability are affected by its contentiousness and occupancy duration in the targeted resource, respectively. Once these application characteristics are identified in isolation, these characterizations can be integrated to determine the application's combined behaviors for both resource contention and soft errors. These application behaviors help determine the thread-mappings for a

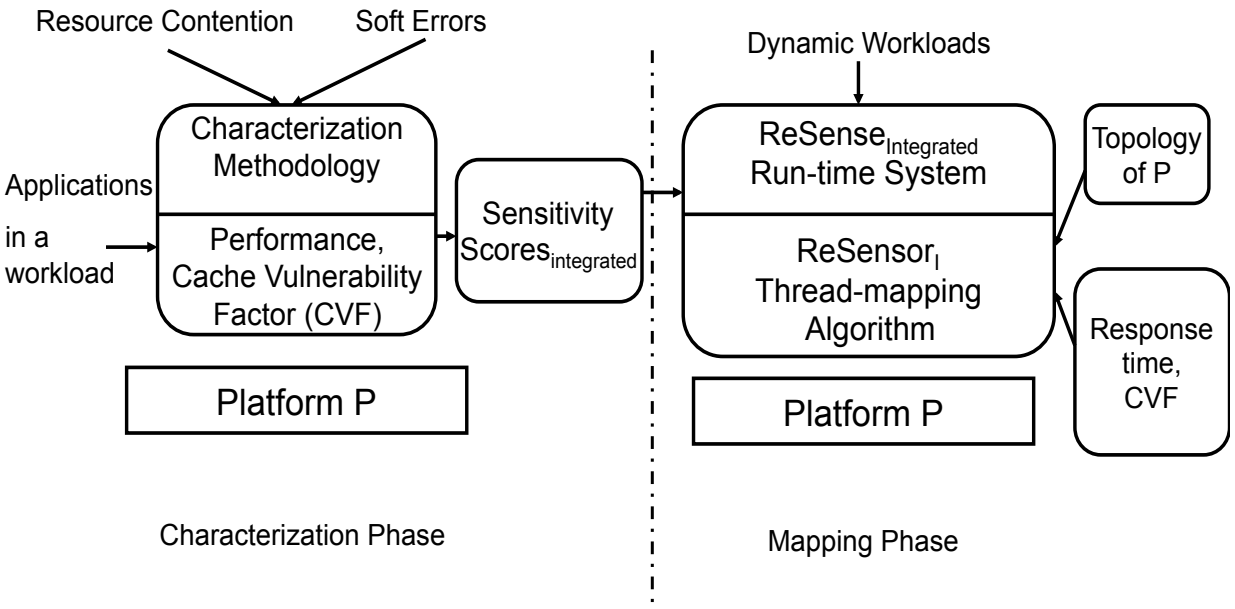


Figure 6.1: Components of the ReSense_Integration Instance

trade-off between performance and reliability improvements.

This approach leads to creating an *integrated instance* of the ReSense framework, ReSense_Integration, for achieving a trade-off between performance and reliability improvements. Figure 6.1 shows the components of ReSense_Integration. The characterization phase of the instance applies the characterization methodology of the framework to characterize each multi-threaded application in a workload based on its contentiousness and occupancy duration in a shared cache. It uses application performance and cache vulnerability factor as the characterization metrics and calculates $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ for each application. These sensitivity scores are combined to calculate $\text{SensitivityScore}_{\text{integrated}}$, which represents the combined contention and vulnerability characteristics of each multi-threaded application in a workload.

In the mapping phase of the instance, run-time system of the ReSense framework is instantiated as $\text{ReSense}_{\text{Integrated}}$, and thread-mapping algorithm is instantiated as $\text{ReSensor}_{\text{I}}$. The $\text{ReSense}_{\text{Integrated}}$ run-time system employs the $\text{ReSensor}_{\text{I}}$ algorithm and $\text{SensitivityScores}_{\text{integrated}}$ to map application threads from a dynamic workload. $\text{ReSensor}_{\text{I}}$ dynamically determines the thread-mappings of the applications in a workload using the pre-determined integrated

characteristics of the applications on the targeted platform. The algorithm optimizes the objective function of this instance, which is to determine a trade-off between response time and cache vulnerability factor reductions.

The outline of this chapter is as follows: Section 6.2 describes the characterization phase, which includes the integration of a multi-threaded application's contentious and resource occupancy characteristics for shared caches using the general characterization methodology of ReSense. Section 6.3 describes the mapping phase, which includes the $\text{ReSensor}_{\text{I}}$ thread-mapping algorithm and the $\text{ReSense}_{\text{Integrated}}$ run-time system. Section 6.4.1 presents the integrated characteristics of the PARSEC benchmarks for shared caches. Section 6.4.2 describes the experimental methodology and evaluation metrics of the mapping phase and presents the evaluation results of the workloads using the $\text{ReSense}_{\text{Integrated}}$ run-time system. Section 6.5 concludes the chapter.

6.2 Characterization for Resource Contention and Vulnerability

For this instance, to combine application characteristics for contention and vulnerability, the applications should be characterized for the same targeted resource. Because we determine an application's cache vulnerability for the ReSense_Reliability instance, we consider shared caches as the targeted resource.

6.2.1 Characterization Metric

To characterize a multi-threaded application based on its contentiousness and resource occupancy behavior for the shared caches, we use performance and cache vulnerability factor, CVF_{mc}, as the characterization metrics. We characterize each multi-threaded application in a workload based on how contention for the shared cache affects its performance and resource occupancy in the shared caches affects its CVF_{mc}.

6.2.2 Characterization Methodology

According to the characterization methodology of ReSense, application threads are placed in two characterization configurations. In the non-sharing configuration, the application threads are placed on the cores that use two separate shared caches. In the sharing configuration, the application threads are placed to use the same shared cache. Both configurations use the same number of application threads, which equals the number of cores that use the same shared cache. Both configurations use the same number of private caches so that the effect of private cache usage is not included in the characterization. In both characterization configurations, we measure each application's performance and CVF_{mc} and calculate $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ according to Equation 3.1.

6.2.3 $\text{SensitivityScore}_{\text{integrated}}$: Sensitivity Scores for Performance and Reliability

To determine a trade-off between performance and reliability improvements, we combine two characterizations, $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$, to determine the integrated sensitivity scores of the applications using an addition function and Equation 6.1.

$$\text{SensitivityScore}_{\text{integrated}} =$$

$$w_P * SensitivityScore_{\text{performance}} + w_R * SensitivityScore_{\text{CVF}} \tag{6.1}$$

Here, $\text{SensitivityScore}_{\text{performance}}$ represents a multi-threaded application's sensitivity to contention for shared cache, and $\text{SensitivityScore}_{\text{CVF}}$ represents a multi-threaded application's sensitivity to its vulnerability to soft errors in a shared cache. We choose an addition function as a simple way of combining the sensitivity scores so that the trade-off between performance and reliability improvements can be easily manipulated. Other functions can be used, which are left for more exploration as future work.

As both $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ represent an application's characterization for a shared cache, we can combine them. W_P and w_R are the weighting factors, whose summation ranges from 0 to 1. W_P is the performance weighting factor, and w_R is the reliability weighting factor. These weighting factors are used as knobs to control the application behaviors that are prioritized for the integrated instance. If an application's contentious behavior is preferred more to its vulnerability, w_P is set higher than w_R, and vice versa. In general, for ReSense_Performance, $w_P = 1.0$, $w_R = 0$, and for ReSense_Reliability, $w_P = 0$, $w_R = 1.0$. For the integrated instance, $0 <= w_P + w_R <= 1.0$. Thus, the objective of the integrated instance is configurable using these weighting factors as they are combined with the simple addition function.

Similar to $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$, $\text{SensitivityScore}_{\text{integrated}}$ has both a sign and magnitude, which are used by the $\text{ReSensor}_{\text{I}}$ thread-mapping algorithm to determine a trade-off between performance and reliability improvements. The sign and magnitude of $\text{SensitivityScore}_{\text{integrated}}$ depend on the signs and magnitudes of both $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$. If both $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ have positive signs, then $\text{SensitivityScore}_{\text{integrated}}$ has a positive sign. The positive sign means the application has both improved performance and reduced cache vulnerability to soft errors when its threads share the same shared cache. If both $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ have negative signs, then $\text{SensitivityScore}_{\text{integrated}}$ has a negative sign. The negative sign means the application has both degraded performance because of cache contention and increased cache vulnerability to soft errors when its threads share the same shared cache.

If $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ have different signs, then their magnitudes and the weighting factors determine the sign of $\text{SensitivityScore}_{\text{integrated}}$. The weighted sensitivity score with the higher magnitude determines the sign of $\text{SensitivityScore}_{\text{integrated}}$. Assume, $\text{SensitivityScore}_{\text{performance}}$ is positive, $\text{SensitivityScore}_{\text{CVF}}$ is negative, and $w_P > w_R$. If $| w_P * \text{SensitivityScore}_{\text{performance}} | > | w_R * \text{SensitivityScore}_{\text{CVF}} |$, then $\text{SensitivityScore}_{\text{integrated}}$

has a positive sign. In this case, the positive sign represents that the application's sharing behavior is preferred more than its vulnerability for shared caches for the chosen weights. The positive sign means that sharing the same cache improves the application performance; however, the cache vulnerability increases because of its $\text{SensitivityScore}_{\text{CVF}}$'s negative sign. Here, performance is prioritized over reliability if these threads are mapped to use the same cache.

Assuming the same magnitudes of the sensitivity scores, if the weighting factors are chosen such that $\mid w_P * \text{SensitivityScore}_{\text{performance}} \mid < \mid w_R * \text{SensitivityScore}_{\text{CVF}} \mid$, then $\text{SensitivityScore}_{\text{integrated}}$ has a negative sign. The negative sign means that the application's vulnerability behavior is preferred more to its sharing behavior in shared caches. Reliability is prioritized over performance if these threads are mapped to use separate shared caches. This mapping may degrade application performance because its data sharing behavior is not considered. These two examples show the trade-off between performance and reliability improvements.

Other cases are also possible when applications with different signs and magnitudes of $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ are considered and different weighting factors are chosen. Because $\text{SensitivityScore}_{\text{integrated}}$ is calculated using an application's both contentious and vulnerability characteristics, it helps determine a trade-off between performance and reliability improvements for a given workload.

If the targeted platform has multiple levels of shared caches, we characterize the applications and determine its $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ for the shared caches at each level. These sensitivity scores are used to calculate $\text{SensitivityScore}_{\text{integrated}}$ for each shared cache, which is saved in the $SV_{integrated}$ vector. If the targeted platform has two levels of shared caches, then this $SV_{integrated}$ vector is a two-element vector for each multi-threaded application. This vector is passed as an input parameter to the $\text{ReSense}_{\text{Integrated}}$ thread-mapping algorithm.

6.3 Mapping Co-located Multi-threaded Applications for Performance and Reliability Trade-off

We describe the components of the mapping phase for the integrated instance, including the $\text{ReSensor}_{\text{I}}$ algorithm and the $\text{ReSense}_{\text{Integrated}}$ run-time system, in the following sections.

6.3.1 The $\text{ReSensor}_{\text{I}}$ Thread-mapping Algorithm

Depending on the characteristics of the applications in a workload, one mapping can improve application performance at the cost of less reliable execution, and a different mapping may ensure application reliability at the cost of application performance degradation. The goal of the integrated $\text{ReSensor}_{\text{I}}$ thread-mapping algorithm is to dynamically map the application threads from the input workload such that the mapping prioritizes the preferred characteristics of the applications, i.e., contentiousness or vulnerability, as chosen by the weighting factors.

If the multi-threaded applications' resource contention characteristics are preferred, the $\text{ReSensor}_{\text{I}}$ thread-mapping algorithm maps the application threads to mitigate contention by prioritizing performance over reliability. Similarly, if the applications' vulnerability characteristics for shared caches are preferred by choosing the appropriate weighting factors, the thread-mapping algorithm dynamically adjusts the mapping to reduce cache occupancy by prioritizing reliability over performance. Because an application's $\text{SensitivityScore}_{\text{integrated}}$ is determined using both its contentiousness and resource occupancy characteristics, it is used by the algorithm to determine the thread-mappings that prioritize the preferred characteristics of the applications in a workload.

Algorithm 5 contains the pseudocode of the $\text{ReSensor}_{\text{I}}$ algorithm. The $\text{ReSensor}_{\text{I}}$ algorithm is instantiated from the $\text{ReSensor}_{\text{Generic}}$ thread-mapping algorithm of the framework, where applications' $\text{SensitivityScores}_{\text{integrated}}$ are used as the sensitivity scores. The algorithm maps threads from a workload WL consisting of any number of multi-threaded applications on a particular platform P. Platform P can have shared resources, i.e., caches, in multiple

Algorithm 5 The ReSensor$_{\mathrm{I}}$ Algorithm: Mapping application threads to determine a trade-off between performance and reliability improvements

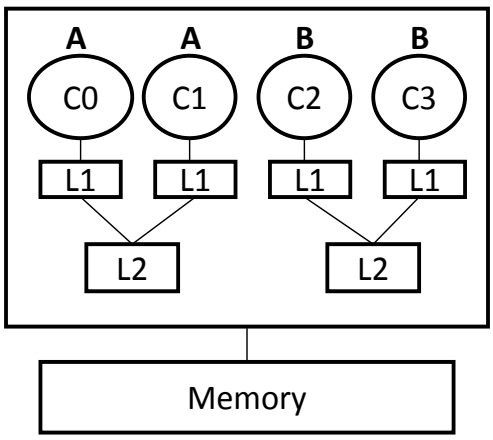
```
1: INPUT: Workload WL, Topology of the experimental platform P, Sensitivity vectors SV_integrated of the applications in WL on P
2: [Apps] ← multi-threaded applications in WL
3: nApps ← number of multi-threaded applications in WL
4: for each level MHL in the memory hierarchy of P do
5:    R ← shared cache at MHL
6:    NR ← number of R at MHL
7:    [C+] ← set of cores that use or share the same R on P
8:    [C−] ← set of cores that do not use or share the same R on P
9:    [SS_I] ← SV_integrated[R] of the applications in [Apps]
10:   sort [SS_I] array in descending order of the magnitude of the SensitivityScore_integrated and re-arrange [Apps] accordingly
11:   if NR >= nApps then
12:      /* Scenario 1: equal or more shared resources than the number of applications */
13:      for ( i = 0 ; i < nApps ; i++ ) do
14:         if SS_I[i] > 0 AND [C+] has available core(s) then
15:            map Apps[i]-threads on the available cores from [C+]
16:         else if SS_I[i] < 0 AND [C−] has available core(s) then
17:            map Apps[i]-threads on the available cores from [C−]
18:         else
19:            /* [C+] or [C−] does not have available core(s) */
20:            map Apps[i]-threads on any core on P
21:         end if
22:      end for
23:   else
24:      /* Scenario 2: fewer shared resources than the number of applications */
25:      for ( i = 0 ; i < nApps / 2 ; i++ ) do
26:         if SS_I[i] > 0 AND [C+] has available core(s) then
27:            map Apps[i]- and Apps[nApps − i − 1]-threads on the available cores from [C+]
28:         else if SS_I[i] < 0 AND [C−] has available core(s) then
29:            map Apps[i]- and Apps[nApps − i − 1]-threads on the available cores from [C−]
30:         else
31:            /* [C+] or [C−] does not have available core(s) */
32:            map Apps[i]- and Apps[nApps − i − 1]-threads on any core on P
33:         end if
34:      end for
35:   end if
36: end for
```

levels of the memory hierarchy. $\text{ReSensor}_{\text{I}}$ considers the shared caches from the bottom to the top of the memory hierarchy, i.e., from L3-caches to L2-caches, to determine the thread-mappings (line 4). The algorithm counts the number of shared caches R at a particular memory hierarchy level MHL (line 6) and determines the $[C_+]$ and $[C_-]$ arrays (line 7, 8). These arrays are used to identify the cores that share the same or different resource R and keep track of the availability of free cores, on which application threads are not mapped yet.

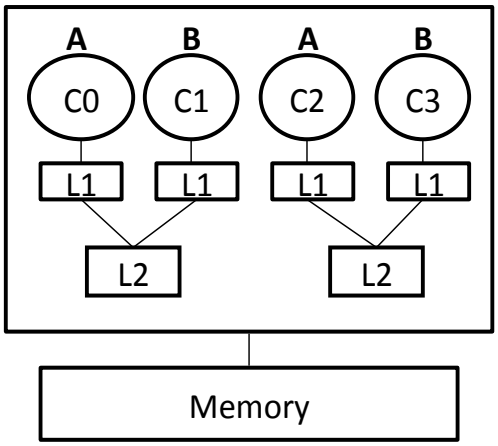
Each application's $\text{SensitivityScore}_{\text{integrated}}$ for resource R is saved into the $[SS_I]$ array (line 9). This $[SS_I]$ array is sorted in the descending order of the magnitude of $\text{SensitivityScore}_{\text{integrated}}$ metric so that the algorithm can prioritize the applications according to their integrated sensitivity to the resource R and re-arranges $[Apps]$ accordingly (line 10).

Depending on the number of applications in the workload, $nApps$ and NR, the algorithm handles two scenarios as the generic $\text{ReSensor}_{\text{Generic}}$ algorithm (with no problem-specific special cases).

Scenario 1: There are the same or more targeted resources than the number of applications, $nApps$, in the workload at a particular time on platform P. The algorithm maps the application threads the same way as $\text{ReSensor}_{\text{Generic}}$, according to the sign and magnitude of the $\text{SensitivityScore}_{\text{integrated}}$ after sorting the $[SS_I]$ array (line 13 - 22). If the sign of $\text{SensitivityScore}_{\text{integrated}}$ of i-th application is positive, then the threads are mapped on the cores from $[C_+]$ (line 15). Depending on the performance and vulnerability sensitivity scores of the application and the weighting factors, this configuration maps application threads prioritizing its preferred characteristics, targeting either more improved performance or reliability. For example, if $\text{SensitivityScore}_{\text{performance}}$ is positive and $\text{SensitivityScore}_{\text{CVF}}$ is negative for the i-th application and the weighting factors are chosen such that $\mid w_P * \text{SensitivityScore}_{\text{performance}} \mid > \mid w_R * \text{SensitivityScore}_{\text{CVF}} \mid$, then this application's resource contention behavior is preferred, and the threads are mapped on the $[C_+]$-cores to prioritize its sharing behavior and reduce application response time. This mapping may increase its cache vulnerability because of its negative $\text{SensitivityScore}_{\text{CVF}}$.



(a) Scenario 1: Example 1



(b) Scenario 1: Example 2

Figure 6.2: Mapping decisions for the two examples in Scenario 1

On the other hand, if the sign of the $\text{SensitivityScore}_{\text{integrated}}$ metric is negative, the threads are mapped on the $[C\_]$-cores that do not share the targeted resource R (line 17). If $\text{SensitivityScore}_{\text{performance}}$ is positive and $\text{SensitivityScore}_{\text{CVF}}$ is negative for i-th application and the weighting factors are chosen such that $|w_P$ * $\text{SensitivityScore}_{\text{performance}}| < |w_R$ * $\text{SensitivityScore}_{\text{CVF}}|$, then this application's shared cache vulnerability to soft errors is preferred, and the threads are mapped on the $[C\_]$-cores to reduce the overall cache vulnerability. However, this mapping can degrade application performance because its $\text{SensitivityScore}_{\text{performance}}$ is positive and the application threads are mapped to use separate caches. If there is no core available for the remaining applications, then the threads are mapped on any cores (line 20). As these applications are the least-sensitive ones (because they are at the end of the sorted array), they do not impact the performance or reliability trade-offs significantly.

Let us consider a workload that has two multi-threaded applications, A and B each with two threads, and these applications are to be mapped on the quad-core platform (e.g., Intel-Yorkfield), shown in Figure 6.2. Here, $nApps = NR = 2$, $[Apps] = [A, B]$, $[C_+] = [\{C0,C1\}, \{C2,C3\}]$, and $[C\_] = [\{C0,C2\}, \{C1,C3\}]$. The shared L2-cache is the targeted resource. Let us assume, A's $\text{SensitivityScore}_{\text{performance}}$ is $+a_P$ and $\text{SensitivityScore}_{\text{CVF}}$ is

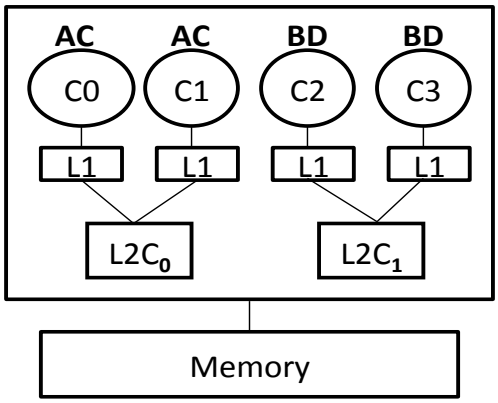
$-a_R$. B's $\text{SensitivityScore}_{\text{performance}}$ is $-b_P$ and $\text{SensitivityScore}_{\text{CVF}}$ is $+b_R$.

For ReSense_Performance, if $|a_P| > |b_P|$, the applications are mapped by $\text{ReSensor}_{\text{P}}$ such that A's threads can share the same cache, as shown in Figure 6.2(a). This mapping improves A's performance more than B's performance degradation, resulting into overall response time improvement for the workload. From the reliability perspective, this mapping reduces CVF_{mc} of the cache shared by B (positive b_R), but increases the CVF_{mc} of the cache used by A (negative a_R). Therefore, better performance is achieved at the cost of reduced reliability.

For ReSense_Reliability, if $|a_R| > |b_R|$, A-threads are mapped to share the same cache with B-threads according to the $\text{ReSensor}_{\text{R}}$ algorithm to minimize both caches' overall CVF_{mc}, as shown in Figure 6.2(b). This mapping may degrade the more sensitive application A's performance because its threads are *not* mapped considering its sharing behavior in cache and result in overall performance degradation, i.e., increased response time. Therefore, better reliability is achieved at the cost of degraded performance.

For ReSense_Integration, let us assume the weighting factors are chosen such that A's and B's contentious characteristics are preferred and $| w_P * a_P| > |w_R * a_R|$, $|w_P * b_P| > |w_R * b_R|$. Considering the integrated characterization, because $|w_P * a_P| > |w_R * a_R|$, A has a positive $\text{SensitivityScore}_{\text{integrated}}$, $+a_I$. And because $|w_P * b_P| > | w_R * b_R|$, B has a negative $\text{SensitivityScore}_{\text{integrated}}$, $-b_I$. Depending on the magnitudes of a_I and b_I, one of the two mapping configurations (shown in Figure 6.2) is chosen by the $\text{ReSensor}_{\text{I}}$ algorithm. If the mapping in Figure 6.2(a) is chosen, the algorithm prioritizes the sharing behavior of A more and improves response time at the cost of reduced reliability. If the mapping in Figure 6.2(b) is chosen, the algorithm prioritizes B's cache contentiousness more and improves response time at the cost of its increased vulnerability. Both mappings improve the response time of the workload and increase cache vulnerability, which demonstrates the trade-off between performance and reliability effectively determined by $\text{ReSensor}_{\text{I}}$.

Scenario 2: For this scenario, the workload has more applications than the number of



(a) Scenario 2: Example 1

Figure 6.3: Mapping decision for an example in Scenario 2

shared caches on P. The algorithm maps multiple applications prioritizing the more sensitive application's behavior (line 25 - 34), which is the same way as $\text{ReSensor}_{\text{Generic}}$ maps the application threads. It maps the most-sensitive (highest magnitude) application from the first half of [*Apps*] with the least-sensitive ones (lowest magnitude) from the second half of [*Apps*], prioritizing the characteristics of the most-sensitive application. If the $\text{SensitivityScore}_{\text{integrated}}$ of the most-sensitive application is positive, it maps its threads and least-sensitive application threads to the available cores from $[C_+]$ (line 27). This mapping prioritizes the characteristics of the most-sensitive application for R. If the $\text{SensitivityScore}_{\text{integrated}}$ of the most-sensitive application is negative, the algorithm maps its threads and the least-sensitive application threads to the available cores from $[C_-]$ (line 29). If there are no available cores from $[C_+]$ or $[C_-]$, the algorithm maps threads on any core on P (line 32). The algorithm terminates when there are no applications left in the workload whose thread-mappings are not determined.

For example, consider a 4-applications workload on the same quad-core platform. The four applications are A, B, C, and D with $\text{SensitivityScore}_{\text{integrated}}$ of $+a_I$, $-b_I$, $+c_I$, $-d_I$ respectively, and assume after sorting $[SS_I] = [+a_I, -b_I, -d_I, +c_I]$, where $|a_I| > |b_I| > |d_I| > |c_I|$. The algorithm chooses the most sensitive application A and the least-sensitive application C, and maps them to share the same cache on the $[C_+]$-cores as the sign of a_I is positive. If A's $\text{SensitivityScore}_{\text{performance}}$ is $+a_P$ and $\text{SensitivityScore}_{\text{CVF}}$ is $-a_R$, this mapping improves

A's performance when $|w_P * a_P| > |w_R * a_R|$. Assuming, C's $\text{SensitivityScore}_{\text{performance}}$ of $-c_P$ and $\text{SensitivityScore}_{\text{CVF}}$ of $+c_R$ where $|w_P * c_P| < |w_R * c_R|$, this mapping reduces $L2C_0$'s ACE lifetime when $|a_R| < |c_R|$ and degrades C's performance. The algorithm maps the rest of the application threads on the remaining cores. The final mapping is shown in Figure 6.3(a). If B has $\text{SensitivityScore}_{\text{performance}}$ of $+b_P$ and $\text{SensitivityScore}_{\text{CVF}}$ of $-b_R$, and D has $\text{SensitivityScore}_{\text{performance}}$ of $-d_P$ and $\text{SensitivityScore}_{\text{CVF}}$ of $+d_R$, then this mapping improves B's performance and degrades D's performance as realized by the signs of their $\text{SensitivityScores}_{\text{performance}}$. However, this mapping can increase $L2C_1$'s vulnerability when $|b_R| > |\ d_R|$ because this mapping cause B's ACE resource occupancy to increase.

For the overall workload, this mapping configuration prioritizes the performance characteristics of application A as its $\text{SensitivityScore}_{\text{integrated}}$ has the highest magnitude and A's sharing behavior is preferred by choosing the appropriate values of the weighting factors. Thus, this mapping can reduce the workload's response time and may increase the overall vulnerability of the caches. On the hand, if applications' vulnerable characteristics and its ACE resource occupancy is preferred by adjusting the weighting factors, then a different mapping would reduce the overall vulnerability of the shared caches. Thus, depending on the applications' characteristics for its cache contention and vulnerability and the chosen weighting factors, the $\text{ReSensor}_{\text{I}}$ determines a trade-off between the workload's response time and vulnerability reductions.

6.3.2 The $\text{ReSense}_{\text{Integrated}}$ Run-time System

The $\text{ReSense}_{\text{Integrated}}$ run-time system is instantiated from the ReSense framework. It detects any change in the total number of threads in a workload and uses the $\text{ReSensor}_{\text{I}}$ thread-mapping algorithm and $\text{SensitivityScore}_{\text{integrated}}$ to map the application threads on the targeted platform.

Benchmark	$\text{SensitivityScore}_{\text{performance}}$	$\text{SensitivityScore}_{\text{CVF}}$
bodytrack (BT)	+ 0.48	- 34.99
facesim (FA)	- 0.29	- 29.91
fluidanimate (FL)	- 0.27	- 18.89
freqmine (FQ)	+ 0.06	- 5.66
raytrace (RT)	- 0.07	- 51.74
streamcluster (SC)	- 0.52	+ 6.03
swaptions (SW)	0	+ 44.44
x264 (X2)	- 0.05	+ 11.86

Table 6.1: $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ of the PARSEC benchmarks on Simics for a shared L2-cache

6.4 Evaluation of the Integrated Instance

To evaluate $\text{ReSense}_{\text{Integrated}}$'s effectiveness in determining a trade-off between performance and reliability improvements, we use multi-threaded applications from the PARSEC benchmark suite. For this evaluation, we use the same eight applications used in Section 5.4.1 with the *simlarge* input set. For this integrated instance, because we consider application characteristics for vulnerability, we use the same Simics configuration described in Table 5.1 as the evaluation platform to emulate the Intel-Yorkfield platform.

6.4.1 Characterization: Integrated Characteristics of the PARSEC benchmarks

We characterized the PARSEC benchmarks for resource contention using real hardware for ReSense_Performance. However, for the integrated instance, we need to characterize these applications using the same platform as ReSense_Reliability so that we can combine their characterizations. Therefore, we characterize the eight PARSEC applications using the Simics simulation infrastructure. The $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ of these eights applications are summarized in Table 6.1.

From the table, we observe that the performances of *bodytrack* and *freqmine* improve in the sharing configuration. The cache miss rate of *bodytrack* reduces from 6% to 4%, and

Benchmark	Sensitivity $\text{Score}_{\text{integrated}}$	Sensitivity $\text{Score}_{\text{integrated}}$	Sensitivity $\text{Score}_{\text{integrated}}$
Weighting Factors	$w_P = 0.99$ $w_R = 0.01$	$w_P = 0.30$ $w_R = 0.70$	$w_P = 0.20$ $w_R = 0.80$
bodytrack (BT)	+ 0.12	- 24.35	- 27.89
facesim (FA)	- 0.58	- 21.02	- 23.98
fluidanimate (FL)	- 0.45	- 13.13	-15.16
freqmine (FQ)	+ 0.003	- 4.17	- 4.78
raytrace (RT)	- 0.59	- 36.24	- 41.40
streamcluster (SC)	- 0.45	+ 4.06	+ 4.72
swaptions (SW)	+ 0.44	+ 31.11	+35.55
x264 (X2)	+ 0.07	+ 8.22	+ 9.48

Table 6.2: $\text{SensitivityScore}_{\text{integrated}}$ of the PARSEC benchmarks for L2-cache using different weighting factors

the application has shared write operations, which causes the application performance to improve when the threads share the same cache. Although the cache miss rate of *freqmine* increases by 1% in the sharing configuration, its performance slightly improves because of the reduced invalidation bus transactions. On the other hand, the performances of *facesim*, *fluidanimate*, *raytrace*, *streamcluster*, and *x264* degrade when the threads are mapped in the sharing configuration. This performance degradation is caused by the increased cache miss rate when the application threads share the same L2-cache. Lastly, the performance of *swaptions* does not change from the non-sharing to the sharing configuration because it is a computation-intensive application, and is not affected by cache contention. Therefore, it has a $\text{SensitivityScore}_{\text{performance}}$ of 0.

Comparing the magnitudes of $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ of the eight applications in Table 6.1, we observe that these PARSEC applications have much higher sensitivity towards their cache vulnerability characteristics than that of cache contention on the targeted platform. The low magnitudes of $\text{SensitivityScore}_{\text{performance}}$ can be caused by the *simlarge* input set. This input set is smaller in size than the *native* input, thus, having smaller impact on the contentious characteristics of the applications.

Table 6.2 shows the calculated $\text{SensitivityScore}_{\text{integrated}}$ of the PARSEC benchmarks using

two sets of weighting factors. The first set, which uses $w_P = 0.99$ and $w_R = 0.01$, is biased towards performance improvement by preferring an application's contention characterization. The second set, which uses $w_P = 0.30$ and $w_R = 0.70$, is biased towards reliability improvement by preferring an application's vulnerability characterization. The third set, which uses $w_P = 0.20$ and $w_R = 0.80$, is also biased towards reliability improvement.

6.4.2 Mapping: Experimental Results and Analyses

For the evaluation of $\text{ReSense}_{\text{Integrated}}$, we use the same ten pairs of PARSEC applications that were used in ReSense_Reliability. We use an additional four-application workload consisting of *SC, SW, BT* and *FL*. As the evaluation metrics, we use the total response time and overall cache vulnerability CVF_{mc-avg}, defined using Equation 6.2 and 5.6. Here, n is the total number of applications in a workload and *Execution Time* is the average wall-clock execution time of an application.

$$Total\ Response\ Time = \sum_{i=1}^{n} ExecutionTime_i \tag{6.2}$$

Each workload is mapped by the $\text{ReSense}_{\text{Integrated}}$ run-time system that uses the $\text{ReSensor}_{\text{I}}$ algorithm. Because each pair-wise workload has two applications each with two threads, there are two mapping configurations shown in Figure 5.10. Depending on how the weighting factors are chosen and the $\text{SensitivityScore}_{\text{integrated}}$ of the applications, the $\text{ReSensor}_{\text{I}}$ algorithm maps the application threads in one of these configurations, which coincides with either the $\text{ReSensor}_{\text{P}}$-mapping or $\text{ReSensor}_{\text{R}}$-mapping.

Figure 6.4 shows the performance results of the PARSEC pairs when the application threads are mapped using the $\text{ReSense}_{\text{Integrated}}$ run-time system and the $\text{ReSensor}_{\text{I}}$ algorithm for the first set of weighting factors ($w_P = 0.99$ and $w_R = 0.01$). These weighting factors are chosen to be biased towards application performance improvements (lower response time). The graph shows the total response time for the baseline OS and the $\text{ReSensor}_{\text{P}}$ mapping for

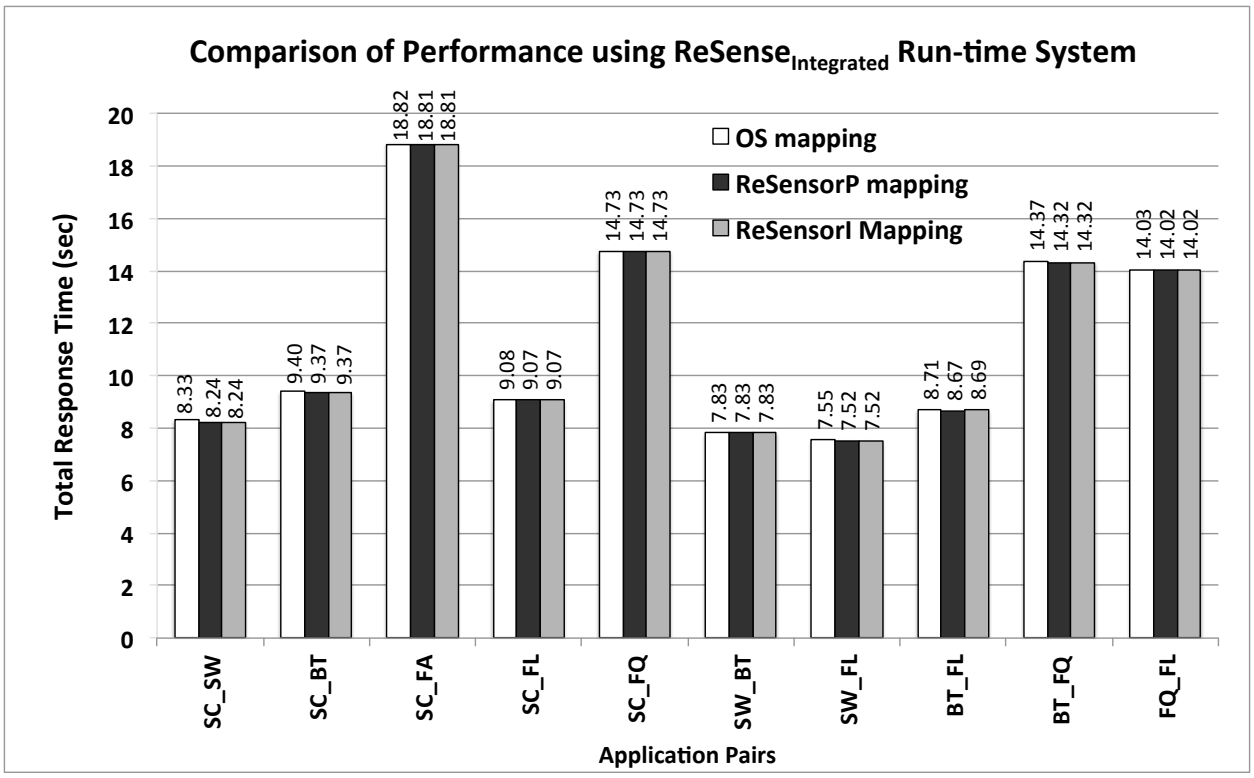


Figure 6.4: Experimental results showing total response time for the pairs of PARSEC application using OS, ReSensor$_{\mathrm{P}}$ and ReSensor$_{\mathrm{I}}$ mapping for w_P = 0.99 and w_R = 0.01 (Lower bar is better)

comparison. From the figure we observe that ReSensor$_{\mathrm{I}}$ maps the applications in the same configuration as ReSensor$_{\mathrm{P}}$ for nine pairs and reduces total response time over the native OS for eight pairs. Here, the magnitudes of performance improvement are not high because the applications are not very sensitive for L2-cache contention for using the *simlarge* input (low magnitudes of SensitivityScore$_{\mathrm{performance}}$). For BT_FL pair, ReSensor$_{\mathrm{P}}$ maps the threads in the non-sharing configuration considering the higher positive SensitivityScore$_{\mathrm{performance}}$ of *bodytrack*. On the other hand, ReSensor$_{\mathrm{I}}$ maps the threads from BT_FL pair in the space-sharing configuration considering the higher negative SensitivityScore$_{\mathrm{integrated}}$ of *fluidanimate*, which causes the total response time to increase by 0.23% over the ReSensor$_{\mathrm{P}}$-mapping.

Figure 6.5 shows the reliability results of the PARSEC pairs when the application threads are mapped using the ReSense$_{\mathrm{Integrated}}$ run-time system and the ReSensor$_{\mathrm{I}}$ algorithm for the first set of weighting factors ($w_P = 0.99$ and $w_R = 0.01$). The graph shows CVF_{mc-avg} for

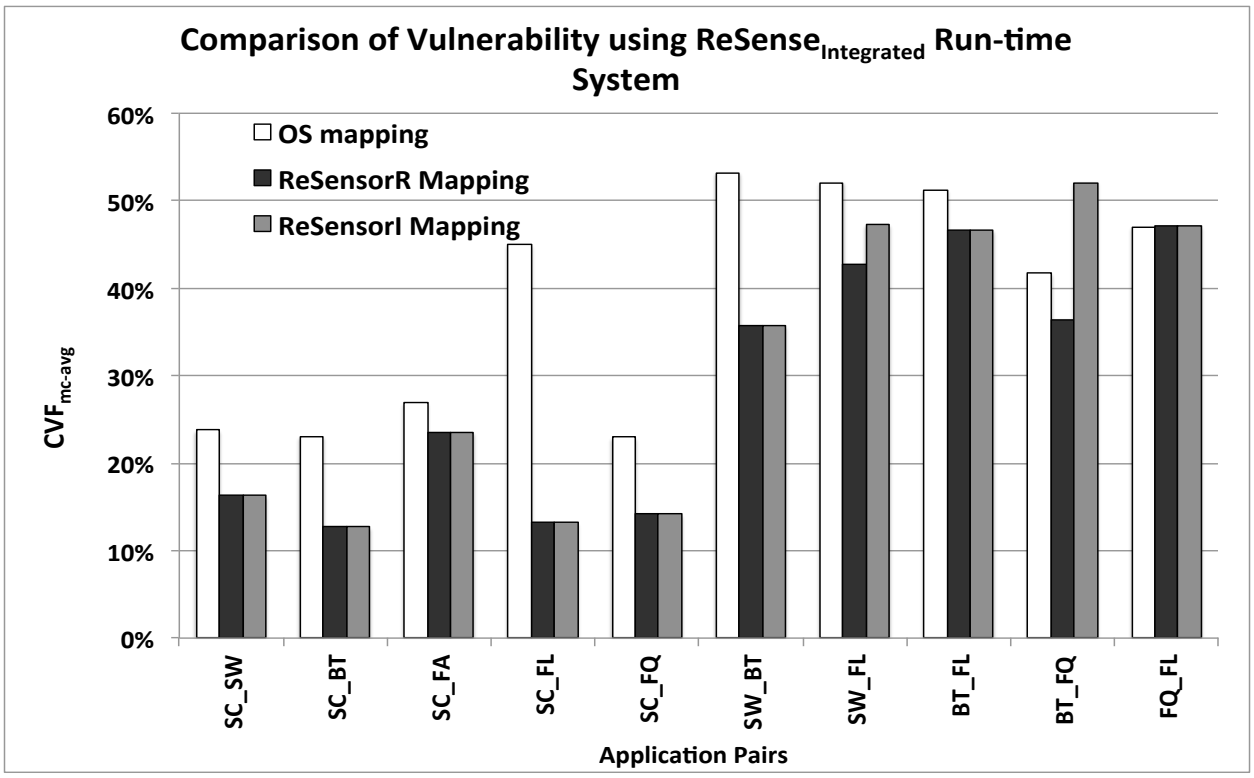


Figure 6.5: Experimental results showing average CVF_{mc} of the shared caches for pairs of PARSEC application using OS, ReSensor$_{\mathbf{R}}$ and ReSensor$_{\mathbf{I}}$ mapping for $w_P = 0.99$ and $w_R = 0.01$ (Lower bar is better)

the baseline OS and the ReSensor$_{\mathrm{R}}$ mapping for comparison. Because the weighting factors are chosen to be biased towards application performance improvements, ReSensor$_{\mathrm{I}}$ increases CVF_{mc-avg} for two pairs SW_FL and BT_FQ more than 5% and 25%, respectively than the ReSensor$_{\mathrm{R}}$-mapping. From these two results for using the weighting factors biased towards performance, we conclude that ReSense$_{\mathrm{Integrated}}$ uses the ReSensor$_{\mathrm{I}}$ algorithm to map the application threads in the right configuration that is biased more towards performance than reliability for almost all the pairs using the SensitivityScores$_{\mathrm{integrated}}$ of the applications in a pair.

Figure 6.6 shows the reliability results of the PARSEC pairs when the application threads are managed by the ReSense$_{\mathrm{Integrated}}$ run-time system and the ReSensor$_{\mathrm{I}}$ algorithm for the second set of weighting factors ($w_P = 0.30$ and $w_R = 0.70$). The weighting factors are chosen to be biased towards application reliability improvements (lower cache vulnerability). The

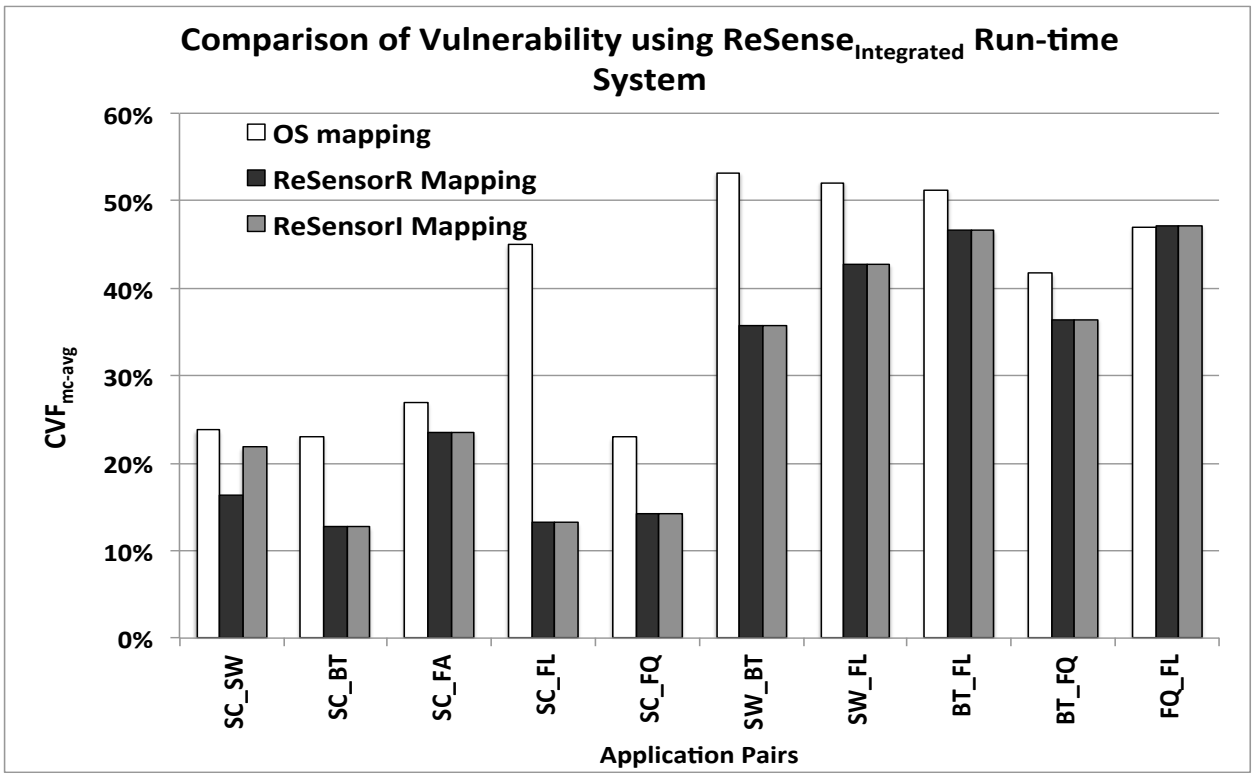


Figure 6.6: Experimental results showing average CVF_{mc} of the shared caches for pairs of PARSEC application in OS, ReSensor$_{\mathbf{R}}$ and ReSensor$_{\mathbf{I}}$ mapping for $w_P = 0.30$ and $w_R = 0.70$ (Lower bar is better)

graph shows CVF_{mc-avg} for the baseline OS and the ReSensor$_{\mathrm{R}}$ mapping for comparison. From the results, we observe that ReSensor$_{\mathrm{I}}$ performs competitively with ReSensor$_{\mathrm{R}}$ and maps the applications from nine out of ten pairs in the same configuration as ReSensor$_{\mathrm{R}}$-mapping. ReSense$_{\mathrm{Integrated}}$ reduces CVF_{mc-avg} over the native OS by up to 70%. For *SC_SW* pair, ReSensor$_{\mathrm{I}}$ maps the application in the non-sharing configuration because of higher positive magnitude of SensitivityScore$_{\mathrm{integrated}}$ for *swaptions* and increases CVF_{mc-avg} by 5% than the ReSensor$_{\mathrm{R}}$-mapping.

Figure 6.7 shows the performance results of the PARSEC pairs when the application threads are mapped using the ReSense$_{\mathrm{Integrated}}$ run-time system and the ReSensor$_{\mathrm{I}}$ algorithm for the second set of weighting factors ($w_P = 0.30$ and $w_R = 0.70$). The graph shows the total response time for the baseline OS and the ReSensor$_{\mathrm{P}}$ mapping for comparison. Because the weighting factors are chosen to be biased towards application reliability improvement,

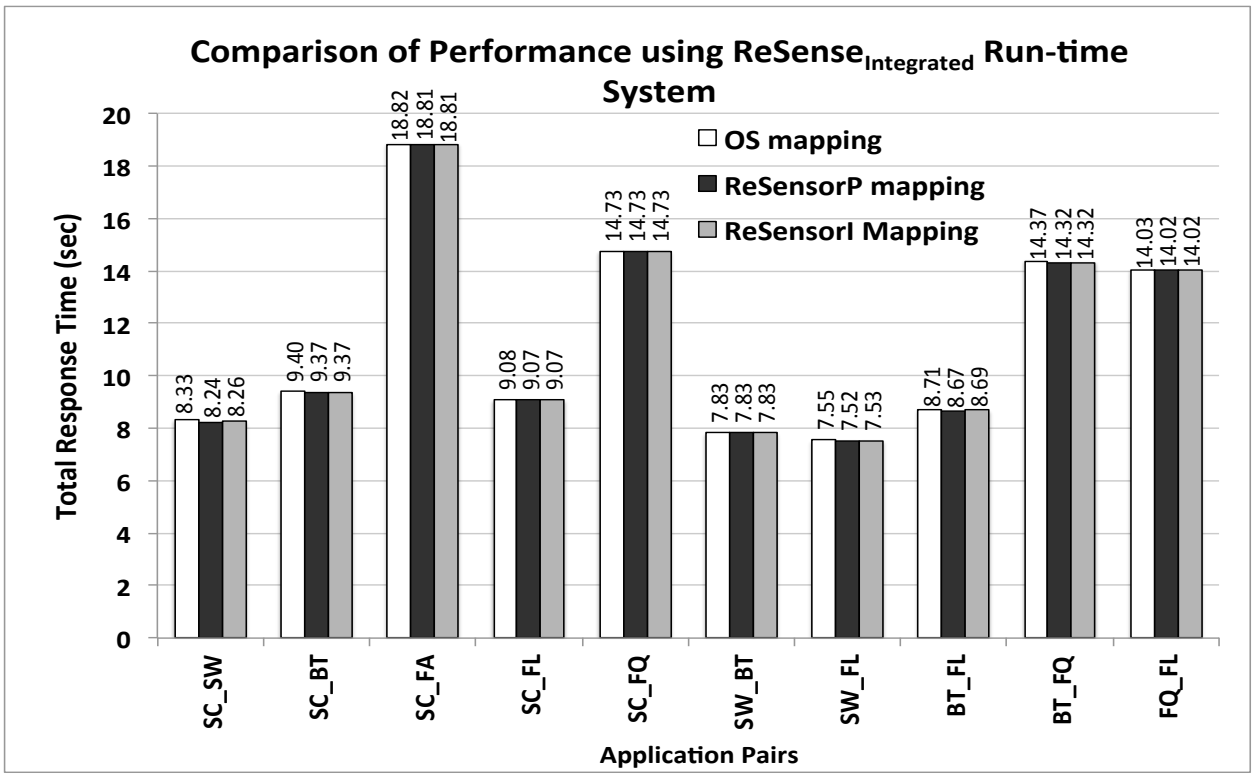


Figure 6.7: Experimental results showing total response time for the pairs of PARSEC application in OS, ReSensor$_{\mathbf{P}}$ and ReSensor$_{\mathbf{I}}$ mapping for $w_P = 0.30$ and $w_R = 0.70$ (Lower bar is better)

ReSensor$_{\mathrm{I}}$ increases total response time for *SC_SW, SW_FL* and *BT_FL* pairs by up to 0.24% compared to ReSensor$_{\mathrm{P}}$-mapping. From these two results for using the weighting factors biased towards reliability improvement, we conclude that ReSense$_{\mathrm{Integrated}}$ uses the ReSensor$_{\mathrm{I}}$ algorithm to dynamically map application threads in the right configuration as ReSensor$_{\mathrm{R}}$-mapping for almost all the pairs and reduces CVF_{mc-avg} of the shared caches using SensitivityScores$_{\mathrm{integrated}}$ of the applications in a pair.

Table 6.3 shows the experimental results of the four-application workload, *SC_SW_BT_FL*, for the third set of weighting factors $w_P = 0.20$ and $w_R = 0.80$. From the table, we observe that ReSensor$_{\mathrm{I}}$ reduces CVF_{mc-avg} by 14.70% over the native OS as preferred by the weighting factor. However, as the weighting factors are biased towards reliability, ReSensor$_{\mathrm{I}}$ increases the total response time of the workload by 4.42% compared to OS and 5.86% compared to ReSensor$_{\mathrm{P}}$-mapping. This performance degradation is caused by the positive

Mapping Configuration	CVF_{mc-avg} (%)	Total Response Time (sec)
OS-mapping	34.69	31.65
$\text{ReSensor}_{\text{P}}$	27.37	31.22
$\text{ReSensor}_{\text{R}}$	26.27	32.07
$\text{ReSensor}_{\text{I}}$	29.59	33.05

Table 6.3: Experimental results showing CVF_{mc-avg} and total response time for a four-application PARSEC workload in different mapping configurations for w_P = 0.20 and w_R = 0.80 (Lower number is better)

$\text{SensitivityScore}_{\text{integrated}}$ of *streamcluster*, which is mapped to share the same cache even though it suffers from cache contention. However, from reliability perspective, $\text{ReSensor}_{\text{I}}$ performs competitively with $\text{ReSensor}_{\text{R}}$ and determines a trade-off between performance and reliability improvements.

6.5 Summary

In this chapter, we presented a preliminary exploration of the usage of the ReSense framework and addressed the challenges of determining a trade-off between the performance and reliability improvements on modern multi- and many-core machines by creating ReSense_Integration, an integrated instance of the ReSense framework.

The characterization phase of the instance applied the characterization methodology of the framework to characterize each multi-threaded application in a workload based on its contentiousness and occupancy duration for a shared cache. It used application performance and cache vulnerability factor as the characterization metrics and calculated $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ for each application. These sensitivity scores were combined to calculate $\text{SensitivityScore}_{\text{integrated}}$, which represented the combined contention and vulnerability characteristics of each multi-threaded application in a workload. This $\text{SensitivityScore}_{\text{integrated}}$ was calculated using an equation that weighted $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$, using two weighting factors. W_P was the performance weighting factor, and w_R was the reliability weighting factors. The sign

and magnitude of a $\text{SensitivityScore}_{\text{integrated}}$ depended on the signs and magnitudes of the weighted $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$. We characterized eight PARSEC benchmarks for L2-cache contention and vulnerability to soft errors and determined their $\text{SensitivityScores}_{\text{integrated}}$. Comparing their characteristics, we observed that these applications had much higher $\text{SensitivityScore}_{\text{CVF}}$ than $\text{SensitivityScore}_{\text{performance}}$ for a L2-shared cache on the targeted platform.

In the mapping phase of the instance, the run-time system of the ReSense framework was instantiated as $\text{ReSense}_{\text{Integrated}}$, and the thread-mapping algorithm was instantiated as $\text{ReSensor}_{\text{I}}$. The $\text{ReSense}_{\text{Integrated}}$ run-time system employed the $\text{ReSensor}_{\text{I}}$ algorithm and $\text{SensitivityScores}_{\text{integrated}}$ to map application threads from a dynamic workload. $\text{ReSensor}_{\text{I}}$ dynamically determined the thread-mappings of the applications in a workload using $\text{SensitivityScores}_{\text{integrated}}$, the pre-determined integrated characteristics of the applications on the targeted platform. The algorithm optimized the objective function of this instance, which was to determine a trade-off between response time and cache vulnerability factor reductions for the chosen weighting factors. To evaluate the effectiveness of $\text{ReSense}_{\text{Integrated}}$, we used ten application pairs and one four-application workload by choosing benchmarks from PARSEC. From the pair-wise results, we observed that $\text{ReSensor}_{\text{I}}$ mapped the application threads in the same configuration chosen by $\text{ReSensor}_{\text{P}}$ for nine pairs when w_P was set higher than w_R and reduced total response time for eight pairs over the native OS. This mapping increased CVF_{avg-mc} by up to 25% than that of the $\text{ReSensor}_{\text{R}}$-mapping. Similarly, $\text{ReSensor}_{\text{I}}$ mapped the application threads in the same configuration chosen by $\text{ReSensor}_{\text{R}}$ when w_R was set higher than w_P and reduced CVF_{avg-mc} over the native OS by up to 70%. The $\text{ReSensor}_{\text{I}}$-mapping increased the total response time over $\text{ReSensor}_{\text{P}}$-mapping by 0.24%. For the four-application workload and using the weighting factors biased towards reliability, $\text{ReSensor}_{\text{I}}$ reduced CVF_{mc-avg} by 14.70% over the native OS and increased total response time by 5.86% relative to $\text{ReSensor}_{\text{P}}$-mapping.

From the results in the mapping phase, we can conclude that $\text{ReSense}_{\text{Integrated}}$ dynamically

adjusted the application thread-mappings and determined a trade-off between performance and reliability improvements for a targeted platform. The run-time system also demonstrated a novel approach in determining the performance and reliability trade-off using thread mapping, which is an application-level technique.

Chapter 7

Conclusion and Future Work

The contributions of this dissertation can have a significant impact on designing run-time systems and thread-mapping algorithms that tackle important challenges on multicore architectures using the ReSense framework. As we conclude the thesis, in this chapter, we summarize the contributions of the dissertation research, its possible impact in the research community, and the some future directions to move this research forward.

7.1 Summary of the Contributions

7.1.1 The ReSense Framework

In this thesis, we designed and developed a unified framework, ReSense, which can be used to address multicore research problems that are influenced by the characteristics of the applications in a workload. The framework included five components: a general characterization methodology, a characterization metric, a sensitivity score, a thread mapping algorithm, and a run-time system. An instance of the framework was applied in two phases: characterization and mapping. The *characterization* phase utilized the general characterization methodology and characterization metric to identify a multi-threaded application's key behavior with respect to resource usage for a targeted problem. The application characterizations were performed offline only once for each targeted resource on a particular platform. These

characteristics and behaviors were represented as *sensitivity scores* for each application in a workload. The characterization methodology can be used as a stand-alone technique to determine the resource usage behavior of any multi-threaded application.

In the online *mapping* phase, the run-time system used a thread-mapping algorithm and the sensitivity scores of the applications in a workload to determine the thread-mappings that optimize a problem-specific objective function. The run-time system was capable of handling complex dynamic workloads. Whenever the number of threads changed in the workload, the run-time system invoked the mapping algorithm to dynamically adjust the thread-mappings for the new or modified workload in the system.

Shared-resource contention and soft errors are the two major problems that limit the possible performance gains by the increasing numbers of cores on multicore machines, as well as, reliable execution. As the multicore machines continue to grow into many-core machines and transistors continue to shrink because of technology scaling, these two problems become more critical to realize the full potential of the platforms. We addressed these two problems on multicore architectures using the ReSense framework.

7.1.2 ReSense_Performance: The Performance Instance of ReSense

The performance instance of the ReSense framework, ReSense_Performance, addressed the challenges of mitigating shared-resource contention in the memory hierarchy caused by multi-threaded applications on modern multi- and many-core machines.

In the characterization phase, we instantiated the general methodology of the ReSense framework to characterize a multi-threaded application for both intra- and inter-application contention for the shared resources in the memory hierarchy using performance as the characterization metric. To demonstrate the methodology, we characterized the applications in the widely used PARSEC and NPB benchmark suites for shared-memory resource contention on four different multicore platforms. Each application's characteristic for a particular shared

resource usage was represented as $\text{SensitivityScore}_{\text{performance}}$, which was determined offline as the application ran solely using the methodology developed for intra-application contention.

In the mapping phase, the ReSense run-time system was instantiated as $\text{ReSense}_{\text{Performance}}$. The $\text{ReSense}_{\text{Performance}}$ run-time system employed the $\text{ReSensor}_{\text{P}}$ algorithm to map application threads from the input dynamic workloads. $\text{ReSensor}_{\text{P}}$ dynamically determined the thread-mappings of the multi-threaded applications in a workload in the presence of any number of co-runners using each application's $\text{SensitivityScore}_{\text{performance}}$ for a particular platform. The algorithm optimized the objective function of this instance, which was to minimize the workload's average response time and maximize throughput. $\text{ReSense}_{\text{Performance}}$ did not require the application's source code modifications.

The experimental results include:

- Thorough contention characterization of several PARSEC-2.1 and NPB-OMP-3.3 benchmarks. Two of the thirteen PARSEC benchmarks exhibited no intra-application contention for the cache resources at any level of the memory hierarchy. Nine PARSEC benchmarks exhibited inter-application contention for the L2-cache. Contention for the front-side bus and memory controller was a major factor with most the benchmarks and degraded application performance by more than 11%. All benchmarks, except one, performed better when the sibling threads used the same memory socket connection, which reduced remote memory access and its latency. On the other hand, two of the nine NPB benchmarks suffered from intra-application contention for all the shared resources on the platforms including the memory controller, L3-cache, and memory socket. Three out of nine NPB benchmarks suffered from memory socket contention among the sibling threads.

- $\text{SensitivityScores}_{\text{performance}}$ for 22 benchmarks from PARSEC-2.1 and NPB-OMP-3.3 benchmark suites for the shared resources in the memory hierarchy on four state-of-the-art platforms.

- A comprehensive empirical evaluation of the effectiveness of $\text{ReSense}_{\text{Performance}}$, which demonstrated improved average response time and throughput of dynamic workloads consisting of multiple multi-threaded applications by up to 29.34% and 46.56% respectively, over the native operating system (OS) using real hardware.

- A performance comparison of $\text{ReSensor}_{\text{P}}$'s effectiveness in mitigating contention and improving application performance with that of the optimal thread-mapping, which demonstrated that the maximum average differences with the experimentally determined optimal performance was 1.49% for average response time and 2.08% for throughput for two different workloads.

7.1.3 ReSense_Reliability: The Reliability Instance of ReSense

The reliability instance of the ReSense framework, ReSense_Reliability, addressed the challenges of minimizing the effect of soft errors in the memory resources on modern multicore architectures.

The characterization phase of ReSense_Reliability instantiated the general methodology of the ReSense framework to characterize a multi-threaded application based on its resource occupancy duration in the shared caches. It used cache vulnerability factor as the characterization metric and calculated $\text{SensitivityScores}_{\text{CVF}}$ for the applications in a workload. Each $\text{SensitivityScore}_{\text{CVF}}$ represented the vulnerability characteristics of a multi-threaded application.

In the online mapping phase of ReSense_Reliability, the run-time system of the ReSense framework was instantiated as $\text{ReSense}_{\text{Reliability}}$. The $\text{ReSense}_{\text{Reliability}}$ run-time system dynamically managed the mappings of the application threads from a given workload by employing a thread-mapping algorithm, $\text{ReSensor}_{\text{R}}$ and the pre-determined $\text{SensitivityScores}_{\text{CVF}}$. The $\text{ReSensor}_{\text{R}}$ algorithm determined the thread-mappings of the multi-threaded applications in a workload using the $\text{SensitivityScores}_{\text{CVF}}$ of the applications. The algorithm optimized the objective function of this instance, which was to minimize the overall cache vulnerability

factor to reduce the effect soft errors in caches on a workload's execution. $\text{ReSense}_{\text{Reliability}}$ did not require the application's source code modifications.

The experimental results include:

- The vulnerability characterization of several multi-threaded applications from the PARSEC benchmark suite, represented as $\text{SensitivityScore}_{\text{CVF}}$, for a write-back, write-allocate shared L2-cache.

- Several important insights from the characterization. Different thread-mapping configurations caused the cache vulnerability of different benchmarks to range from 4% up to 78%. Three benchmarks had decreased cache vulnerability and five benchmarks had increased cache vulnerability as the threads shared the same L2-cache. From the characterization results, we had two observations regarding when an application's CVF_{mc} decreased. The first observation is when application threads had a high cache read miss rate, it caused the cache-lines to be frequently updated with the fetched data blocks, which increased non-vulnerable W-W interval of its total lifetime and decreased the CVF_{mc}. The second observation is when the application threads had frequent write operations, it resulted into modified cache-lines. If there was a cache miss and a modified cache-line needed to replaced, the write-back operation of the dirty cache-line contributed to the D-Repl lifetime and increased the ACE lifetime of the cache block and consequently CVF_{mc}. These two observations and insights about application resource usage were used to reduce CVF_{mc} of the shared caches. In particular, by increasing the miss rate significantly in shared caches by choosing cache-intensive co-runner(s) and allowing applications to use multiple caches to decrease the duration of frequent write-back operations, the mapping algorithm could decrease an application's vulnerability to soft errors in a shared cache.

- The evaluation of $\text{ReSense}_{\text{Reliability}}$'s effectiveness, which demonstrated that $\text{ReSense}_{\text{Reliability}}$ effectively used $\text{ReSensor}_{\text{R}}$ mapping algorithm and reduced the overall cache vul-

nerability by up to 70%. The statistical analyses revealed that the probability of $\text{ReSense}_{\text{Reliability}}$ reducing the workload's average cache vulnerability to soft errors over the native operating system was very high, 99.5%, and the reduction of average cache vulnerability could be as high as 52%.

7.1.4 ReSense_Integration: The Integrated Performance and Reliability Instance of ReSense

In this dissertation, we presented a preliminary exploration of the usage of the ReSense framework to develop ReSense_Integration, which addressed the challenges of determining a trade-off between the performance and reliability improvements for modern multi- and many-core machines using multi-threaded applications.

The characterization phase of the instance applied the characterization methodology of the framework to characterize each multi-threaded application in a workload based on its contentiousness and occupancy duration for a shared cache. It used application performance and cache vulnerability factor as the characterization metrics and calculated $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$ for each application. These sensitivity scores were combined to calculate $\text{SensitivityScore}_{\text{integrated}}$, which represented the combined contention and vulnerability characteristics of each multi-threaded application in a workload. This $\text{SensitivityScore}_{\text{integrated}}$ was calculated using an equation that weighted $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$, using two weighting factors. W_P was the performance weighting factor, and w_R was the reliability weighting factors. The sign and magnitude of a $\text{SensitivityScore}_{\text{integrated}}$ depended on the signs and magnitudes of the weighted $\text{SensitivityScore}_{\text{performance}}$ and $\text{SensitivityScore}_{\text{CVF}}$.

In the mapping phase of the instance, the run-time system of the ReSense framework was instantiated as $\text{ReSense}_{\text{Integrated}}$, and the thread-mapping algorithm was instantiated as $\text{ReSensor}_{\text{I}}$. The $\text{ReSense}_{\text{Integrated}}$ run-time system employed the $\text{ReSensor}_{\text{I}}$ algorithm and $\text{SensitivityScores}_{\text{integrated}}$ to map application threads from the input workload. $\text{ReSensor}_{\text{I}}$

dynamically determined the thread-mappings of the applications in a workload using $\text{SensitivityScores}_{\text{integrated}}$, the pre-determined integrated characteristics of the applications on the targeted platform. The algorithm optimized the objective function of this instance, which was to determine a trade-off between response time and cache vulnerability factor reductions for the chosen weighting factors.

The experimental results include:

- Combined characterization of cache contention and vulnerability for eight PARSEC applications. The characterization revealed that these applications had higher $\text{SensitivityScore}_{\text{CVF}}$ than $\text{SensitivityScore}_{\text{performance}}$ and consequently, were more vulnerable to soft errors than being contentious in L2-caches on the same targeted platform.

- The evaluation of $\text{ReSense}_{\text{Integrated}}$'s effectiveness in determining a trade-off between performance and reliability improvements. From the pair-wise results, we observed that $\text{ReSensor}_{\text{I}}$ mapped the application threads in the same configuration chosen by $\text{ReSensor}_{\text{P}}$ for nine pairs when w_P was set higher than w_R and reduced total response time for eight pairs over the native OS. This mapping increased CVF_{avg-mc} by up to 25% than that of the $\text{ReSensor}_{\text{R}}$-mapping. Similarly, $\text{ReSensor}_{\text{I}}$ mapped the application threads in the same configuration chosen by $\text{ReSensor}_{\text{R}}$ when w_R was set higher than w_P and reduced CVF_{avg-mc} over the native OS by up to 70%. The $\text{ReSensor}_{\text{I}}$-mapping increased the total response time over $\text{ReSensor}_{\text{P}}$-mapping by 0.24%. For the four-application workload and using the weighting factors biased towards reliability, $\text{ReSensor}_{\text{I}}$ reduced CVF_{mc-avg} by 14.70% over the native OS and increased total response time by 5.86% relative to $\text{ReSensor}_{\text{P}}$-mapping.

7.2 Future Work

In this section, we describe the future direction of the research presented in this thesis.

7.2.1 Applying the framework to other instances

In this research, we have not considered the effect of hardware prefetchers on an application's contention or vulnerability behavior. An application's memory access behavior affects the way the pre-fetchers fetch the cachelines to reduce cache miss latency. Therefore, hardware prefetchers have an impact on an application's cache contention behavior and eventually would affect its performance [110]. In addition, as the cachelines are prefetched before the application accesses them, the prefetchers can also affect the vulnerable lifetime of the caches. Therefore, ReSense framework can be instantiated to mitigate cache contention and reduce cache vulnerability by considering the effect of hardware prefetchers.

The described ReSense framework can be also applied to address other research problems that are influenced by application characteristics. One of the challenges on multicore machines is the management of hardware environment from the application-level, e.g., temperature and power. Both power and temperature are influenced and affected by application resource usage behavior. For example, if an application has many floating-point operations, then it accesses the floating-point unit very frequently, which eventually increases the processor temperature and power consumption. We can apply the ReSense framework to create temperature and power instances to address the power and thermal challenge for a multicore architecture. These instances first need to characterize multi-threaded applications based on its temperature and power consumption behaviors, and later manage the application execution by choosing the mapping configurations that reduce thermal emergency and power consumption on the targeted CMPs.

7.2.2 Phase-level characterization

In this research, we designed the ReSense framework to characterize an application based on its entire execution. That is, the framework is designed for mapping application threads based on its *application-level* characterization. One direction is to extend the framework to perform a *phase-level* characterization by determining an application's contention and

vulnerability characteristics for individual phases during its execution. However, some phase changes are regular or periodic, and others are random. Therefore, it is very difficult to detect and handle phase changes during application execution because each phase change can happen anytime and can last for any time duration.

Using prediction techniques to determine when a future phase change would occur and how long it would last, the framework can be extended for phase-level characterization and mapping. Depending on when a phase change would happen, the characterization phase needs to apply the characterization methodology to determine an application's behaviors for each phase. Then all these phase-level characterizations can be represented as an array of sensitivity scores. Once the phase-level characterization is done, the mapping phase can select the thread-mappings that optimize the objective function for the applications in a workload for each phase change, using the corresponding sensitivity scores.

7.2.3 Instance for minimizing vulnerability of micro-architectural resources and write-though caches

In this research, we presented the reliability instance of the framework that minimized an application's cache vulnerability to soft errors in shared write-back caches. We consider a shared cache as the targeted resource because it occupies a bigger die area in the processor and is more vulnerable than the core resources and L1-cache. The framework can be used to reduce application vulnerability for micro-architectural resources and write-though caches.

In the characterization phase of the framework, a multi-threaded application's vulnerability characteristics can be determined by following the methodology in Section 5.2.3 and using AVF as the characterization metric. For a micro-architectural structure, the AVF is the percentage of cycles that it processes ACE bits or instructions. For characterizing a multi-threaded application for its vulnerability to soft errors in the micro-architectural resources, we measure an application's AVF in the configurations described by the characterization methodology, and determine the $\text{SensitivityScores}_{\text{AVF}}$ using Equation 7.1. Here, $AVF_{non-sharing}$ is the

average AVF of the targeted micro-architectural resource when the threads use two separate resources, and $AVF_{sharing}$ is the AVF of the targeted micro-architectural resource when the threads use the same resource. Once the scores are determined, then the $\text{ReSensor}_{\text{R}}$ algorithm is used to determine the thread-mappings of the multi-threaded applications in a workload.

$$SensitivityScore_{AVF} = \frac{(AVF_{non-sharing} - AVF_{sharing}) * 100}{AVF_{non-sharing}} \tag{7.1}$$

A write-through cache is only vulnerable during the R-R and W-R intervals because the replaced blocks in the write-through caches need not be written back to the resources at the lower level of the memory hierarchy. Therefore, for a write-through cache, the ACE lifetime consists of R-R and W-R time intervals, and the CVF can be calculated using Equation 7.2. Here, $Lifetime_{R-R,W-R}$ represents the vulnerable lifetime of $block_i$ for a write-through cache, which consists of time intervals of W-R and R-R cache access patterns. Equation 7.3 can be used to calculate the CVF_{mc} for a write-through cache with MESI protocol for multicore machines. $Lifetime_{S-S,S-E,E-E,E-S,M-E,M-S}$ represents the total vulnerable lifetime between the cacheline state changes from S to S, S to E, E to E, E to S, M to E, M to S, according to the MESI protocol. Then the same characterization methodology and mapping algorithm for a write-back cache can be used to minimize vulnerability for a write-through cache.

$$CVF_{uniprocessor} = \frac{\sum_{i=1}^{n} Lifetime_{R-R,W-R}(block_i)}{\sum_{i=1}^{n} TotalLifetime(block_i)} \tag{7.2}$$

$$CVF_{mc} = \frac{\sum_{i=1}^{n} Lifetime_{S-S,S-E,E-E,E-S,M-E,M-S}(block_i)}{\sum_{i=1}^{n} TotalLifetime(block_i)} \tag{7.3}$$

7.2.4 Combine techniques for vulnerability minimization

In this research, we present an application-level technique, thread-mapping, as an error prevention technique for soft errors in shared caches. As discussed in Section 2.3, there exists other architecture-level and circuit-level techniques that are also used to reduce application

vulnerability in caches. One possible future work would be to combine the application-level, architecture-level, and circuit-level techniques as a hybrid cross-layer approach to minimize application vulnerability towards soft errors.

7.2.5 Model CVF on real hardware

In the reliability instance of ReSense, we used a simulation infrastructure to characterize and map application threads based on the CVF model. We used a simulation infrastructure because we needed some very detailed lifetime information of each cache-line to calculate CVF of the caches. One future direction would be to model CVF such that the reliability instance can be applied on real hardware. One possible way to model CVF is by using the hardware performance counters to obtain the lifetime information. The major challenge in designing the CVF model using real hardware components is model validation. Such model validation requires rigorous analyses for different cache hierarchies, configurations, and parameters to make it very general. We leave this model design and verification as future work.

7.2.6 Different variations of the integrated instance

In this dissertation, we explored the usage of the ReSense framework for addressing the targeted problems of cache contention and soft errors in the shared caches. We used an addition function to combine the contentious and vulnerability characterization of the applications. This instance can be explored more and extended by choosing different functions to combine the contentious and vulnerability characteristics of the multi-threaded applications. More experiments are needed in the integrated instance to fully understand the integration of problems using ReSense.

Bibliography

[1] Wei Zhang. Computing cache vulnerability to transient errors and its implication. In *International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2005.

[2] V. Huard, F. Cacho, and X. Federspiel. Technology scaling and reliability challenges in the multicore era. In *2013 IEEE International Reliability Physics Symposium (IRPS)*, pages 3A.5.1–3A.5.7, April 2013.

[3] Lei Jin and Sangyeun Cho. SOS: A software-oriented distributed shared cache management approach for chip multiprocessors. In *Int'l. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2009.

[4] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[5] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[6] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[7] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[8] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Int'l. Symp. on Microarchitecture (MICRO)*, 2006.

[9] Yuejian Xie and Gabriel H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Int'l Symp. on Computer Architecture (ISCA)*, 2009.

[10] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: online contention detection and response. In *Int'l Symp. on Code Generation and Optimization (CGO)*, 2010.

[11] A. Krishna, A. Samih, and D. Solihin. Data sharing in multi-threaded applications and its impact on chip design. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[12] Xuejun Yang, Zhiyuan Wang, Jingling Xue, and Yun Zhou. The reliability wall for exascale supercomputing. *Computers, IEEE Transactions on*, 61(6):767 –779, june 2012.

[13] R. Baumann. Soft errors in advanced computer systems. *Design Test of Computers, IEEE*, 22(3), may-june 2005.

[14] Wangyuan Zhang, Xin Fu, Tao Li, and JosÃĬ A. B. Fortes. An analysis of microarchitecture vulnerability to soft errors on simultaneous multithreaded architectures. In *Int'l Symp. on Performance Analysis of Systems Software (ISPASS)*, 2007.

[15] S.S. Mukherjee, J. Emer, and S.K. Reinhardt. The soft error problem: an architectural perspective. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2005.

[16] Vilas Sridharan, Hossein Asadi, Mehdi B. Tahoori, and David Kaeli. Reducing data cache susceptibility to soft errors. *IEEE Transactions on Dependable and Secure Computing*, 3(4), October 2006.

[17] Kristen R. Walcott, Greg Humphreys, and Sudhanva Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *International symposium on Computer architecture (ISCA)*, 2007.

[18] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *International symposium on Computer Architecture (ISCA)*, 2002.

[19] T. M. Jones, M. F. P. OâĂŹBoyle, and O. Ergin. Evaluating the effect of compiler optimizations on AVF. In *Workshop on Interaction Between Compilers and Computer Architecture*, 2008.

[20] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[21] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Optimal task assignment in multithreaded processors: a statistical approach. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[22] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *International Symposium on Computer Architecture (ISCA)*, ISCA '06, 2006.

[23] Zao Liu, Tailong Xu, S.X.-D. Tan, and Hai Wang. Dynamic thermal management for multi-core microprocessors considering transient thermal effects. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013.

[24] S. Reda, R. Cochran, and A.K. Coskun. Adaptive power capping for servers with multithreaded workloads. *Micro, IEEE*, 32(5):64–75, Sept 2012.

[25] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *International Symposium on Computer Architecture*, 2009.

[26] Isil Oz, Haluk Rahmi Topcuoglu, Mahmut Kandemir, and Oguz Tosun. Performance-reliability tradeoff analysis for multithreaded applications. In *Design Automation and Testing in Europe (DATE)*, 2012.

[27] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2011.

[28] Jason Mars and Mary Lou Soffa. Synthesizing contention. In *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[29] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Int'l Symp. on Microarchitecture (MICRO)*, 2011.

[30] Y Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In *In Proc. of CMP-MSI, held in conjunction with ISCA-35*, 2008.

[31] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[32] Haoqiang Jin, Robert Hood, Johnny Chang, Jahed Djomehri, Dennis Jespersen, and Kenichi Taylor. Characterizing application performance sensitivity to resource contention in multicore architectures. Technical report, NASA Ames Research Center, 2009.

[33] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Int'l Symp. on Computer Architecture (ISCA)*, 2011.

[34] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring interference between live datacenter applications. In *Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, 2012.

[35] Hao Luo, Xiaoya Xiang, and Chen Ding. Characterizing active data sharing in threaded applications using shared footprint. In *International Workshop on Dynamic Analysis (WODA)*, 2013.

[36] Ragavendra Natarajan and Mainak Chaudhuri. Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies. In *IISWC*, 2013.

[37] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Int'l Symp. on Workload Characterization (IISWC)*, 2009.

[38] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Int'l Conf. on Parallel architectures and compilation techniques (PACT)*, 2008.

[39] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[40] Jun Yan and Wei Zhang. Compiler-guided register reliability improvement against soft errors. In *ACM international conference on Embedded software (EMSOFT)*, 2005.

[41] Jongeun Lee and A. Shrivastava. Static analysis to mitigate soft errors in register files. In *Design, Automation Test in Europe (DATE)*, 2009.

[42] Vilas Sridharan and David R. Kaeli. Using hardware vulnerability factors to enhance avf analysis. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[43] Vilas Sridharan and David R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *Int'l. Symp. on High Performance Computer Architecture (HPCA)*, 2009.

[44] Abhinav Agrawal Bagus Wibowo and James Tuck. Toward a cross-layer approach for dynamic vulnerability estimation. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[45] Shuai Wang. Characterizing system-level vulnerability for instruction caches against soft errors. In *DFT*, 2011.

[46] Arijit Biswas, Niranjan Soundararajan, Shubhendu S. Mukherjee, and Sudhanva Gurumurthi. Quantized avf: A means of capturing vulnerability variations over small windows of time. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2009.

[47] Niranjan Soundararajan, Anand Sivasubramaniam, and Vijay Narayanan. Characterizing the soft error vulnerability of multicores running multithreaded applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, 2010.

[48] Xin Fu, J. Poe, Tao Li, and J. A B Fortes. Characterizing microarchitecture soft error vulnerability phase behavior. In *Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.

[49] Lide Duan, Lu Peng, and Bin Li. Predicting architectural vulnerability on multi-threaded processors under resource contention and sharing. *IEEE Transactions on Dependable and Secure Computing*, March 2013.

[50] Arun Arvind Nair, Lizy Kurian John, and Lieven Eeckhout. AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[51] Hamed Tabkhi and Gunar Schirner. Application-specific power-efficient approach for reducing register file vulnerability. In *Design Automation and Test In Europe (DATE)*, 2012.

[52] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[53] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing Application Memory Error Vulnerability to Optimize Data Center Cost via Heterogeneous-Reliability Memory. In *DSN*, 2014.

[54] Zhe Ma, Trevor Carlson, Wim Heirman, and Lieven Eeckhout. Evaluating application vulnerability to soft errors in multi-level cache hierarchy. In *International Conference on Parallel Processing - Volume 2*, 2012.

[55] Shuai Wang, Jie Hu, and Sotirios G. Ziavras. On the characterization and optimization of on-chip cache reliability against soft errors. *IEEE Trans. Comput.*, 58(9), September 2009.

[56] Jongeun Lee and Aviral Shrivastava. A compiler optimization to reduce soft errors in register files. In *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems(LCTES)*, 2009.

[57] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[58] Livio Soares, David Tam, and Michael Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Int'l Symp. on Microarchitecture (MICRO)*, 2008.

[59] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2008.

[60] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for cmps. In *International Conference on Supercomputing (ICS)*, 2010.

[61] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[62] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. No more backstabbing... a faithful scheduling policy for multithreaded programs. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[63] Murali Krishna Emani, Zheng Wang, and Michael F.P. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Int'l Symp. on Code Generation and Optimization (CGO)*, 2013.

[64] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, , and Mani Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2013.

[65] Elkin Garcia, Daniel Orozco, Robert Pavel, and Guang R. Gao. A discussion in favor of dynamic scheduling for regular applications in many-core architectures. In *Workshop on Multithreaded Architectures and Applications*, 2012.

[66] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2010.

[67] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Symp. on Parallel Algorithms and Architectures (SPAA)*, 2007.

[68] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[69] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, Jr., and Joel Emer. CRUS: cache replacement and utility-aware scheduling. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[70] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2005.

[71] Chi Xu, Xi Chen, R.P. Dick, and Z.M. Mao. Cache contention and application performance prediction for multi-core systems. In *Int'l Symp. on Performance Analysis of Systems Software (ISPASS)*, 2010.

[72] Sangyeun Cho and Lei Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Int'l. Symp. on Microarchitecture (MICRO)*, 2006.

[73] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[74] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *European Conf. on Computer Systems (EuroSys)*, 2009.

[75] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. SRM-buffer: An OS buffer management technique to prevent last level cache from thrashing in multicores. In *Conf. on Computer Systems*, 2011.

[76] C. L. Chen. Error-correcting codes for semiconductor memories. In *International symposium on Computer architecture (ISCA)*, 1984.

[77] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Int'l Symp. on Computer Architecture (ISCA)*, 2002.

[78] Nicholas J. Wang, Student Member, and Sanjay J. Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Trans. Dependable Secur. Comput*, 3, 2006.

[79] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *International symposium on Computer Architecture (ISCA)*, 2003.

[80] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Int'l Symp. on Computer architecture (ISCA)*, 2000.

[81] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Int'l Symp. on Code Generation and Optimization (CGO)*, 2007.

[82] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *International symposium on Code Generation and Optimization (CGO)*, 2005.

[83] Quming Zhou and K. Mohanram. Transistor sizing for radiation hardening. In *IEEE International Reliability Physics Symposium (RPS)*, 2004.

[84] Y. Zorian, V.A. Vardanian, K. Aleksanyan, and K. Amirkhanyan. Impact of soft error challenge on soc design. In *IEEE International On-Line Testing Symposium (OLTS)*, 2005.

[85] N. Derhacobian. Embedded memory reliability: The ser challenge. In *International Workshop on Memory Technology, Design and Testing (MTDT)*, pages 104–110, 2004.

[86] V. Degalahal, R. Ramanarayanan, N. Vijaykrishnan, Y. Xie, and M.J. Irwin. The effect of threshold voltages on the soft error rate [memory and logic circuits]. In *International Symposium on Quality Electronic Design (ISQED)*, 2004.

[87] Arijit Biswas, Paul Racunas, Razvan Cheveresan, Joel Emer, Shubhendu S. Mukherjee, and Ram Rangan. Computing architectural vulnerability factors for address-based structures. In *Int'l Symp. on Computer Architecture (ISCA)*, 2005.

[88] G.-H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli. Balancing performance and reliability in the memory hierarchy. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2005.

[89] A.M. Saleh, J.J. Serrano, and J.H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *Reliability, IEEE Transactions on*, 39(1):114–122, Apr 1990.

[90] Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 37–42, Washington, DC, USA, 2004. IEEE Computer Society.

[91] Reiley Jeyapaul and Aviral Shrivastava. Smart cache cleaning: energy efficient vulnerability reduction in embedded processors. In *International conference on Compilers, architectures and synthesis for embedded systems (CASES)*, 2011.

[92] Melina Demertzi, Murali Annavaram, and Mary Hall. Analyzing the effect of compiler optimizations on application reliability. In *Int'l Symp. on Workload Characterization (IISWC)*, 2011.

[93] A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F.R.P. Pinto, H. Guzman-Miranda, and M.A. Aguirre. Compiler-directed soft error mitigation for embedded systems. *Dependable and Secure Computing, IEEE Transactions on*, 2012.

[94] S. Rehman, M. Shafique, and J. Henkel. Instruction scheduling for reliability-aware compilation. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2012.

[95] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. Raise: Reliability-aware instruction scheduling for unreliable hardware. In *Asia and South Pacific Design Automation Conference (ASP-DAC),*, 2012.

[96] Ismail Kadayif and Mahmut Kandemir. Modeling and improving data cache reliability: 1. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2007.

[97] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications (SC), 1991.

[98] S. Eranian. Perfmon2: A flexible performance monitoring interface for Linux. In *Linux Symp.*, 2006.

[99] Wei Wang, Tanima Dey, Ryan Moore, Mahmut Aktasoglu, Bruce R. Childers, Jack Davidson, Mary Jane Irwin, Mahmut Kandemir, and Mary Lou Soffa. REEact: A customizable virtual execution manager for multicore platforms. In *Int'l Conf. on Virtual Execution Environments (VEE)*, 2012.

[100] V. P. S. Siddha and A. Mallick. Chip multi processing (cmp) aware linux kernel scheduler. In *In Ottawa Linux Symposium*, 2005.

[101] J. Susan Milton and Jesse C. Arnold. *Introduction to Probability and Statistics*. Tata McGraw Hill Publishing Company, New Delhi, India, fourth edition, 2003.

[102] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: positional effects in dram and sram faults. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[103] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *International Symposium on Microarchitecture (MICRO)*, 2003.

[104] Tanima Dey, Steven Raasch, Jon Stephan, and Arijit Biswas. SDC virus: An application for ser model validation. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[105] A. Biswas, C. Recchia, S.S. Mukherjee, V. Ambrose, L. Chan, A. Jaleel, A.E. Papathanasiou, M. Plaster, and N. Seifert. Explaining cache ser anomaly using due avf measurement. In *Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2010.

[106] S Mitra, Ming Zhang, N. Seifert, T. M. Mak, and Kee Sup Kim. Built-in soft error resilience for robust system design. In *International Conference on Integrated Circuit Design and Technology (ICICDT)*, May 2007.

[107] Yu Cheng, Anguo Ma, Yuxing Tang, and Minxuan Zhang. Accurate vulnerability estimation for cache hierarchy. In *International Conference on Networked Computing and Advanced Information Management (NCM)*, June 2011.

[108] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[109] Simics simulator. http://http://www.windriver.com/products/simics/.

[110] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Poster: A study of the effect of prefetching in share-resource contention. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.