THREAD VULNERABILITY FOR MULTICORE ARCHITECTURES

by

Işıl Öz

B.S., Computer Engineering, Marmara University, 2004

M.S., Computer Engineering, Marmara University, 2008

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

Graduate Program in Computer Engineering
Boğaziçi University
2013

THREAD VULNERABILITY FOR MULTICORE ARCHITECTURES

APPROVED BY:

Prof. Oğuz Tosun (Thesis Supervisor)	....
Prof. Haluk Rahmi Topçuoğlu (Thesis Co-supervisor)	....
Prof. Can Özturan	....
Assoc. Prof. Alper Şen	....
Assist. Prof. Zeki Bozkuş	....

DATE OF APPROVAL: 08.05.2013

ACKNOWLEDGEMENTS

I would like to thank my advisors, Prof. Oğuz Tosun and Prof. Haluk Topçuoğlu for their academic support, guidance, and encouragement throughout my graduate study and the completion of this thesis. I would like to thank Prof. Mahmut Kandemir from Pennsylvania State University for his valuable feedback and guidance during my dissertation. I would also like to thank Prof. Can Özturan, Assoc. Prof. Alper Şen, and Assist. Prof. Zeki Bozkuş for serving my thesis committee.

I gratefully acknowledge the financial support of The Scientific and Technological Research Council of Turkey (TUBITAK) with a research grant (Project Number: 108E035). Additionally a partial fund is provided by The Marmara University Scientific Research Committee with a research grant (Project Number: FEN-A-200611-0210, 2011) and the Boğaziçi University Foundation (BUVAK).

I would like to thank my colleagues from the department of Computer Engineering at Marmara University for their support and friendship. In particular, I would like to thank Sanem Arslan Yılmaz, Berna Kiraz, Betül Demiröz Boz, Fatma Çorut Ergin, and Emel Küpçü, for their invaluable support throughout the years. I would also like to thank Fuat Geleri for his academic and technical support as well as his friendship.

I am grateful to my unique family; my mother Vahide Hasırcıoğlu, my father Ayhan Hasırcıoğlu for their endless love and great support during my dissertation like as during my whole life. Finally, I would like to thank my husband Dindar Öz for his patience, support, motivation, and everything that I cannot express with words.

ABSTRACT

THREAD VULNERABILITY FOR MULTICORE ARCHITECTURES

Continuously reducing transistor sizes and aggressive low power operating modes employed by modern architectures tend to increase transient error rates. A metric of reliability is required in order to evaluate approaches that address soft errors. This thesis explores a soft error vulnerability analysis of parallel applications running on multicore architectures. We propose and evaluate a novel metric, Thread Vulnerability Factor, in order to quantify thread vulnerability and to qualify the relative vulnerability of parallel applications to soft errors. We present the analytical definition of our metric, and develop a framework to calculate the metric value by gathering application data. To demonstrate the validity of the metric, fault injection based experiments are conducted for multithreaded applications. This thesis also presents the performance-vulnerability analysis of parallel applications for a variety of applications and discusses the effects of design choices on system performance and reliability. By considering tradeoff between these two concerns, we observe that design choice becomes clear for some of the applications which provide different vulnerability values with almost equal performance. Additionally, we propose and evaluate reliability-aware core partitioning schemes for multicore architectures. A simulation study with various workloads consisting of multiple multithreaded applications is performed in order to evaluate the proposed partitioning schemes. We also present a thread-level vulnerability assessment tool by considering user preferences; and we propose a novel critical thread identification algorithm to determine critical thread and critical thread region in a multithreaded application. We utilize our algorithm to determine the thread for redundant execution in a partial fault tolerance system and demonstrate the efficiency of the method by providing vulnerability values for executions with different redundancy levels.

ÖZET

ÇOK ÇEKİRDEKLİ MİMARİLERDE İŞ PARÇACIĞI GÜVENİLİRLİĞİ

Modern işlemci teknolojisinde transistör boyutlarının gittikçe küçülmesi ve transistörlerin çok daha hızlı frekanslarda çalışması nedenleri ile, yonga bileşenlerinin geçici hata oranları artmaktadır. Geçici hatalar için sunulan çözümlerin değerlendirilmesi için bir güvenilirlik metriğine ihtiyaç duyulmaktadır. Bu tez, çok çekirdekli mimarilerde çalışan paralel uygulamaların geçici hata hassasiyetlerini incelemektedir. İlk olarak, iş parçacıklarının hata hassasiyetlerini ölçen ve paralel uygulamaların göreceli hata hassasiyetlerini belirleyen, İş Parçacığı Hasar Görebilirlik Faktörü olarak isimlendirdiğimiz bir metrik önerilmektedir. Çalışmamız kapsamında, metriğin analitik tanımı verilerek uygulama verisinden metrik değerini hesaplayacak bir yapı oluşturulmuştur. Metriğin doğrulanmasına yönelik olarak, paralel uygulamalar için hata enjeksiyon deneyleri uygulanmıştır. Bu tezde ayrıca, farklı problemlerin paralel uygulamaları için performans-hata hassasiyeti analizi yapılarak farklı tasarım seçeneklerinin sistem performansı ve güvenilirliği üzerindeki etkileri incelenmiştir. Bu iki özelliği hesaba katarak yaptığımız analizler sonucunda, birbirine yakın performans değerlerine sahip ancak farklı hata hassasiyeti gösteren iki seçenek için tercih belirgin bir şekilde ortaya çıkmaktadır. Bu tez ayrıca, çok çekirdekli sistemler için güvenilirlik tabanlı çekirdek paylaştırma stratejileri önermektedir. Çekirdek paylaştırma stratejilerimizi değerlendirmek için, çok iş parçacıklı birden fazla uygulamadan oluşan iş yükleriyle deneysel çalışmalar yapılmıştır. Bu tezde ayrıca, iş parçacığı seviyesinde hassasiyet analizi yapılarak uygulamadaki kritik iş parçacığı ve iş parçacığı bölgesi tespiti için bir kritik iş parçacığı belirleme algoritması önerilmiştir. Bu algoritma, güvenilirliği arttırmak için kullanılan kısmi çoklama yönteminde en önemli kod parçacıklarının tespitinde kullanılmış, farklı çoklama seviyeleriyle ölçülen hassasiyet değerleriyle tekniğin etkinliği gösterilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$LVF(T_i)$	Local vulnerability factor for thread i
RV	Parameter for total number of remote memory accesses
$RVF(T_i)$	Remote vulnerability factor for thread i
$rVF(T_i, T_j)$	Remote vulnerability factor for thread i induced by thread j
$TVF(T_i)$	Thread vulnerability factor for thread i
TVF_{RF}	Thread vulnerability factor for register file resources
TVF_{ALU}	Thread vulnerability factor for ALU resources
TVF_{mem}	Thread vulnerability factor for memory resources
w_L	Local vulnerability weight coefficient
w_R	Remote vulnerability weight coefficient

LIST OF ACRONYMS/ABBREVIATIONS

ACE	Architecturally Correct Execution
AVF	Architectural Vulnerability Factor
CMP	Chip Multiprocessor
DMR	Dual Modular Redundancy
ECC	Error Correcting Code
LVF	Local Vulnerability Factor
NWRL	Normalized Weighted Reliability Loss
NWS	Normalized Weighted Speedup
NWVDPG	Normalized Weighted Vulnerability-Delay Product Gain
PVF	Program Vulnerability Factor
RVF	Remote Vulnerability Factor
SDC	Silent Data Corruption
SEU	Single Event Upset
SMT	Simultaneous Multithreading
QoS	Quality of Service
TMR	Triple Modular Redundancy
TVF	Thread Vulnerability Factor
VDP	Vulnerability Delay Product
WRL	Weighted Reliability Loss
WS	Weighted Speedup
WVDPG	Weighted Vulnerability-Delay Product Gain

1. INTRODUCTION

The performance of microprocessors has been increasing exponentially by faster processor and memory chips as well as increasing number of the resources in the system [6]. Although the techniques to accelerate single processor do not bring much performance gain with excessive power consumption and high design complexity, chip multiprocessors (CMPs), which have multiple cores in a single chip, yield high performance by the execution of multiple concurrent software processes in the system in a power and area efficient way. Replacing a large processor with several small processors in a single die has been proven successful in CMP architectures. In these architectures, one small core works slowly than one large core but overall throughput of the system becomes much higher by simultaneous execution of several cores in the same chip size. The communication overhead resulted from the execution of multiple threads at different cores is minimized in the shared-memory CMP architectures. While the distinct threads are executed in distinct processors in a distributed multiprocessor system which causes high latency due to inter-thread communications, there is not much performance decrease due to communication delays in a chip multiprocessor system which has several cores in the same chip. With these several advantages, CMPs have been accepted as the most promising architecture for higher performance [6].

To be able to place large number of transistors on a single chip in CMPs, it is necessary to scale down the size of the transistors with higher frequencies. Since the size of transistors continuously reduces and the transistors operate in higher frequencies with low power modes, the chip components become more error prone. Transient errors are temporary corruptions of hardware elements' operations due to environmental effects [7]. The transient error rate in the chip processors increases by the trend in the transistor size. Moreover, reduction of the voltage of the transistors reduces noise margin, thus the chip is more susceptible to transient fault [8].

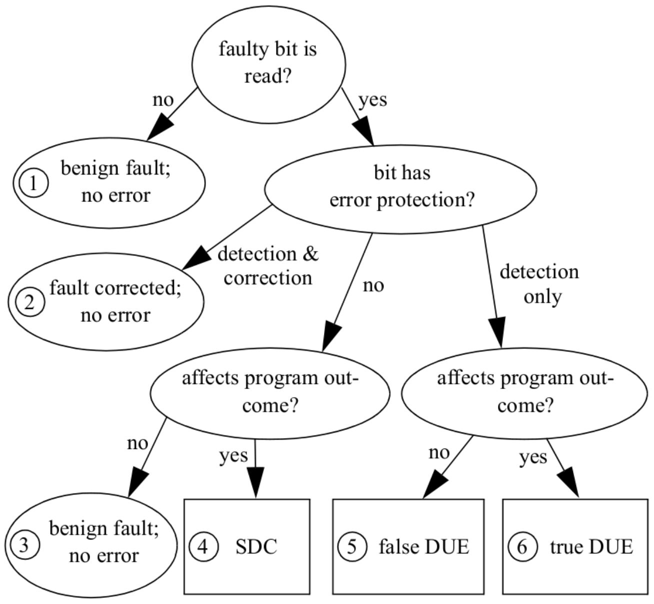


Figure 1.1. Possible outcomes of a single-bit fault [1].

1.1. Soft Error Resilience

Soft errors, which constitute a class of transient errors, result from a fault in a single-bit (single-event upsets (SEU)) in the processors due to particle strikes, cosmic rays, electrical noise, or other environmental effects [1, 7, 8]. Figure 1.1 [1] shows the possible outcomes of a single-bit fault in a hardware structure. The most critical case is silent data corruption (SDC) which is represented by Outcome 4 in the figure, since it results in unnoticed erroneous outputs in the system. The errors called DUE (detected unrecoverable errors) are detected by the system but they are not recovered by any mechanism. The events which cause faulty bit read, but do not affect program output are called as *false* DUE, while *true* DUE events affect program output by affecting single bit. In order to reduce error rates, fault-tolerance methods including both error detection and recovery techniques are applied to the system by eliminating the cases 3 through 6 in the figure.

In order to solve the problems resulting from soft errors in chip multiprocessors, many fault-tolerance techniques have been proposed in the literature. They are based on redundancy which implies the addition of some functionalities that are not needed for carrying out the jobs the user demands from the system [9]. Redundancy is not used to implement the normal system operations that are supposed to perform, but it is used to guarantee that the system functions are performed correctly even if the errors exist. The system has to pay extra cost in any case of redundancy which uses extra resource.

There are various reliability techniques in the literature based on hardware or software redundancy in the system. Moreover, hybrid methods which merge hardware and software solutions provide more reliable and low-cost solutions.

Hardware-based reliability techniques rely on hardware redundancy which corresponds to the physical replication of the hardware components of a system. Three approaches for hardware redundancy implementation are the *static redundancy*, the *dynamic redundancy*, and the *hybrid redundancy* [10]. The aim of static or masking redundancy techniques is to mask the faults in the system by using a voting mechanism. The redundancy system, known as *Triple Modular Redundancy (TMR)*, has three identical versions of the system which are connected to a majority voter [11–14]. The system output is determined by using the output of the voter which relies on the majority of the versions. If one version fails, the system continues by using the other two fault-free modules' output. Although static redundancy provides tolerance to errors by fault masking, *dynamic redundancy* techniques deal with the problem by considering error detection, error location, and error recovery. The redundancy method, known as *standby sparing*, has one operating module and one or more spare modules. As soon as an error is detected and localized in the operating module, the operating module is replaced by one spare module to recover the error. On the other hand, the *hybrid redundancy* combines static and dynamic redundancy techniques. It uses the fault masking to prevent the system to produce erroneous output (static) and error detection, location, recovery to restore the erroneous module to a fault-free state (dynamic).

Fault-tolerant processor architectures have been proposed in the literature in order to deal with soft errors. One of the hardware-based solutions, a dynamic implementation verification architecture (DIVA), utilizes dual execution units in order to increase the reliability of the system [15]. DIVA uses the speculation mechanism of the processors for detecting errors in the processor core, where it consists of two main components including DIVA core and DIVA checker. DIVA core is composed of all microprocessor units including fetch, decode, execute stages except retirement stage. It stores the speculative results in the re-order buffer after execution. The instructions, input operands and the results are sent to DIVA checker. The checker verifies the correctness of the results and sends the verified results to the commit phase. If any errors are detected by the checker, it corrects the computation and restarts the processor at the next instruction. It is a hardware solution which adds DIVA checker architecture to the core processor.

Simultaneous and Redundantly Threaded (SRT) processor [16] and Redundant Multithreading (RMT) [17] are two architectures based on Simultaneous Multithreading (SMT) that provides fault-tolerance by running two identical copies of the same program as independent threads and comparing their outputs. An enhanced store queue is proposed for output comparison of the threads. After stores from the leading thread are appended to the store buffer, trailing thread stores enter the same store buffer and they are compared to the corresponding store operation for error detection.

DMR (Dual Modular Redundancy), which is the duplication of a hardware structure and decides the fault by voting the results of the duplicated units, has been used for two cores of chip multiprocessors by switching the DMR-on and the DMR-off modes in order to balance performance-reliability trade-off [18]. The applications requiring high reliability are executed by turning on the DMR mode, while the applications requiring high performance are executed by turning off DMR in order to avoid DMR penalty.

In order to improve system reliability of multicore architectures, an asymmetric multicore architecture, which consists of cores with different reliabilities, has been proposed [19]. The architecture has multiple cores, which have identical Instruction

Set Architecture (ISA), and the same speed and performance but different fault tolerant hardware and properties. The software processes, which are classified as critical, run on more reliable cores, while non-critical processes, which requires less reliability, execute on the cores with smaller fault-tolerance.

On the other hand, *software-based redundancy* approaches propose the replication of the application code by adding additional instructions to the original program to implement software information and time redundancy [9]. Information redundancy adds redundant information to data for error detection, masking, and correction. For instance, single-bit parity code adds one extra bit to a binary data in order to detect error on the data by checking the number of 1s in the data. Time redundancy performs the same operation at different times and compares the results to detect the error. Since the techniques require no hardware addition or modification, they come free of cost.

EDDI is a software-based fault detection mechanism which duplicates the instructions and uses redundant execution to achieve fault tolerance [20]. The instructions are duplicated by the compiler and executed through the original program flow. Each copy uses different registers and memory locations. At the synchronization points, extra check instructions are added in order to check whether the results of the original instructions and the added ones are the same.

SWIFT is another technique that proposes a compiler-based transformation [21]. It recompiles the source files by replicating the instructions and adding comparison points, in order to detect the errors in the execution as in EDDI [20]. In addition to EDDI, it eliminates memory penalty by using enhanced control-flow checking mechanism. It restricts the control-flow checking only to blocks which have stores in them.

While hardware-based fault-tolerance techniques provide almost-perfect fault coverage with low performance degradation, they are not suitable for the cases restricted by high hardware cost. Although software-based techniques are more reasonable with zero hardware cost, they result in performance degradation due to extra software. Moreover, software solutions are not capable of directly examining microarchitectural components of the system. Therefore *hybrid approaches*, which combine hardware-based and software-implemented techniques, have been proposed.

In the literature, there are three hybrid techniques that are called as CompileR-Assisted Fault Tolerance (CRAFT), which combine the software-only technique SWIFT with hardware modifications from RMT by achieving nearly perfect reliability, low performance degradation and low hardware cost [22]. The first technique, CRAFT:Checking Store Buffer (CSB), duplicates store instructions in the same way that it duplicates all other instructions. The compiler schedules store instructions, which are tagged as original or duplicate, so that the duplicate stores happen in the same order as the original stores. When the original and duplicated versions' store values and addresses are the same, it is validated. The code is run on CSB which is normal store buffer except it does not commit entries to the memory until they are validated. The second technique, CRAFT:Load Value Queue (LVQ), considers vulnerability between address validation and address consumption as well as vulnerability between load instruction and value duplication. Redundant load execution is achieved by the LVQ structure by accessing memory for only original load instruction and bypassing the load value for the duplicated load from the structure. On the other hand, the last one, CRAFT:CSB+LVQ, duplicates both store and load instructions for the fault-tolerance of the system.

To quantify the system vulnerability to soft errors and evaluate the efficiency of fault-tolerance techniques, a metric of vulnerability comparison is required. Architectural Vulnerability Factor (AVF) has been defined as the probability of an error in the program output which results from a fault in a hardware structure [23]. AVF looks at the vulnerability problem from an architectural perspective, but it does not quantify the program level vulnerability to soft errors. While AVF is a hardware-based metric, Program Vulnerability Factor (PVF) quantifies program codes' vulnerability [24]. Although PVF looks at the vulnerability from software perspective, it is for single threaded applications and does not work directly for multithreaded programs. Given emerging trends toward multicore architectures and their ensembles, one needs a new metric to quantify vulnerability of multithreaded applications.

1.2. Thesis Contributions

This thesis explores the soft error reliability analysis of parallel applications running on multicore architectures. We can summarize the main contributions of this thesis as follows:

- We propose and evaluate Thread Vulnerability Factor (TVF), a reliability metric for quantifying relative vulnerability of multithreaded applications. We calculate the local and remote TVF values using a set of multithreaded applications from the PARSEC and SPLASH-2 benchmark suites. We also perform a validation study to compare our metric values and SDC (Silent Data Corruption) rates of fault-injection experiment results.
- We perform a performance-reliability tradeoff analysis of different multithreaded applications running on multicore architectures. We calculate TVF values and gather execution clock cycles on different versions of a set of parallel programs. We discuss the effect of design choices on the system performance and reliability by comparing different implementations. By considering tradeoff between these two concerns, we observe that design choice becomes clear for some of the applications which provide different vulnerability values with almost equal performance.
- We propose and evaluate *reliability-aware* core partitioning schemes for multicore architectures. Our schemes consider the reliability of the system which has a performance bound to satisfy the quality of service. The goal of our reliability-oriented core partitioning scheme is to maximize the reliability of the system while allocating available cores for the multithreaded applications. Another scheme that we propose considers both the system performance and reliability, and partitions the residual cores (i.e., the cores remaining after satisfying the performance bounds) to maximize the value of the combined metric defined as Vulnerability-Delay Product (VDP). We perform a simulation study with various workloads consisting of multiple multithreaded applications to evaluate our proposed partitioning schemes.
- We present a reliability assessment tool for multithreaded applications which takes into account user preferences. Our tool evaluates the target application for

the most vulnerable threads or thread regions, then it recommends the most efficient way for the replicated execution. We propose a critical thread identification algorithm to evaluate the most critical thread in a parallel application for the redundancy. We extend our analysis to eliminate the thread regions that do not contribute to the system vulnerability. We validate the effectiveness of our tool by performing fault injection based analysis.

1.3. Outline of the Thesis

The remainder of the thesis is organized as follows: Chapter 2 presents definition and implementation details of our vulnerability metric including motivating examples. The validation study for TVF metric is presented in Chapter 3. Chapter 4 and Chapter 5 demonstrate the utilization of our vulnerability metric to analyze performance-reliability tradeoff for multithreaded applications and perform reliability-aware core partitioning on multicore architectures. Thread-level partial fault tolerance scheme based on critical thread analysis is presented in Chapter 6. We conclude the thesis in Chapter 7.

2. THREAD VULNERABILITY FACTOR

In this chapter, we introduce a novel metric, Thread Vulnerability Factor (TVF), for quantifying the relative vulnerability of multithreaded applications to soft errors on multicore architectures. Since TVF comprises vulnerability values resulted from the dependency of the threads, our metric presents more effective and detailed information on vulnerability of applications.

As part of this chapter, a set of vulnerability metrics presented in the literature and their limitations are given in Section 2.1. The definition and implementation details of our metric, TVF, is presented in Section 2.2. Finally, Section 2.3 and Section 2.4 presents the details of experimental setup and results of performance evaluation to demonstrate the effectiveness of our metric, respectively.

2.1. Vulnerability Metrics in the Literature

In this part, we summarize the details of four vulnerability metrics presented in the literature, which are Architectural Vulnerability Factor (AVF) [23], Register Vulnerability Factor (RVF) [25], Instruction Vulnerability Factor (IVF) [26], and Program Vulnerability Factor (PVF) [24].

2.1.1. Architectural Vulnerability Factor (AVF)

Architectural Vulnerability Factor (AVF) has been defined as the probability that a fault in a processor structure will result in an error in the program output [23]. While some transient errors in the processor or memory structures' bits cause incorrect execution of the instructions, they do not affect the program output. On the other hand, some bits which have been called Architecturally Correct Execution (ACE) bits affect the program execution and output. ACE bits have been used in the calculation of AVF of a hardware structure. These bits can be classified as architectural and microarchitectural bits.

In the ACE analysis, all bits have been considered as ACE-bit unless it is figured out otherwise and un-ACE bits, which have no effect on the program output, have been identified at both architectural and microarchitectural levels. Microarchitectural un-ACE bits are processor state bits that do not affect the committed instruction path, whereas architectural un-ACE bits are hardware-independent bits that affect the instruction execution path but do not change the program output. Microarchitectural un-ACE bits are classified into four situations:

(i) Idle or Invalid State. When a data or status bit is idle or does not contain any valid information, the error on the bit does not affect the output of the program.

(ii) Mis-speculated State. When speculated operations such as branch prediction and speculative memory disambiguation are incorrectly speculated, they do not have any effect on the execution.

(iii) Predictor Structures. An error on predictor structures such as branch predictors, jump predictors, return stack predictors, store-load dependence predictors causes only wrong prediction, it does not have any effect on the program execution.

(iv) Ex-ACE State. ACE bits, which have become un-ACE bits after their last use, do not affect the execution.

On the other hand, architectural un-ACE bits are classified into five situations:

(i) NOP Instructions. Many instruction sets implement NOP instructions that do not affect the state of the processor. These instructions are used to align instructions to address boundaries or to fill instruction templates.

(ii) Performance-Enhancing Instructions. An error in performance-enhancing instructions only causes the performance not to be improved.

(iii) Predicated-False Instructions. Predicated instruction-set architectures allow instruction execution based on a predicate register that means if the register is true, the instruction commits. All bits in the predicated-false instructions except predicate register are un-ACE bits.

(iv) Dynamically Dead Instructions. The instructions whose results are not used by any other instructions are dynamically dead instructions.

(v) Logical Masking. Some bits that belong to operands in an instruction do not affect the output of the program.

After ACE analysis, AVF value of a hardware structure is calculated by the division of the average number of ACE bits in one cycle by the total number of bits in the hardware structure (Equation 2.1) [23].

$$AVF = \frac{total\ residency\ of\ all\ ACE\ bits\ in\ a\ structure\ (in\ cycles)}{total\ number\ of\ bits\ in\ the\ hardware\ structure\ \times\ total\ execution\ cycles} \tag{2.1}$$

ACE analysis has been used to evaluate soft error rates of target designs [27] as well as dynamic estimation of vulnerability for soft errors [28, 29].

While AVF looks at the vulnerability problem from an architectural perspective, it does not quantify the program level vulnerability to soft errors. Therefore, AVF is not appropriate vulnerability metric for a software-centric analysis which needs a hardware-independent metric to measure the soft error vulnerability of a program.

2.1.2. Register Vulnerability Factor (RVF)

The register file susceptibility to soft errors has been studied; consequently, Register Vulnerability Factor (RVF) has been defined as the probability that a soft error in registers can be propagated to other system components [25]. While AVF is interested in the effect of soft error propagation, RVF focuses on the probability of soft error propagation to other hardware elements. The concept of AVF, which provides an estimation of soft error rate for hardware components by considering ACE bits affecting the final program output, can be used for register files. However, the soft errors in the register file can be overlapped by the write operations to the register file which leads AVF analysis insufficient.

In general, a value is first written into a register, then it is read by one or more times and finally another value is written into the same register, which terminates the lifetime of the old value and begins the lifetime of the new value. The access to the register file can be divided into four patterns, namely intervals including write-read (W-R), read-read (R-R), read-write (R-W), and write-write (W-W). The register file is vulnerable in the W-R and R-R intervals. The soft errors occurred during R-W and W-W intervals can be overlapped by the last write operation. RVF has been defined with the following equation [25]:

$$RVF = \frac{\sum_{i=1}^{n} SusceptibleTime(RV_i)}{\sum_{i=1}^{n} Lifetime(RV_i)}, \tag{2.2}$$

where n represents the number of register values, RV_i represents any register value, $SusceptibleTime(RV_i)$ represents the time intervals that RV_i is exposed to the susceptible intervals (i.e., W-R and R-R intervals for RV_i), and $Lifetime(RV_i)$ represents the lifetime of RV_i, which is the interval between the time that register is allocated and overlapped by another value.

Compiler-guided techniques based on instruction re-scheduling and reliability-oriented register assignment have also been proposed to improve register file reliability in terms of Register Vulnerability Factor metric [25]. One technique schedules the register read-write operations by delaying the write operations as late as possible and schedules the read operations as early as possible which causes shortening in the vulnerable W-R and R-R intervals and thus reduces RVF values. Another technique modifies the register allocation algorithm by distinguishing the registers with Error-Correcting Code (ECC) and the registers without ECC, by assuming that some registers are protected by ECC codes. If the registers with the highest RVF values are not protected by ECC, the compiler re-assigns the registers so that the registers with ECC have the highest RVF values. The reliability of the register file can be improved since the most vulnerable registers are protected by a hardware protection method.

Similarly, to measure the cache susceptibility to soft errors accurately and quantitatively, Cache Vulnerability Factor (CVF) has been defined [30,31]. Both RVF and CVF values are calculated by considering susceptible times of register values and cache blocks respectively. [32] has also proposed a static analysis method to estimate program vulnerability in caches.

2.1.3. Instruction Vulnerability Factor (IVF)

Instruction Vulnerability Factor (IVF) has been defined to evaluate how faults in every instruction affect the final application output [26]. Although IVF is similar to AVF, it measures how much of the final output is corrupted due to faults in every instruction instead of faults in a hardware structure. IVF estimation requires monitoring how different faults in every instruction propagate to the final application output. This can be accomplished by fault-injection experiments and offline-profiling. Since injection of all the possible faults into every executed instruction would require too large number of experiments, random fault injection into every instruction has been preferred in the IVF estimation process. The final application output has been compared to the correct one and the output corruption has been measured. These measurements have been performed offline and the results have been saved in a program binary as a separate table, which maps application instructions to the IVF values. These IVF values have been used to determine protection levels for each instruction by loading into a special hardware buffer.

Instruction-Level Fault Tolerance Configurability (ILCOFT) hardware-based technique [33] has been supported by IVF estimation process. In ILCOFT, the programmer is able to specify which instructions are critical, and should be highly fault tolerant and which instructions are not. The application instructions have been protected (duplicated for redundancy) according to this specification. The criticality of the instructions can be determined and ILCOFT can be automatized by using IVF estimation. IVF-based selective instruction duplication for fault tolerance has resulted in performance improvements over full instruction duplication by small fault coverage.

2.1.4. Program Vulnerability Factor (PVF)

To define the vulnerability of a software to transient errors, Program Vulnerability Factor (PVF) has been proposed in the literature [2, 24]. This metric measures the vulnerability of a program to the hardware faults in a microarchitecture independent way and provides the relative vulnerability of the programs to be able to make decisions about the reliability of these programs. Moreover, instead of applying full redundancy to protect all application, it is possible to protect the most vulnerable part of a program, which is evaluated by PVF analysis, that reduces cost for the reliability.

While calculating PVF, microarchitectural resources that are hardware dependent are not considered and only architectural resources are taken into account by the software's view. As a processor consists of many hardware structures, a program is composed of several architectural resources. For instance, an arithmetic addition operation is visible to a program in architectural level. However, in the microarchitectural level; this operation is implemented by more complex micro operations.

PVF is calculated for each architectural resource including register file, load/store queue, and ALU units by division of instructions, which have effect on the output by total number of instructions. The average PVF of a program is obtained by combining them and weighting appropriately for their size.

PVF can be defined as a component of AVF as in the following representation [2]:

$$AVF_H = \frac{\sum_{n=1}^{N} ACE\ m\text{-}bits\ in\ H\ at\ cycle\ n}{B_H \times N}, \tag{2.3}$$

$$PVF_R = \frac{\sum_{i=1}^{I} ACE\ a\text{-}bits\ in\ R\ at\ instruction\ i}{B_R \times I}, \tag{2.4}$$

where AVF_H is the AVF of the hardware structure H with size B_H over a period of N cycles, and PVF_R is the PVF of architectural resource R with size B_R over I instructions.

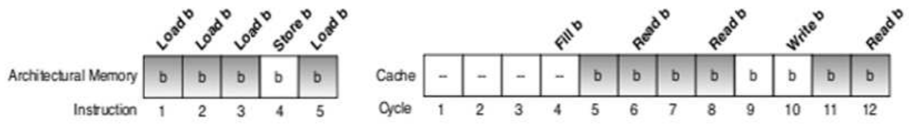


Figure 2.1. A set of program instructions and the resulting hardware operations [2].

Figure 2.1 shows a series of operations to byte *b* in architectural memory. In the left, the operations are represented by the programs's point of view and the PVF of *b* is calculated as follows:

$$PVF_b = \frac{\sum_{i=1}^{I} ACE\ a\text{-}bits\ in\ R\ at\ instruction\ i}{B_R \times I}$$

$$PVF_b = \frac{(8 + 8 + 8 + 0 + 8)}{8 \times 5} = 80\%$$

AVF is calculated as following by using conversion:

$$AVF_H = \frac{\sum_{i=1}^{I} ACE\ a\text{-}bits\ at\ instruction\ i \times m_i \times n_i}{B_R \times I \times M_H \times N_I}$$

$$AVF_H = \frac{(8 \times 0 \times 4) + (8 \times 1 \times 2) + (8 \times 1 \times 2) + 0 + (8 \times 1 \times 2)}{8 \times 5 \times \frac{8}{8} \times \frac{12}{5}} = 50\%$$

Although PVF is a software-based metric, it is useful for single-thread programs and does not work directly for multithreaded programs. Given emerging trends toward multicore architectures and their ensembles, one needs a new metric to quantify vulnerability of multithreaded applications.

2.2. Thread Vulnerability Factor (TVF)

In this section, we introduce our proposed metric, Thread Vulnerability Factor (TVF) [34, 35]. To provide better understanding, we first describe data sharing mechanism between threads in a multithreaded application, then illustrate TVF using examples.

In shared memory systems, threads of a given multithreaded application can read from and write to shared memory [36]. Figure 2.2 illustrates a data sharing scenario (write/read order representation) between different threads in a multithreaded application running on a shared memory system. In this scenario, four threads running in parallel, share data elements along their execution. $Thread_2$ reads the value of variables (X) and (Z) that $Thread_1$ and $Thread_3$ have written, respectively. Therefore, we can say that the reader thread ($Thread_2$) is dependent on the writer threads ($Thread_1$ and $Thread_3$). Moreover, $Thread_4$ reads data (Q) which $Thread_3$ has written. In this case, $Thread_4$ is dependent on $Thread_3$ as well as on $Thread_2$ and $Thread_1$. Another data dependency between threads results from variable (Y), which is written by $Thread_2$ and read by $Thread_3$. In this case, $Thread_3$ is dependent $Thread_2$ and $Thread_1$ which has previously written a value (X) read by $Thread_2$. In the following subsections, we illustrate our metric called Thread Vulnerability Factor (TVF) using examples.

2.2.1. Formal Definition of Thread Vulnerability Factor

TVF measures the vulnerability of a thread to hardware faults. Since threads may be dependent on each other in a multithreaded application, the vulnerability of one thread is also dependent on that of other threads that share data with that thread. Therefore, we also have to consider these threads to calculate the TVF of one thread. TVF of thread i can be calculated as follows:

$$TVF(T_i) = [\, w_L \times LVF(T_i)\,] + [\, w_R \times RVF(T_i)\,], \tag{2.5}$$

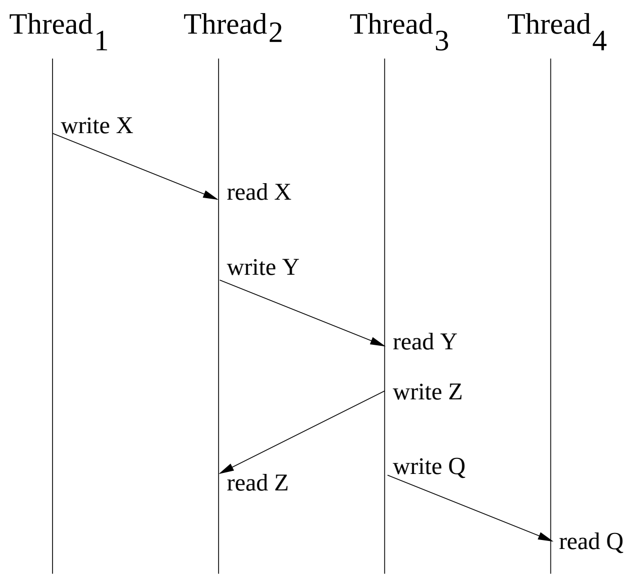


Figure 2.2. Data sharing between four threads.

where $LVF(T_i)$ is the local vulnerability factor for thread i and $RVF(T_i)$ is the remote vulnerability factor for thread i which iterates over all threads that sends data to thread i. w_L and w_R are weight values for local and remote vulnerability factors, respectively. The weight values provide the analysis of different cases that needs to consider the weights of local and remote terms, and may get values between 0 and 1. We use equal local and remote vulnerability weights (0.5), unless otherwise stated. A sensitivity analysis is conducted to examine the effect of different weights (see Section 2.4.2).

Remark 2.1. *Since both $LVF(T_i)$ and $RVF(T_i)$ terms can only take values between 0 and 1 (as explained in the following subsections) and these terms are multiplied by weight values (positive values less than 1) in TVF calculation, TVF of a thread (i.e., $TVF(T_i)$ term) can only have values between 0 and 1.*

Remark 2.2. *While TVF considers the vulnerability of threads that have data sharing with target thread, TVF value of one thread which has no remote memory access (i.e., $RVF = 0$) simply equals to Local Vulnerability Factor (LVF) value.*

Remark 2.3. *We consider three hardware components in our TVF evaluation including register file, ALU unit, and memory (cache), which are evaluated separately and denoted as TVF_{RF}, TVF_{ALU} and TVF_{mem} in the equations, respectively.*

2.2.1.1. Local Vulnerability Factor (LVF). The local vulnerability of the thread (LVF) is calculated in a similar fashion given for PVF [24], where the vulnerability values of the software resources in the thread are combined and normalized by considering their vulnerable intervals. The resources that we consider are registers, ALU (addition, subtraction etc.), and memory (load/store) operations.

- *Vulnerability of Registers:* The vulnerability of register resources is defined between Write-Read and Read-Read operations on these registers [24]. A register is vulnerable during the lifetime of a value that starts with a write operation on that register and ends with the last read operation. As soon as another write operation on the register occurs, its value is updated and different lifetime is activated. The hardware faults between these lifetime intervals may affect the register. LVF of a register R for thread i can be calculated as follows:

$$LVF_R(T_i) = \frac{\sum_{j=1}^{n} Vinterval_j}{Tinstruction}, \tag{2.6}$$

  where $Vinterval$ represents the number of instructions in the interval that the register R is vulnerable, n represents the number of vulnerable intervals for register R, and $Tinstruction$ represents the total number of instructions in the application. We may consider the following code fragment as an example:

$$\begin{aligned}
1 &: \quad ld\ r1 = [r4] \\
2 &: \quad ld\ r2 = [r5] \\
3 &: \quad ld\ r3 = [r6] \\
4 &: \quad add\ r1 = r1, r2 \\
5 &: \quad add\ r1 = r1, r3 \\
6 &: \quad st[500] = r1
\end{aligned}$$

  In the code above, $r1$ is written by the first instruction by loading a value from memory location. It is both read and then written by the fourth and fifth instructions; and it is read by the sixth instruction, which stores the calculated value to the memory location. Therefore, $r1$ is vulnerable between instructions 1 and 4, instructions 4 and 5, and instructions 5 and 6. The LVF value of $r1$ register in the code segment can be calculated as:

$$LVF_{r1} = [(4-1)+(5-4)+(6-5)]/6 = 5/6.$$

To calculate the vulnerability factor of the thread with respect to all register resources, LVF value for registers can be averaged by dividing the sum of LVF values into the total number of registers [24]. Since it is possible to use different number of registers on the different code segments, it is essential to evaluate an average vulnerability value.

- *Vulnerability of ALU Operations:* We consider ALU resource as the architecturally-visible arithmetic-logic operations (i.e., addition, subtraction, and, xor etc.) in the code segment. The vulnerability is obtained by dividing the number of ALU operations by the total number of instructions [2]. For instance, LVF of ALU operations (LVF_{ALU}) in the code fragment below is equal to 2/5 (i.e., number of arithmetic operations divided by the total number of instructions).

$$\begin{aligned}
1:&\quad ld\ r3 = [100]\\
2:&\quad ld\ r2 = [200]\\
3:&\quad add\ r3 = r3, r2\\
4:&\quad add\ r3 = r3, r2\\
5:&\quad st[300] = r3
\end{aligned}$$

Since our goal is to look at the vulnerability from compiler perspective, we do not distinguish between different ALUs, as they are not visible to the compiler. The compiler just sees an ADD, it does not know which ALU will execute it at runtime. However, individual (logical) registers such as $r1$, $r2$ are visible to the compiler. Therefore, our criterion is compiler (software) visibility.

- *Vulnerability of Memory Operations:* We assume a multi-level cache structure and our baseline architecture has one L2 cache (protected) shared by all processor cores, and L1 private cache for each core. A soft error may hit data residing in private L1 cache structure. Additionally, main-memory and L2 cache units are assumed to be well-protected (e.g., using some sort of ECC). Therefore, a memory error means that data catches a soft error while in L1 data cache. We also assume that the cache write-policy in the architecture is write-through. That is, the cache locations are only vulnerable between write and read operations as in the register case. If a soft error hits the cache block which is evicted before a read operation, the error does not affect the vulnerability of the program since the correct value in the higher level of storage will be used in the subsequent read operations. Therefore, we do not have to consider the data in the cache which is evicted or replaced by any other block before it is used. Whenever a memory read operation (*load*) occurs, the write-read interval is considered as vulnerable unless it is a miss in the L1 cache. The vulnerability is calculated for only L1 hit read operations.

If we consider the following code fragment executed by single thread, it is clear that location X in L1 is vulnerable between the instructions 1 and 4 assuming that *load X* operation is a hit in L1 cache structure.:

$$\begin{array}{ll} 1: & st[X] = r1 \\ 2: & .... \\ 3: & .... \\ 4: & ld\ r2 = [X] \end{array}$$

The LVF value of memory location X in the code segment can be calculated as:

$$LVF_X = (4-1)/4 = 3/4.$$

The situation is similar when threads share data, i.e., store ($st[X] = r1$) is executed by one thread (in Core 1) and load ($ld\ r2 = [X]$) is executed by another thread (in Core 2). In this case, the flow is follows: Thread 1 stores $r1$ in X, X

is in both L1 of Core 1 and shared L2; thus X is vulnerable in Thread 1 (Core 1)'s L1 cache, that is, an error may hit the updated X while in L1. Thread 2 executes load ($ld\ r2 = [X]$), and (updated and possibly erroneous) value in X is transferred to L1 of Thread 2 (via shared L2). Memory vulnerability of X memory location is between all instructions (of Thread 1) executed between Store (in Thread 1) and Load (in Thread 2). Once X is loaded to $r2$ (of Thread 2), any vulnerability of it is the register vulnerability of $r2$ (of Thread 2).

We measure the vulnerable intervals of each memory location accessed by the application and calculate the vulnerability values for these locations. To calculate the vulnerability factor of the entire thread with respect to all memory operations, LVF value of a cache structure can be averaged by dividing the sum of LVF values into the total number of memory locations accessed by the target thread.

Since LVF values are averaged by dividing into the number of resources (registers or memory locations), LVF of a thread (i.e., $LVF(T_i)$) can only take values between 0 and 1.

2.2.1.2. Remote Vulnerability Factor (RVF). RVF represents the vulnerability impact of the threads that interact with the target thread in a multithreaded application. If one thread does not read data written by any other thread previously, its vulnerability becomes LVF term which only considers the code of the target thread itself. Since threads generally have interactions in a multithreaded application, the definition of RVF term, which also considers the code of the other threads, is essential. RVF of thread i can be calculated as follows:

$$RVF(T_i) = \frac{\sum_{j=1}^{n} rVF(T_i, T_j)}{RV}, \tag{2.7}$$

where $rVF(T_i, T_j)$ represents the remote vulnerability factor for thread i induced by thread j which sends data to thread i, and n is the number of threads that sends data to thread i. $RVF(T_i)$ iterates over all these n threads. This is because these threads can pass a corrupted value to thread i. The RV parameter represents the total number of remote memory accesses on thread i. To normalize RVF value, total value is divided by the RV parameter. Due to this normalization operation, RVF of a thread (i.e., $RVF(T_i)$) can only take values between 0 and 1, similarly to $LVF(T_i)$ values.

When a thread reads a data (from shared-memory) that is written by another thread, TVF of the writer thread becomes RVF of the reader thread. Since a soft error affecting the reader thread may propagate to the writer thread (via remote memory operation), the total vulnerability of the writer thread is counted for the remote vulnerability of the reader thread as well. That is, the vulnerability of the writer thread is double-counted. The $rVF(T_i, T_j)$ term in Equation 2.7 can be calculated by considering total TVF value of thread j at the end of the instruction which it stores the value read by thread i.

Since the vulnerability value of the remote thread at the end of the store instruction defines the remote vulnerability factor of the target thread, the schedule of the store operation becomes as important as the store operation itself. It is possible that there can be many interactions among threads of an application, where these interactions do not bring much vulnerability due to remote threads' local behavior before the store operations. On the other hand, an application with smaller number of interactions but larger local vulnerability value (before the remote write operation) may generate larger remote vulnerability factor value. In that case, modeling shared data accesses of threads may not be enough to determine the vulnerability of the application. Therefore, it is essential to consider RVF analysis as well as LVF analysis.

Figure 2.3 represents data sharing between the threads in a multithreaded application, where TVF value of each thread can be calculated as follows:

$$\begin{aligned} TVF(T_1) &= LVF(T_1) \\ &= LVF(X+Z). \end{aligned}$$

$$TVF(T_2) = [\, w_L \times LVF(T_2)\,] + [\, w_R \times RVF(T_2)\,]$$

$$
\begin{aligned}
&= [\,w_L \times LVF(T_2)\,] + [\,w_R \times rVF(T_2, T_1)\,] \\
&= [\,w_L \times LVF(Y + Q + P)\,] + [\,w_R \times LVF(X)\,].
\end{aligned}
$$

$$
\begin{aligned}
TVF(T_3) &= [\,w_L \times LVF(T_3)\,] + [\,w_R \times RVF(T_3)\,] \\
&= [\,w_L \times LVF(T_3)\,] + [\,w_R \times rVF(T_3, T_2)\,] \\
&= [\,w_L \times LVF(T_3)\,] + \\
&\quad [\,w_R \times (w_L \times LVF(T_2) + w_R \times rVF(T_2, T_1))\,] \\
&= [\,w_L \times LVF(R + S)\,] + \\
&\quad [\,w_R \times (w_L \times LVF(Y + Q) + w_R \times LVF(X))\,].
\end{aligned}
$$

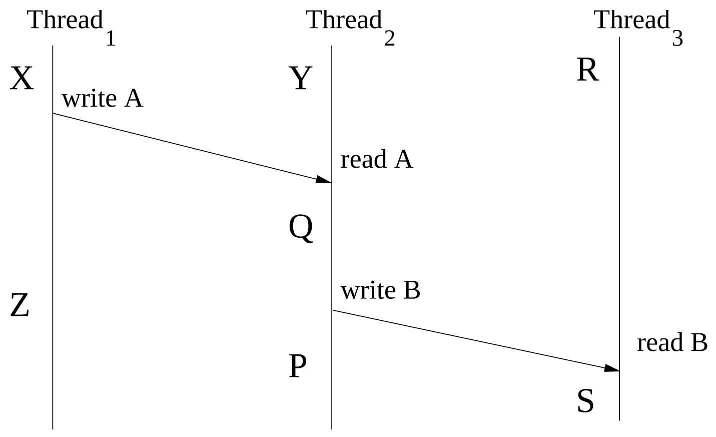


Figure 2.3. An example for representing data sharing between threads.

To calculate TVF of a thread, the vulnerability of each thread on which that thread is dependent should be considered in order to capture the effect of these threads on the vulnerability of the thread investigated. Since $Thread_1$ is not dependent on other threads, its TVF is calculated using only its instructions. However, the portion of $Thread_1$ should also be considered to obtain TVF of $Thread_2$. Since $Thread_2$ reads data written by $Thread_1$ at the end of the code fragment shown as X (in Figure 2.3), the vulnerability of X affects the vulnerability of $Thread_2$. Similarly, to calculate TVF of $Thread_3$, code portions of both $Thread_1$ and $Thread_2$, which affect the vulnerability of $Thread_3$, should be considered.

Table 2.1. Memory location access of two threads.

$Thread_1$		$Thread_2$	
100:	store X	50:	...
...	...	...	...
...	...	200:	load X
250:	load X	...	...
...	...	300:	store X
...	...	...	...
...	...	400:	load X

Table 2.1 presents sample locations and corresponding memory operations that belong to $Thread_1$ (T_1) and $Thread_2$ (T_2) in the code segment. The memory location X is vulnerable on code segment $Thread_1$ as well as $Thread_2$. The value is written to the location by 100th instruction and read by 250th instruction in the code segment of $Thread_1$. The vulnerability factor (VF) of the location (at the end of the 250 instructions) is calculated as follows:

$$LVF_X(T_1) = (250 - 100)/250 = 0.60.$$

Since there is a store operation by $Thread_1$ before the load operation of $Thread_2$ in 200th instruction, the memory location of X is vulnerable both locally and remotely due to this load operation. X location is vulnerable in $Thread_2$ between 50 and 200 instructions locally (assuming that X is in the L1 cache), because the store operation by $Thread_1$ is executed in 50th instruction of $Thread_2$. Moreover, the location has remote vulnerability due to the store operation is performed by $Thread_2$ which is not the thread itself reading the value from memory address.

The vulnerability factor (VF) calculation of $Thread_2$ (at the end of the 400 instructions) due to load operation in 200th instruction of $Thread_2$ consists of two phases

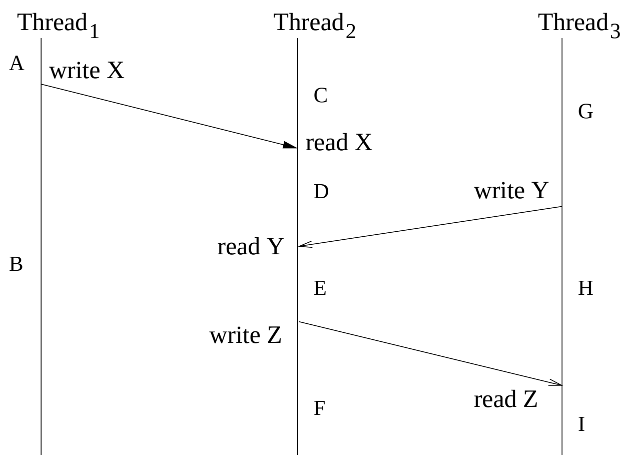


Figure 2.4. An example for more complex data sharing scenario.

including local memory VF and remote VF:

$$LVF_X(T_2) = (200 - 50)/400 = 0.375$$
$$RVF(T_2) = rVF(T_2, T_1)$$
$$total\ TVF\ value\ in\ the\ 100th\ instruction\ of\ T_1.$$

There is another load operation on X by $Thread_2$ in the 400th instruction. In this case, the last store operation is performed by the $Thread_2$ itself. Therefore, only local memory vulnerability of the location X is considered and calculated as follows:

$$LVF_X(T_2) = (400 - 300)/400 = 0.25.$$

To obtain total local memory VF value of X location at the end of the instructions given above, the local memory VF values are added for this memory location:

$$LVF_X(T_2) = 0.375 + 0.25 = 0.625.$$

The vulnerability of any threads having various data sharing characteristics can be calculated by using Thread Vulnerability Factor. Figure 2.4 represents more complex data sharing scenario between the threads in a multithreaded application. For this case, $Thread_1$ writes X value to the shared memory location at the end of the code segment A and $Thread_3$ writes Y value to the shared memory location at the end of the code segment G. Then these X and Y values are loaded by $Thread_2$ at the end of C and $C + D$ code segments, respectively. Another shared value is Z written by $Thread_2$ and then read by $Thread_3$. TVF value of each thread can be calculated as follows:

$$\begin{aligned}
TVF(T_1) &= LVF(T_1) \\
&= LVF(A+B).
\end{aligned}$$

$$\begin{aligned}
TVF(T_2) &= [\,w_L \times LVF(T_2)\,] + [\,w_R \times RVF(T_2)\,] \\
&= [\,w_L \times LVF(T_2)\,]+ \\
&\quad [\,w_R \times \frac{rVF(T_2,T_1) + rVF(T_2,T_3)}{2}\,] \\
&= [\,w_L \times LVF(C+D+E+F)\,]+ \\
&\quad [\,w_R \times \frac{LVF(A) + LVF(G)}{2}\,]. \\
TVF(T_3) &= [\,w_L \times LVF(T_3)\,] + [\,w_R \times RVF(T_3)\,] \\
&= [\,w_L \times LVF(T_3)\,] + [\,w_R \times rVF(T_3,T_2)\,] \\
&= [\,w_L \times LVF(T_3)\,]+ \\
&\quad [\,w_R \times (w_L \times LVF(T_2) + w_R \times RVF(T_2))\,] \\
&= [\,w_L \times LVF(G+H+I)\,]+ \\
&\quad [\,w_R \times (w_L \times LVF(C+D+E)+ \\
&\quad\; w_R \times \frac{LVF(A) + LVF(G)}{2})\,].
\end{aligned}$$

It is also valid for this scenario that the vulnerability of each thread on which that thread is dependent should be considered to capture the effect of these threads on the vulnerability of the thread investigated. Since $Thread_1$ is not dependent on other threads, its TVF is calculated using only its code segment. However, the portion of $Thread_1$ and $Thread_3$ should be also considered to obtain TVF of $Thread_2$. Since $Thread_2$ reads data written by $Thread_1$ at the end of the code fragment shown as A and data written by $Thread_3$ at the end of the code fragment shown as G (in Figure 2.4), the vulnerability of A and G affect the vulnerability of $Thread_2$. Similarly, to calculate TVF of $Thread_3$, code portion of $Thread_2$ which affects the vulnerability of $Thread_3$ should be considered. Since $Thread_2$ is affected by $Thread_1$ and $Thread_3$ before its write operation on Z value, the vulnerability values coming from these threads should be also considered.

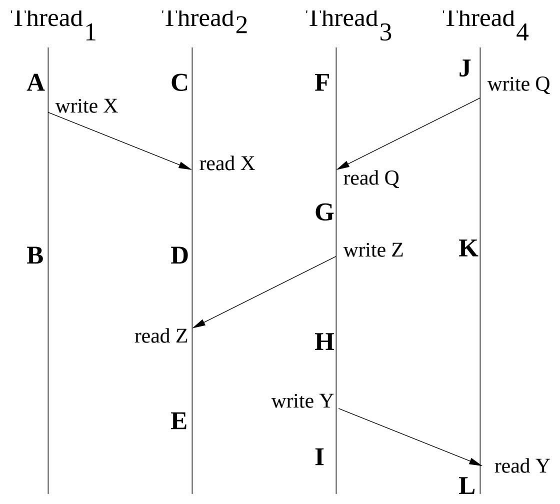


Figure 2.5. The communication of four threads in a multithreaded application.

Figure 2.5 demonstrates another example communication behavior of threads in a multithreaded application executing on a shared memory multicore architecture. Since $Thread_2$ reads data written by two other threads ($Thread_1$ and $Thread_3$), we can say that any soft error that hits $Thread_1$ or $Thread_3$ before their remote data write operations (write X and write Z) affects the execution of $Thread_2$. The vulnerability of $Thread_2$ is dependent on the code segments of these two threads. While local term of TVF metric considers the code segment of the $Thread_2$, remote term counts $Thread_1$ and $Thread_3$. Another memory operation pair, which causes communication between threads, is write and read operations on Q location. $Thread_3$ is similarly dependent on the code segment of $Thread_4$. Moreover, the vulnerability of $Thread_2$ is affected by this memory operation. If there is an error on $Thread_4$ before its write operation, it influences $Thread_3$ directly and $Thread_2$ indirectly. Therefore, the TVF value for $Thread_2$ is calculated by considering the code segment of $Thread_4$ as well. Moreover, the write operation on Y memory location affects the vulnerability of $Thread_4$. The remote vulnerability from the writer thread $Thread_3$ includes both the local vulnerability and the remote vulnerability which results from earlier write operation by $Thread_4$.

TVF value of each thread in Figure 2.5 can be calculated as follows:

$$\begin{aligned}
TVF(T_1) &= LVF(T_1)\\
&= LVF(A+B).\\
TVF(T_2) &= [\,w_L \times LVF(T_2)\,] + [\,w_R \times RVF(T_2)\,]\\
&= [\,w_L \times LVF(T_2)\,] + [\,w_R \times \frac{rVF(T_2,T_1) + rVF(T_2,T_3)}{2}\,]\\
&= [\,w_L \times LVF(C+D+E)\,]+\\
&[\,w_R \times \frac{LVF(A) + [\,w_L \times LVF(F+G) + w_R \times LVF(J)\,]}{2}\,].\\
TVF(T_3) &= [\,w_L \times LVF(T_3)\,] + [\,w_R \times RVF(T_3)\,]\\
&= [\,w_L \times LVF(T_3)\,] + [\,w_R \times rVF(T_3,T_4)\,]\\
&= [\,w_L \times LVF(F+G+H+I)\,] + [\,w_R \times LVF(J)\,].\\
TVF(T_4) &= [\,w_L \times LVF(T_4)\,] + [\,w_R \times RVF(T_4)\,]\\
&= [\,w_L \times LVF(T_4)\,] + [\,w_R \times rVF(T_4,T_3)\,]\\
&= [\,w_L \times LVF(J+K+L)\,]+\\
&[\,w_R \times [\,w_L \times LVF(F+G+H) + w_R \times LVF(J)\,]].
\end{aligned}$$

2.2.2. An Example for Calculating TVF Values of Multiple Threads

In this section, we go through a detailed example to demonstrate how the TVF values of threads in a multithreaded application are obtained. The example deals with register vulnerability factors; and the threads involved are given in Table 2.2. There are three threads which are dependent on each other by data sharing in the example above. The second thread reads the memory address [500], which the first thread has written and the third thread reads the memory address [200], which the second thread has written. The data flow between the threads are illustrated in Figure 2.6. It is

Table 2.2. A sample code for TVF calculation.

	$Thread_1$	$Thread_2$	$Thread_3$
1:	ld r1 = [r2]	ld r1 = [r2]	ld r3 = [r2]
2:	ld r3 = [r2]	ld r3 = [r2]	ld r4 = [r2]
3:	ld r4 = [r2]	ld r4 = [500]	add r3 = r3, r4
4:	ld r5 = [r2]	add r3 = r3, r4	ld r5 = [200]
5:	add r3 = r3, r4	sub r1 = r1, 1	add r5 = r5, r3
6:	add r5 = r5, r1	st[200] = r1	
7:	st[500] = r3		
8:	sub r1 = r1, 1		

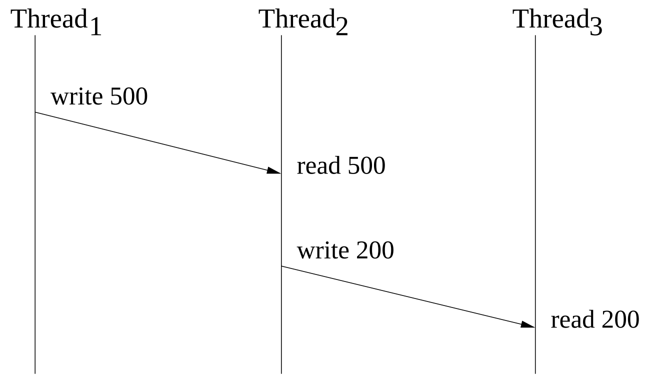


Figure 2.6. Data flow among three threads for the sample code given in Table 2.2.

assumed that the store operation to memory location [500] by $Thread_1$ is performed before the load operation at $Thread_2$ and the store operation to memory location [200] is performed before the load operation at $Thread_3$.

The vulnerability of the First Thread. The TVF value of $Thread_1$ which has no dependent threads can be calculated as:

$$For\ r1\ resource: \ [(6-1)+(8-6)]/8 = 7/8$$

$$For\ r2\ resource: \ [(2-1)+(3-2)+(4-3)]/8 = 3/8$$

$$For\ r3\ resource: \ [(5-2)+(7-5)]/8 = 5/8$$

$$For\ r4\ resource: \ (5-3)/8 = 2/8$$

$$For\ r5\ resource: \ (6-4)/8 = 2/8.$$

The TVF of all (total 5) register resources for $Thread_1$ is:

$$\begin{aligned} TVF_{RF}(T_1) &= LVF_{RF}(T_1) \\ &= (19/8)/5 = 0.475. \end{aligned}$$

The vulnerability of the Second Thread. The TVF value of the $Thread_2$ which is dependent on the first thread is calculated by adding its LVF and TVF of the instructions of the first thread until it writes to the memory. The TVF value of the second thread itself can be calculated similarly and the vulnerability factor values equal to 5/6 for $r1$, 1/6 for $r2$, 2/6 for $r3$ and 1/6 for $r4$. The LVF of all (total 4) register resources is:

$$LVF_{RF}(T_2) = (3/2)/4 = 0.375.$$

The RVF value of the second thread from the dependent first thread (assuming the instructions of the first thread have finished with the seventh store instruction):

$$For\ r1\ resource: \ (6-1)/7 = 5/7$$

$$For\ r2\ resource: \ [(2-1)+(3-2)+(4-3)]/7 = 3/7$$

$$For\ r3\ resource: \ [(5-2)+(7-5)]/7 = 5/7$$

$$For\ r4\ resource: \ (5-3)/7 = 2/7$$

$$For\ r5\ resource: \ (6-4)/7 = 2/7.$$

The vulnerability factor value of all (total 5) register resources is:

$$rVF_{RF}(T_2, T_1) = (17/7)/5 = 0.485.$$

We have defined total TVF of one thread by weighted (equal weights in this case) sum of its local and remote VF values. Thus we have TVF of the second thread:

$$\begin{aligned} TVF_{RF}(T_2) &= [\,0.5 \times LVF_{RF}(T_2)\,] + [\,0.5 \times rVF_{RF}(T_2, T_1)\,] \\ &= (\,0.5 \times 0.375\,) + (\,0.5 \times 0.485\,) = 0.43. \end{aligned}$$

The vulnerability of the Third Thread. The TVF value of the $Thread_3$, which is dependent on the second thread and affected by the first thread, can be calculated by adding its LVF and TVF of the instructions of the second and first thread until they write the data to the memory. The TVF value of the third thread itself can be computed similarly and the vulnerability factor values equal to 1/5 for $r2$, 4/5 for $r3$, 1/5 for $r4$ and 1/5 for $r5$. The LVF of all (total 4) register resources is:

$$LVF_{RF}(T_3) = (7/5)/4 = 0.35.$$

Since the store instruction in the second thread is the last instruction, we need to obtain the entire TVF value of this thread itself. Further, we have already calculated the partial TVF of the first thread which writes to the memory address that the second thread reads. Thus, we have TVF of the third thread:

$$\begin{aligned} TVF_{RF}(T_3) &= [\,0.5 \times LVF_{RF}(T_3)\,] + [\,0.5 \times rVF_{RF}(T_3, T_2)\,] \\ &= [\,0.5 \times LVF_{RF}(T_3)\,] + \\ &\quad [\,0.5 \times (\,0.5 \times LVF_{RF}(T_2) + 0.5 \times rVF_{RF}(T_2, T_1\,)\,] \\ &= (\,0.5 \times 0.35\,) + (\,0.5 \times 0.43\,) = 0.39. \end{aligned}$$

2.3. Experimental Setup

In this section, we present the details of our simulation platform, which is followed by a brief information on benchmarks considered in our experimental evaluation given in Section 2.4.

2.3.1. Simulation Platform

To measure the thread vulnerability factor of multithreaded applications, we use the Simics toolset [3]. Simics has ability to simulate various complete operating systems including Linux, Solaris and Windows. It is fast enough to run the realistic workloads including scientific benchmarks and desktop applications. Microprocessor design, operating system development, fault injection studies and hardware design verification are possible activities to be studied by using Simics. It also provides a multi-processor simulation environment by simulating many different hardware and software platforms.

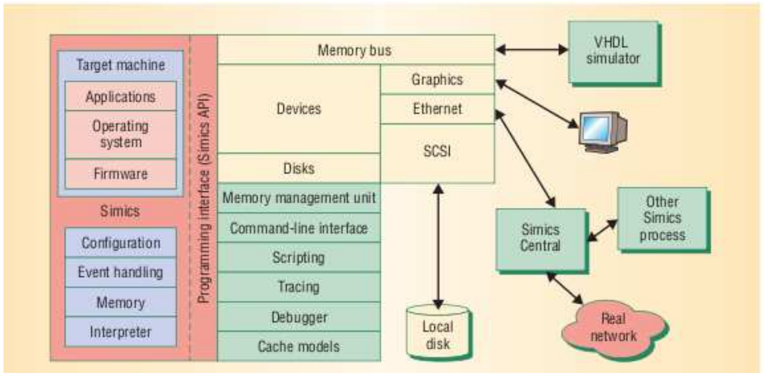


Figure 2.7. Simics architecture [3].

Figure 2.7 shows the overview of the architecture of the Simics. It simulates the target machine in the left which hosts an operating system and applications, this machine is configured by specifying its processor, memory hierarchy properties. Simics application programming interface (API) provides extensibility by allowing users to write new modules, to add new commands, or to modify existing modules. The command-line and graphical-user interfaces are the interface between the Simics and the user. The users can communicate with the Simics via the interfaces by providing commands, using tools such as tracing, debugging, scripting. Simics also have interface to the other simulators having model written in a hardware description language (HDL) such as Verilog.

We extend the *trace* module of Simics to track –at a thread level– different types of instructions including arithmetic and memory operations; and the vulnerability of each target resource (register, ALU, cache) is calculated during simulation using instruction and data traces. Since the architecture simulated in our experiments has X86 Pentium4 processors which implement X86 ISA [37], trace data provides assembly codes with X86 instructions. Our target multicore machine employs private L1 (data and instruction) caches and a shared L2 cache. Salient characteristics of the simulated multicore are given in Table 2.3. We use this configuration in our experiments throughout the thesis.

Table 2.3. Parameters of the simulated multicore architecture.

L1 data cache	16K/core 2-way cache
L1 cache latency	1 cycle
L2 shared cache size	4MB 4-way cache
L2 cache latency	10 cycles
Memory latency	200 cycles

We track hits and misses in these caches to determine the vulnerability of memory locations by modifying the *g-cache* module of Simics. Also, to trace instructions that belong to the application for which we want to calculate TVF, the *linux-process-tracker* tool provided by Simics is used. Since the process-tracker tracks all processes and threads in the system, we add magic instructions that trigger the tracker and enable to track only application threads. The results from this tracking are then used in our modified *trace* module to collect statistics about instructions. Simics infrastructure, which provides our system simulation, is illustrated in Figure 2.8.

A sample trace code obtained from the *blackscholes* application by *trace* module is demonstrated in Table 2.4. The table includes assembly instructions on the left

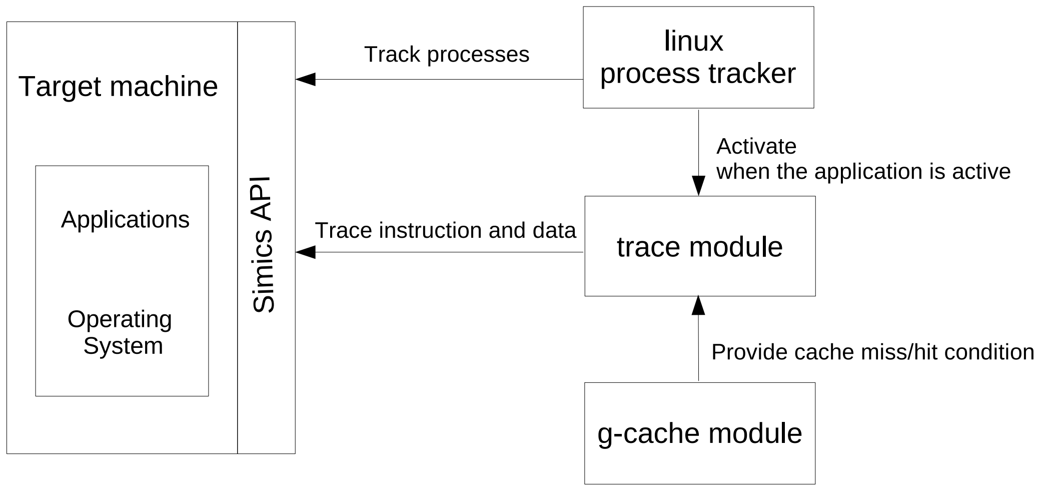


Figure 2.8. Simics architecture used in our experiments.

and the corresponding memory accesses for these instructions on the right side. The instruction sequence numbers (which are given as [xxxxxx]) lead to assembly instructions on the instruction list. The application threads, which have data dependency on each other, have both local and remote vulnerability values. There are two memory locations read or written from the L1 cache in the program flow (*g-cache* module provides the cache statistics which specify that these data accesses are hit on the L1 cache location). Both instruction and data access traces have been obtained via the simulation. First, $0ede7418$ which is pointed by $-24[ebp]$ is vulnerable between two different intervals including $895624 - 895636$ and $895640 - 895653$ and the other memory location $0ede7414$, which is pointed by $-28[ebp]$, is vulnerable between $895643 - 895644$ instructions. Therefore, only one code segment is considered to calculate the vulnerability of the location $0ede7414$ while both intervals are taken into account to get the vulnerability factor of the memory location $0ede7418$. It means that the first memory location in consideration ($0ede7418$) has much higher vulnerability factor value.

Another trace example from the same application illustrates the remote vulnerability of one thread, as shown in Table 2.5. Here, $Thread_5$ loads the value in the

Table 2.4. Sample memory accesses of a thread.

Instruction	**Memory access**
[*895624*] mov dword ptr **-24[ebp]**,0x0	Write to **0x0ede7418**
....	
[*895636*] mov ecx,dword ptr **-24[ebp]**	Read from **0x0ede7418**
....	
[*895640*] mov dword ptr **-24[ebp]**,edx	Write to **0x0ede7418**
....	
[*895643*] sub dword ptr **-28[ebp]**,0x1	Write to **0x0ede7414**
[*895644*] mov eax,dword ptr **-28[ebp]**	Read from **0x0ede7414**
....	
[*895653*] mov ecx,dword ptr **-24[ebp]**	Read from **0x0ede7418**

memory location $cffde000$ which has been stored by $Thread_1$. The remote vulnerability factor value of $Thread_5$ is calculated by the vulnerability factor of $Thread_1$ in the state where the store operation has occurred. The vulnerability factor with respect to ALU, register and memory resources of $Thread_1$ becomes the remote vulnerability factor of $Thread_5$. For each *load* operation in $Thread_5$, the remote vulnerability value is increased by the vulnerability factor of the thread, which is stored on that memory location.

Table 2.5. A sample communication of two threads.

$Thread_1$	$Thread_5$
[*1071340*] sub dword ptr **4[edi]**,0x1 (Write to **0xcffde000**)	...
...	[*1071343*] mov eax,dword ptr **4[edi]** (Read from **0xcffde000**)
...	...
...	[*1071364*] mov eax,dword ptr **4[edi]** (Read from **0xcffde000**)

2.3.2. Benchmarks

To collect the TVF related statistics, we use two parallel benchmark suites: PARSEC [38] and SPLASH-2 [39]. SPLASH-2 suite has been developed to support the study of distributed and shared-address space multiprocessors for multithreaded workloads. PARSEC is a relatively new benchmark suite which has focused on the shared-memory multicore processors. These suites have various applications with different characteristics [40]. We use medium-sized input provided with PARSEC benchmark and default problem size with SPLASH-2 benchmark. *Pthread* implementations of the applications are used in our experiments. We select five applications (*blackscholes, canneal, streamcluster, fluidanimate, swaptions*) from PARSEC suite and five applications (*barnes, fmm, cholesky, water-spatial, radix*) from SPLASH-2 suite for our experimental study. They are chosen to provide a variety of data sharing patterns at different domains, as explained in the following:

- *blackscholes* is an Intel RMS benchmark which computes partial differential equations in order to calculate the prices for a portfolio of European option. The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. It has 100 runs of the price calculation process.
- *canneal* tries to minimize the routing cost of a chip design by using simulated annealing method. It has many synchronization points which causes much inter-thread communication. The medium-sized input set used in our experiments has 64 temperature steps and 15,000 runs (swaps) per temperature step.
- *streamcluster* is an Intel RMS kernel which solves the online clustering problem for a stream of input points. The program employs partitioning of data points and it is memory bound for low-dimensional data while it has intensive computations for higher dimensions. It has 8,192 input points for the medium-sized input set.
- *fluidanimate* is an Intel RMS application which simulates an incompressible fluid for animation purposes by using Smoothed Particle Hydrodynamics method. It executes five different kernels and the work of each kernel is divided into the number of threads. It does not have much data sharing between threads.

- *swaptions* is an Intel RMS workload which computes the prices for a portfolio of swaptions by using Monte Carlo simulation. The medium-sized input set has 32 swaptions and 10,000 simulations.
- *barnes* is implementation of Barnes-Hut N-body method simulating the interaction of a system of bodies. The communication patterns are dependent on the particle distribution and not structured. The default particle number equals to 16,384.
- *fmm* is another application which simulates the interaction of system bodies by using another method called as Fast Multipole Method. Since there is no specific distribution of particle data in main memory, the communication patterns are unstructured. It has also 16,384 particles as default.
- *cholesky* factors a matrix into the product of a lower triangular matrix and its transpose. Since the program works on sparse matrices, it has higher communication than computation. The matrix used in our experiments has 134,579,320 columns and 134,579,328 rows.
- *water-spatial* evaluates forces and potentials occurring in a system of water molecules. The program uses cell grids which access other cells during execution, which causes communication. The default number of molecules is 512.
- *radix* implements an integer radix sort based on an iterative algorithm. Each iteration involves a local histogram generation where all histograms are accumulated into a global histogram. Then a permutation phase which requires all-to-all communication is performed by using this global histogram. Since the permutation is determined by sender threads, the communication is based on write operations. The default number of integers to be sorted is 262,144.

2.4. Experimental Results

While thread vulnerability factors can be calculated in principle for any architectural resource, we focus on three important components: registers, ALUs and cache locations. We want to emphasize that the L2 cache in our simulated multicore machine is assumed to be protected using ECC. Consequently, in this setting, error transfer across cores can happen in two different ways. First, a wrong value can be calculated by a core (e.g., due to a soft error that hit the ALU) and written to shared L2. This (wrong) value can then be read by another core and it corrupts the computation in this consumer core. The second way is that a correct value is written to L1; but before it can get to transfer to L2, it can be hit by a soft error while residing in L1. To summarize, although L2 is protected well, if it gets a wrong value, it will pass it to other cores.

We execute our parallel benchmark applications with different number of threads. Each multithreaded application considered in our experimental study is a parallel *Pthread* implementation and provides executions with different number of threads. The overall work of an application is divided into a number of work units that are equal to the number of threads (i.e., a homogeneous distribution), which are processed concurrently. Our experiments are conducted by using the same number of threads as the core counts in the target architecture. Moreover, we assign one thread per core by mapping one application thread to the specific core in the system.

2.4.1. TVF Results

We calculate the thread vulnerability factor of each thread in our applications with respect to L1 caches, register file, and ALU units. Each thread, which reads data from another thread, has both local and remote vulnerability factor values. Unless otherwise stated, we use equal local and remote vulnerability weights. Table 2.6 and Table 2.7 give the local and remote TVF values for our PARSEC and SPLASH-2 applications in two-thread and four-thread execution scenarios. Local VF values for each thread of a given application are very close to each other since threads in our applications execute similar code bodies and consequently perform similar work. The values with respect to ALU and register resources are almost the same while LVF values resulting from cache memory are slightly different due to the VF calculations which depend on hit/miss patterns. Further, VF values of each thread for two cases (two-thread and four-thread) are similar. Also, the total VF values for each resource, calculated by augmenting the values per core, increase as the number of threads increases.

Table 2.6. TVF values of our PARSEC benchmark applications for 2-core and 4-core executions.

			2-core execution		4-core execution			
			T1	T2	T1	T2	T3	T4
blackscholes	LVF	ALU	0.221	0.221	0.221	0.221	0.221	0.221
		Register	0.222	0.222	0.222	0.222	0.262	0.263
		Memory	0.118	0.167	0.118	0.167	0.222	0.219
	RVF	ALU	0.184	0.147	0.184	0.147	0.147	0.147
		Register	0.205	0.189	0.205	0.189	0.189	0.189
		Memory	0.150	0.125	0.150	0.126	0.126	0.126
fluidanimate	LVF	ALU	0.296	0.296	0.296	0.295	0.296	0.295
		Register	0.475	0.452	0.475	0.481	0.481	0.452
		Memory	0.838	0.828	0.834	0.824	0.826	0.815
	RVF	ALU	0.310	0.290	0.309	0.297	0.287	0.284
		Register	0.310	0.240	0.322	0.265	0.236	0.241
		Memory	0.462	0.321	0.469	0.362	0.316	0.310
streamcluster	LVF	ALU	0.452	0.451	0.450	0.450	0.451	0.451
		Register	0.385	0.403	0.396	0.381	0.390	0.390
		Memory	0.381	0.789	0.210	0.772	0.773	0.773
	RVF	ALU	0.440	0.433	0.434	0.426	0.426	0.425
		Register	0.386	0.366	0.389	0.378	0.377	0.376
		Memory	0.660	0.560	0.629	0.515	0.522	0.521
canneal	LVF	ALU	0.199	0.199	0.199	0.199	0.199	0.199
		Register	0.223	0.332	0.281	0.334	0.278	0.281
		Memory	0.483	0.495	0.450	0.464	0.462	0.464
	RVF	ALU	0.212	0.217	0.212	0.217	0.216	0.216
		Register	0.332	0.328	0.352	0.345	0.343	0.341
		Memory	0.310	0.302	0.313	0.311	0.311	0.311
swaptions	LVF	ALU	0.491	0.491	0.491	0.491	0.491	0.491
		Register	0.255	0.313	0.251	0.309	0.269	0.251
		Memory	0.016	0.016	0.011	0.011	0.011	0.011
	RVF	ALU	0.344	0.198	0.347	0.246	0.248	0.207
		Register	0.189	0.172	0.220	0.203	0.203	0.195
		Memory	0.125	0.087	0.140	0.115	0.116	0.101

Table 2.7. TVF values of our SPLASH-2 benchmark applications for 2-core and 4-core executions.

			2-core execution		4-core execution			
			T1	T2	T1	T2	T3	T4
barnes	LVF	ALU	0.290	0.290	0.290	0.290	0.290	0.290
		Register	0.415	0.348	0.397	0.268	0.327	0.283
		Memory	0.289	0.412	0.267	0.361	0.350	0.381
	RVF	ALU	0.335	0.334	0.328	0.328	0.327	0.329
		Register	0.443	0.457	0.448	0.445	0.450	0.453
		Memory	0.274	0.258	0.271	0.260	0.266	0.259
cholesky	LVF	ALU	0.457	0.512	0.410	0.443	0.456	0.464
		Register	0.430	0.295	0.314	0.239	0.271	0.299
		Memory	0.108	0.088	0.077	0.081	0.073	0.087
	RVF	ALU	0.470	0.473	0.461	0.470	0.447	0.456
		Register	0.288	0.289	0.333	0.323	0.320	0.336
		Memory	0.123	0.149	0.130	0.138	0.131	0.132
fmm	LVF	ALU	0.502	0.503	0.495	0.502	0.494	0.503
		Register	0.368	0.389	0.295	0.327	0.320	0.288
		Memory	0.334	0.355	0.266	0.313	0.312	0.298
	RVF	ALU	0.394	0.333	0.404	0.381	0.390	0.377
		Register	0.344	0.338	0.333	0.338	0.340	0.342
		Memory	0.322	0.299	0.267	0.266	0.264	0.279
radix	LVF	ALU	0.329	0.329	0.329	0.329	0.329	0.329
		Register	0.305	0.229	0.235	0.222	0.305	0.230
		Memory	0.269	0.266	0.264	0.259	0.262	0.255
	RVF	ALU	0.303	0.309	0.308	0.310	0.309	0.310
		Register	0.257	0.269	0.276	0.277	0.276	0.277
		Memory	0.395	0.431	0.408	0.414	0.409	0.413
water-spatial	LVF	ALU	0.242	0.238	0.245	0.238	0.238	0.239
		Register	0.502	0.416	0.455	0.328	0.328	0.353
		Memory	0.620	0.699	0.559	0.718	0.709	0.715
	RVF	ALU	0.220	0.200	0.219	0.203	0.203	0.199
		Register	0.295	0.225	0.300	0.243	0.243	0.240
		Memory	0.425	0.217	0.444	0.253	0.252	0.254

Although the local values are similar for different threads, the remote values exhibit different patterns, which stems from the data sharing behavior of the threads in the application. If we consider the *canneal* benchmark simulated on a four-core architecture, although the local values with respect to register resources are very similar, remote values with respect to the same resource differ among the threads (see Table 2.6).

It is also observed that the source of RVF values changes by the data sharing pattern of the threads. As an example, the RVF values with respect to ALU for the *blackscholes* benchmark simulated on four-thread architecture are different. The last three threads have similar values (which is equal to 0.147), but the first thread has more remote vulnerability value (which is equal to 0.184) for ALU resource. This results from *blackscholes*'s data access and sharing patterns (the first thread has input data at the beginning of the execution and distributes to the other threads). The remote values of the other threads completely come from the first thread. At the end of the execution, the output values generated by the worker threads are sent to the first thread to form the complete output. Therefore, the first thread obtains data from all other threads in the application, and this influences its remote VF values.

On the other hand, the threads in the *canneal* benchmark have stronger data sharing. Each thread in the application frequently reads from and writes to other threads during its lifetime. Therefore, the remote vulnerability factor values of these threads are influenced by all other threads in the application. Further, *streamcluster* threads have distinct data sharing characteristics which results in different RVF values for each thread with respect to ALU and cache resources.

We obtain TVF of an entire multithreaded application by summing TVF values across all threads (cores for one thread per core case). Figure 2.9 presents the remote vulnerability factor (RVF) values for register file, ALUs, and L1 (data) caches as well as execution cycles, where higher number of cores are considered. Since Local vulnerability factor (LVF) values have been counted for remote vulnerability factor (RVF) calculation, we consider only RVF values which include both LVF and RVF values of threads communicating with target thread for Figure 2.9.

To present the values in the same plot, the RVF values and execution cycles are *normalized* to have values between 0 and 1. These plots help us to carry out a performance-reliability tradeoff analysis. Specifically, if we consider the plots given in Figure 2.9, one can see that, while the parallel execution time gets reduced as we increase the number of cores, the RVF values tend to increase.Therefore, based on the amount of performance one can sacrifice, resilience against soft errors may vary. For instance, if we execute the *streamcluster* application using 12 threads, we achieve a normalized execution cycles value of 0.20. The cache RVF value under the same core count is about 0.76. However, if we are willing to work with an execution cycles count of 0.25 (25% worse than 0.20), we can use 8 cores (instead of 12 cores) and achieve a cache RVF value of 0.52 (35% better than 0.74). To sum up, by sacrificing 25% performance, one can obtain about 35% improvement in overall cache RVF value. Similar observations can be made with other benchmarks as well. These results clearly illustrate potential performance-reliability tradeoffs one can explore using thread vulnerability factors.

In our experimental study, the computation time and memory requirements of trace process, which includes tracing of all resources and distinguishing the resources which affect the remote write, are very large. Therefore, we take into account overall TVF of the remote thread to compute the RVF of one thread, instead of taking only TVF of the resources that affect the store operations performed remotely. Specifically, we consider overall LVF_{RF} of $Thread_1$ at the end of the store operation to calculate RVF of $Thread_2$ in the example given in Section 2.2.2. Here, the value of $r3$ register is written to the shared memory location and the resources that may affect the remote write are only the registers used to calculate this value. However, we do not trace $r3$ register to find out these registers and we consider the overall vulnerability of $Thread_1$ at the remote write operation.

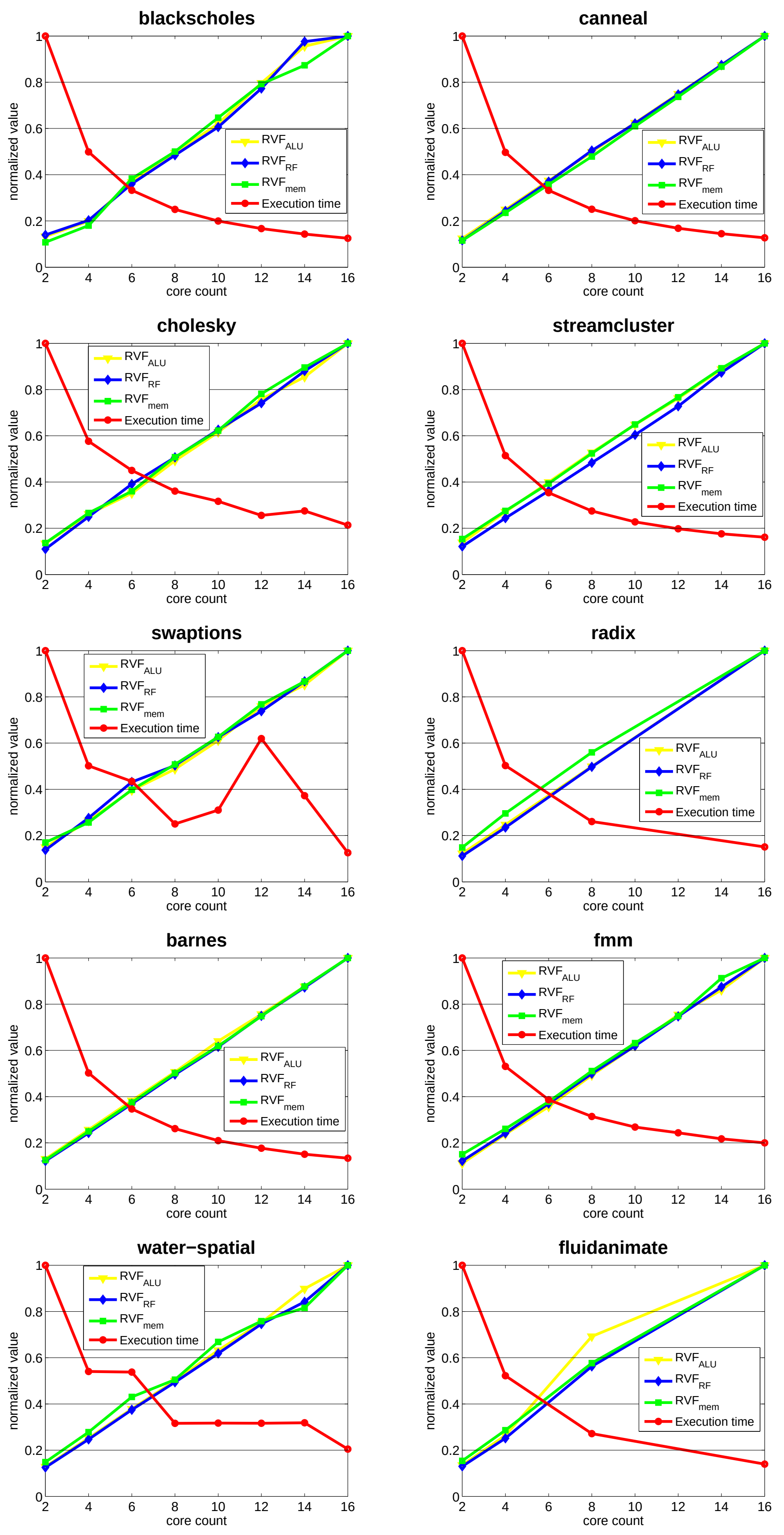


Figure 2.9. Normalized RVF values and normalized execution times of our benchmark applications.

2.4.2. Weight Analysis

The weight values in TVF definition (Equation 2.5) enable us to obtain vulnerability for the cases that need to consider different weights of local and remote terms. While we use equal weights in our experimental study unless otherwise stated, we also conduct experiments with different local and remote vulnerability weights to measure the impact of weight values on our results. We perform our experiments by using the same number of threads as the core counts and mapping one thread per core in the target architecture (Table 2.3). Figure 2.10 and Figure 2.11 show the remote vulnerability factors (averaged across all cores), calculated by three different weight combinations $(w_1, w_2, w3)$ for *blackscholes* and *cholesky* executed on different number of cores. The selected weights are $w_L = w_R = 0.5$ for w_1, $w_L = 0.7, w_R = 0.3$ for w_2 and $w_L = 0.3, w_R = 0.7$ for w_3.

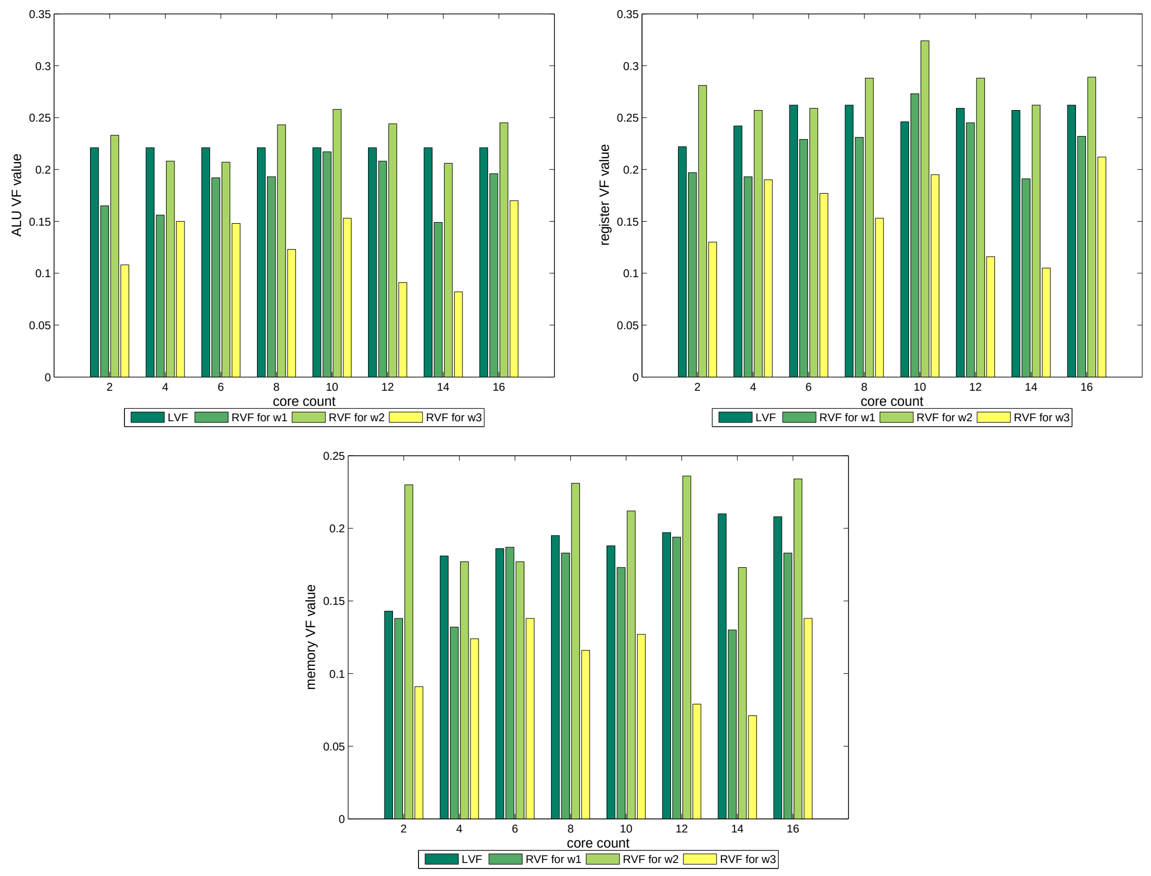


Figure 2.10. RVF values with different vulnerability weights (*blackscholes*).

We observe that the RVF values of these two applications have different variations as weight values change. The RVF values (with respect to all resources including memory, register and ALU units) of *blackscholes* benchmark tend to increase with an increase in local VF weights (w_2 case) and they tend to decrease with an increase in the remote VF weight (w_3 case). On the other hand, *cholesky*'s RVF values (in particular for the case of memory resources) are in the opposite direction which reduce with increasing value of the local VF weight (w_2 case) and increase with increasing value of the remote VF weight (w_3 case). This trend in RVF values is valid for all number of core counts. The values of *cholesky* application do not have such a distinguishing trend for ALU and register resources. Since RVF values are mainly contributed by memory operations, the effect of ALU operations and register resources is not large for that application.

As can be observed from the first two columns in Figure 2.10, which hold data related to the experiments conducted with equal weights, the local VF values for *blackscholes* application are larger than remote VF values in general. These larger local values which have larger contribution to RVF values lead to higher remote values if the weight of the local values which is used to calculate the RVF values are kept larger (70%); and when the weight of the local values is reduced (30%), the remote values decrease. For instance, average LVF value for ALU resource is 0.221 for 8-core execution and RVF value equals to 0.193 for equal weight case. The RVF value, which is calculated by summation of local and remote values, becomes larger (0.243) if we increase LVF weight to 70%. Nevertheless, the value decreases to 0.123 for smaller (30%) LVF weight.

We also conduct statistical tests to analyze the significance of the difference among vulnerability values. One-way ANOVA tests at a 95% confidence interval reveal that local vulnerability values for *blackscholes* application are significantly larger than remote vulnerability values with equal weights for ALU and register resources. The same analysis also shows that RVF values with larger local weight (w_2 case) are the largest remote values and the RVF values with larger remote weight (w_3 case) are the smallest remote values. Although the difference between local and remote values is not significant for memory resources, the remote vulnerability factor values are significantly different from each other and the case with larger local weight (w_2 case) has the largest remote vulnerability value.

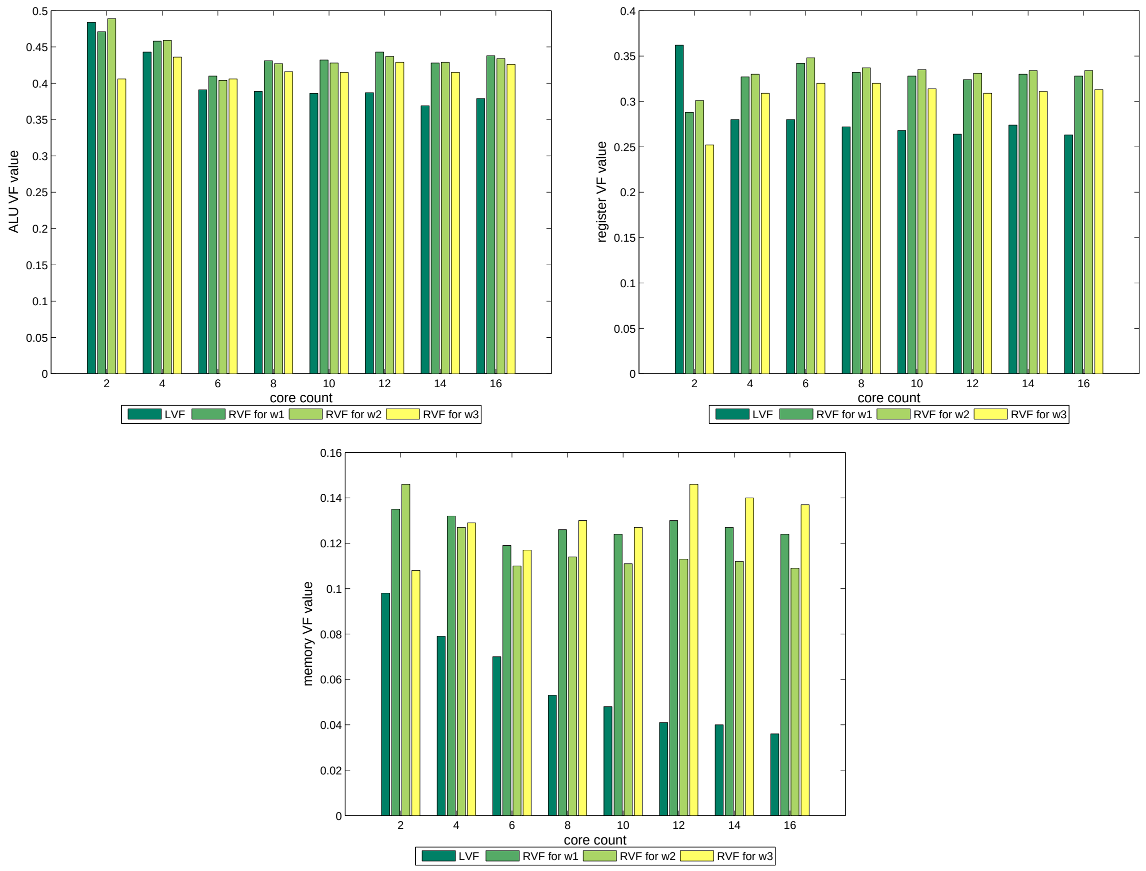


Figure 2.11. RVF values with different vulnerability weights (*cholesky*).

Although the RVF values vary with the weight values, the characteristics of the resources are the same for each case. The VF values with respect to register resources are always the highest values and the values for memory (cache) resources are smaller than the values for ALU resources.

On the other hand, *cholesky* application has different characteristics for vulnerability values. While the local vulnerability values are mostly smaller than remote VF values with equal weights, the variation is not so large for ALU and register resources. Therefore, the difference among remote vulnerability values with different weights is not significant for these resources. However, memory resources have significantly larger remote vulnerability values (with equal weights) than local vulnerability values. Therefore, its RVF values which are mostly affected by (larger) remote values increase by larger remote VF weight (70%). This behavior can be observed from Figure 2.11. 12-core execution has 0.041 LVF value for memory resources while RVF value equals to 0.130. If we decrease the RVF weight (from 50% to 30%), the remote value decreases as well (0.113). However, the RVF value becomes larger (0.146) for larger RVF weight.

It is clear that the vulnerability of threads differs by giving different weights on local and remote vulnerability factors and the weight selection criteria is influential on the VF values.

2.4.3. Cache Size Variation

Since the vulnerability values with respect to memory resources are considered for only data residing on L1 cache location, it is highly possible that the cache size affects these vulnerability values. To analyze the effect of cache size variation in vulnerability of memory (cache) resource, we conduct experiments with configurations having different L1 cache sizes.

As given in Section 2.3, 16K L1 cache size is used in our experiments so far. To be able to observe the effect of cache size variation, we consider two more configurations with 8K and 32K L1 cache sizes for two applications (*canneal*, *barnes*) with different memory access patterns from each benchmark. Figure 2.12 and 2.13 demonstrates TVF values (both LVF and RVF) with respect to memory (cache) resource for *canneal* and *barnes* applications respectively.

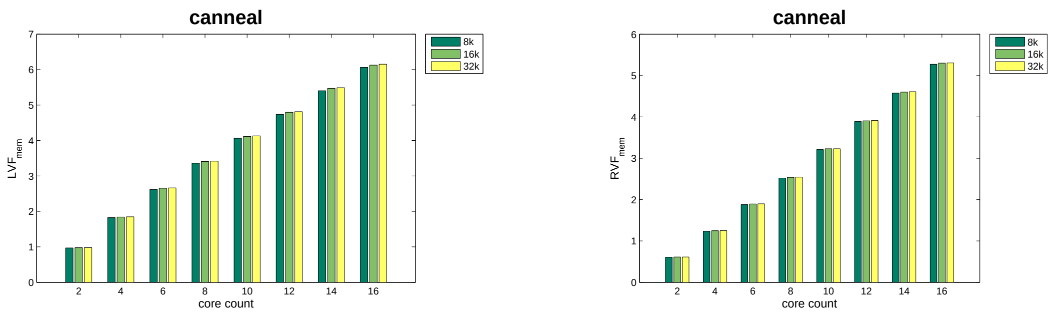


Figure 2.12. Memory TVF values for *canneal* with different cache sizes.

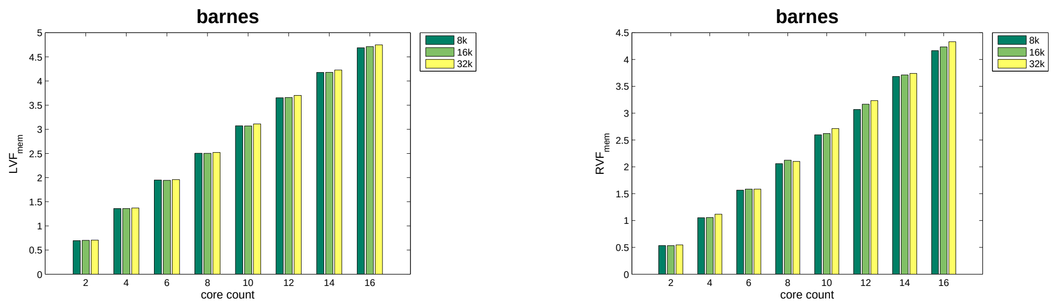


Figure 2.13. Memory TVF values for *barnes* with different cache sizes.

The results reveal that TVF values tend to increase as the cache size increases. Although the variation is not large, it is clear that the cache locations become more vulnerable to soft errors as the size (the probability of data residence) increases. Since cache hit ratio is larger for larger cache structures, our TVF metric for cache resources, which is calculated for cache hit condition, has larger values. For instance, *canneal* application has 0.605, 0.611 and 0.612 RVF values for 8K, 16K and 32K cache sizes respectively for 2-core execution. However, the RVF values are 2.520, 2.537 and 2.543 for 8-core execution; 5.273, 5.300 and 5.303 for 16-core execution.

3. VALIDATING THREAD VULNERABILITY FACTOR

Since our TVF metric is the first attempt for a reliability analysis of multicore architectures, it may not be practical its validation with comparisons due to the lack of any reference work on reliability evaluation for parallel applications. We conduct a validation study based on a set of fault injection experiments on the Simics environment, as an approximate analysis. Additionally, we extend our TVF framework to collect statistics and calculate metric values on a real multicore architecture. We conduct similar fault injection experiments on the real environment, and compare TVF analysis and fault injection results by analyzing the difference between them.

This chapter presents both validation study including fault injection experiments and the details of our TVF framework extension. Section 3.1 provides a background about fault injection-based reliability analysis and we present the details of our fault-injection framework by providing a comparison analysis between TVF values and fault-injection experiment results for a set of applications in Section 3.2. We present our TVF and fault injection framework for real environment in Section 3.3, which is followed by a comparison analysis on the real environment by using a set of selected benchmark applications.

3.1. An Overview on Fault Injection

Fault injection is a dependability validation technique based on the controlled experiments which introduce faults into the system [41]. Fault injection experiment setting requires the identification of fault space which includes fault type, fault location, and injection time.

Hardware fault injection uses additional hardware to introduce low-level faults. While it is possible to inject permanent hardware faults including stuck-at, open, and bridging [42], transient faults at random locations can also be injected by applying heavy ion radiation [43] and electromagnetic fields [44]. On the other hand, software fault injection targets applications and operating systems without any extra hardware requirement [45]. Compile time injection approaches may inject errors to source or assembly code, runtime injection techniques may implement time-out event to trigger injection [46] or use software traps to inject faults [47].

The dependability of parallel systems has been validated via fault injection as well as uni-processor systems. NFTAPE [48] architecture provides a fault injection framework for distributed systems' dependability analysis by supporting several fault injector classes. LOKI [49] implements a fault injection environment by considering global-state of distributed systems. Moreover, fault injection has been used to gather fault behavior of multithreaded applications running on multicore architectures [50].

Fault injection-based analysis [51] has been conducted for Architectural Vulnerability Factor (AVF), which is a metric to quantify the vulnerability as the probability that a fault in a processor structure will result in an error in the program output [23]. The study compares ACE analysis against a fault injection study, which includes single bit flip for pipeline stage, register files, data, and instruction buffers. Fault injection results demonstrate that ACE analysis is a conservative approach by providing a lower bound for reliability.

3.2. TVF Validation by Using Simics Environment

We conduct a simulation-based fault injection by assuming that the failures are uniformly distributed. Our fault injection campaign introduces a single bit flip on a register of one processor core during the execution of the target application. Mainly, our experiments select one bit position (among 32 bits), one register (among 8 registers), one processor core (among n cores), and one instruction (among number of instructions for the target application) for the injection point.

3.2.1. Fault Injection Framework

We use Simics toolset to construct our experiments, and build an automated tool for injection analysis. Figure 3.1 illustrates our fault injection framework, which includes several tools to implement the phases of the experiments. Our framework

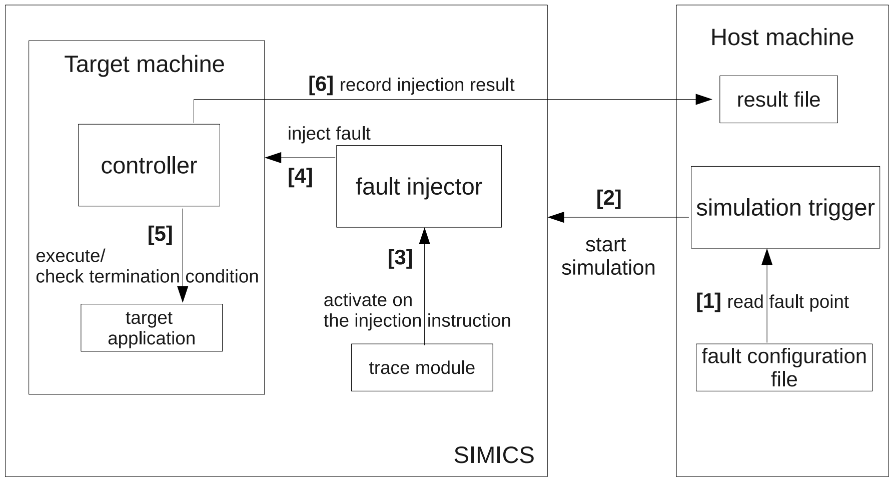


Figure 3.1. Our fault injection framework, where the numbers in the arrows represent the flow of our one experiment.

consists five modules given below:

(i) *Fault injection parameter file:* First of all, we randomly create uniformly distributed fault injection points by specifying fault injection instruction, processor core, register number, and register bit position and store the parameters of each experiment in a configuration file.

(ii) *Simulation trigger:* This module executes on the host machine and basically starts the execution of the fault injection simulations by providing parameters of the fault data. It reads the parameter for the first experiment, triggers the Simics with the first parameter set. After the experiment ends, it starts the following experiment given in the fault injection parameter file.

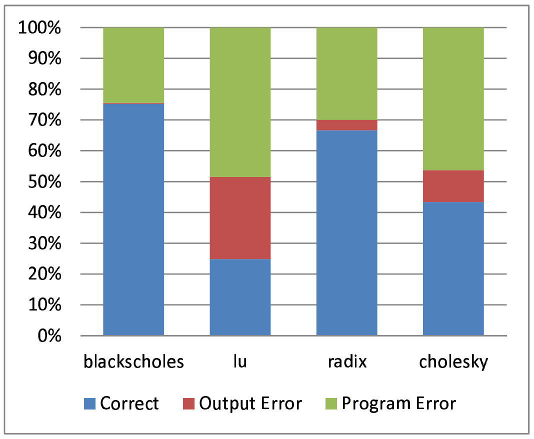
(iii) *Fault injector:* This module executes on the Simics middleware to manage fault injection experiment. It gets the fault injection parameters and enables the trace module, which tracks the target application. When the simulation reaches the fault injection point (the execution of the specified instruction on the specified processor core), trace module activates the related procedure of the fault injector. At this point, the fault injector flips the specified bit on the specified register and lets the simulation end up.

(iv) *Controller:* After the injection of the specified fault, the controller becomes the only active module in the framework. This module waits for the termination of the target application, which executes on the simulated target machine, and gathers the result of the experiment by evaluating the termination condition. If the program terminates with an error code (segmentation fault, floating point exception etc.), the experiment result is defined as *Program Error*. Moreover, the program may never terminate if the fault causes an infinite loop error or similar stuck failure. The controller handles these kind of failures by specifying a timeout for the execution. When the program still continues to execute for an amount of time exceeding the threshold time, the result of this fault injection experiment is also defined as *Program Error*. If the program terminates normally (with zero exit status), there are two different scenarios including *Correct Execution* and *Output Error*. To understand the result of the execution, the controller checks the output file created by the program by comparing it to the golden output file (the output file produced by the correct execution). If there is no difference between the output files, then the result is defined as *Correct Execution*. Otherwise, the result becomes *Output Error* and the details are logged into the fault injection result file.

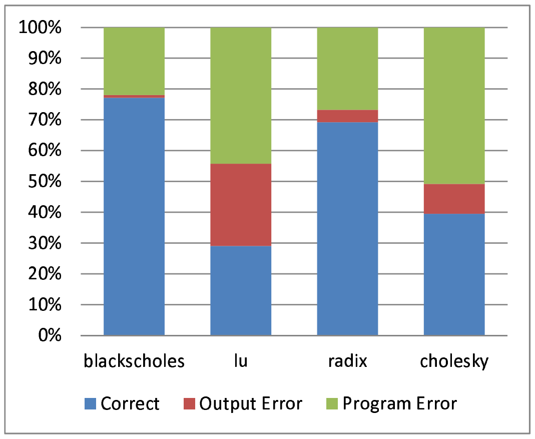
(v) *Fault injection result file:* The controller writes the result of the fault injection experiments to the fault injection result file stored on the host machine. When a new result is logged in the file, the simulation trigger finds out that the experiment finished, and continues with the following experiment. The result file is formed online by executing the experiments and processed offline by an external analysis tool to conclude overall fault injection scenario.

3.2.2. Experiments on the Simics Environment

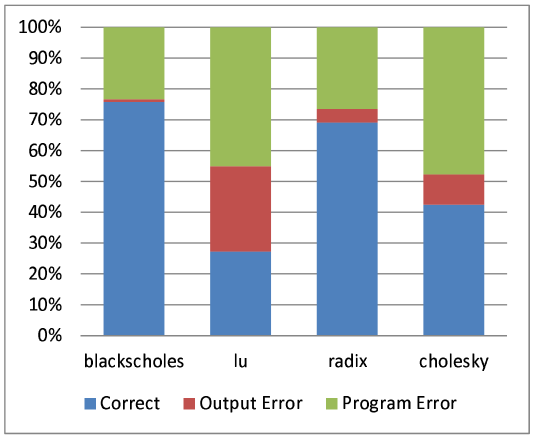
Although the program errors result in corruption of the complete execution, the errors that affect the program output may corrupt the program partially and may be tolerable. Therefore, the output errors may be analyzed and the failure severity may be calculated [52]. To analyze the output errors and detect data corruption, the application output should be deterministic and easy to compare. Therefore, we select a subset of applications that have exact results from the PARSEC and SPLASH-2 benchmarks for our fault injection experiments including *blackscholes*, *LU*, *cholesky*, and *radix*.



(a) 4 core execution



(b) 8 core execution



(c) 16 core execution

Figure 3.2. Fault injection experiment results for 4 benchmark applications.

We analyze the results of the experiments by comparing the correct results (the golden output) with the experimental results. While the program failures such as segmentation fault, floating point error, and the program timeout cases are categorized as the application crash, the termination with a successful exit status cases may represent the correct execution or may represent the data corruption. The former output has no difference from the golden output, while the latter output differs from the golden output.

We execute our target applications on our fault-injection framework to validate our TVF analysis. We consider Silent Data Corruptions (SDCs) which include both self-thread errors and fault propagation errors in a parallel program execution, and use SDC rate as a metric to compare our results. SDC rate is the fraction of the injected faults that results in unacceptable outputs [50]. We assume all data corruptions are unacceptable, because we do not classify the data corruptions as acceptable or unacceptable. Since TVF measures the relative vulnerability of multithreaded applications, we compare the trend of four selected applications for both TVF values and SDC rates.

We conduct fault injection experiments for 4-core, 8-core, and 16-core architectures by introducing one bit flip for one register of a specific processor core during a specific instruction execution. We define two set of bit levels (the lower and upper 16 significant bits), two types of registers (data and address registers), and generate $2 \times 2 \times n \times r$ fault points, where n represents the number of cores in the system and r is the replication count (50 for our case), by uniformly distributing the faults through program execution. We have $2 \times 2 \times 4 \times 50 = 800$ experiments for 4-core architecture, $2 \times 2 \times 8 \times 50 = 1600$ experiments for 8-core architecture, and $2 \times 2 \times 16 \times 50 = 3200$ experiments for 16-core architecture for each selected application. Figure 3.2 represents the fault injection experiment results by reporting execution outcomes.

We calculate RVF values for register resources, and collect SDC rates from the fault injection experiments for each application. Figure 3.3 presents both results in the same scale by considering the relative vulnerability of four selected applications. Since thread vulnerability factor is not a measure to represent absolute reliability, the exact values are different from SDC rates. On the other hand, the values for both measures including vulnerability and fault injection results are close to each other if they are represented in the same scale (Figure 3.3). It is observed that the applications have similar behavior for RVF values and SDC rates, which both represent the vulnerability of the applications. While *lu* has the largest RVF values and SDC rates, *blackscholes* has the smallest values. If we compare the vulnerability of all pair-wise applications, we can see that it is possible to conclude with the same result for RVF metric and SDC rates.

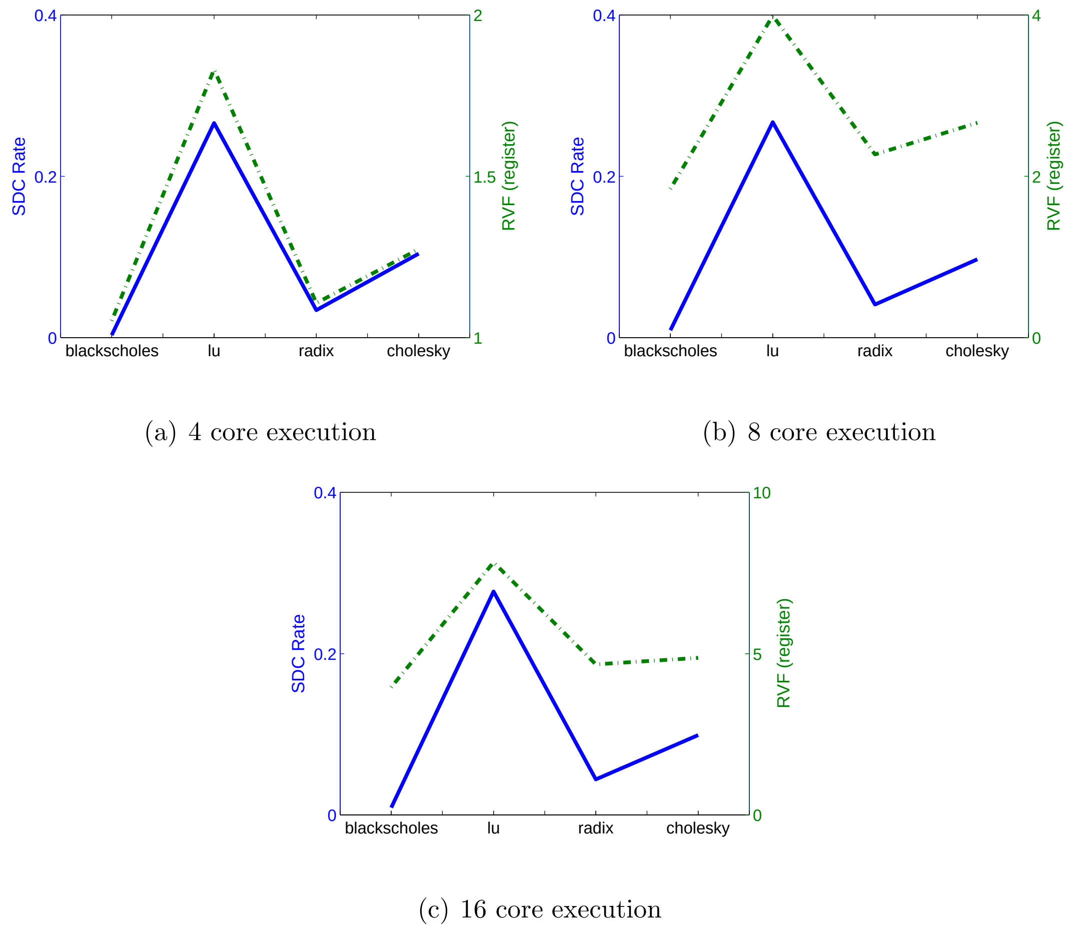


Figure 3.3. Vulnerability values (SDC rate and RVF value) for 4 benchmark applications.

Since the fault-injection experiments include one faulty-case for each execution and it does not differ for different number of cores in the system, SDC rates for different architectures are similar. However, the RVF values increase as the number of cores in the system increases. Therefore, the proportion between RVF values and SDC rates are not the same for the applications while both metric provide the same ordering for all applications.

3.3. TVF Validation on a Multicore Architecture

We implement TVF evaluation on Simics architecture (Section 2.3) to execute multithreaded applications on different architectures having different number of cores, cache level and sizes. To evaluate TVF metric values on a real environment and compare the results with the fault injection results, we also extend our implementation by using Pin system which performs dynamic binary instrumentation for both single-threaded and parallel programs [4, 53].

3.3.1. Pin Overview

Pin is a dynamic binary instrumentation framework that provides the creation of dynamic program analysis tools. The tools created using Pin, called Pintools, make possible the analysis of user programs by performing instrumentation at run time on the binary executable files. As shown in Figure 3.4, Pin environment includes three levels including Pin, Pintool, and the user application. Pintool communicates with Pin via instrumentation and analysis routines. The Pin virtual machine coordinates the application execution [4]. A pintool consists of instrumentation, analysis, and callback routines. Instrumentation routines decide the code insertion point. Analysis routines include the code to execute at insertion points. Callback routines are invoked when an event occurs. Figure 3.5 presents a simple Pintool that generates a trace of all memory addresses referenced by a program. Whenever an instruction is executed by the application, $Instruction$ instrumentation function is called by Pin. It inserts a call to analysis functions $RecordMemRead$ or $RecordMemWrite$, if the instruction is a memory read or memory write, respectively. $Fini$ function is called at the end of the program execution, as registered in $main$ function, which registers instrumentation and callback functions.

Pin also provides callbacks when each thread starts and ends, and makes possible

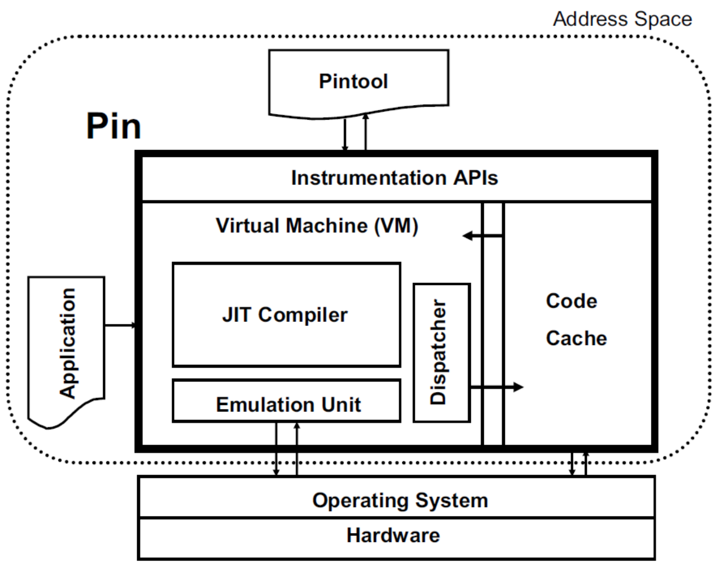


Figure 3.4. Pin's software architecture [4].

tool implementation for multithreaded applications. Instrumenting a multithreaded program requires that the tool be thread safe. Pin API has primitives for locking mechanism (lock, mutex, semaphore) to avoid deadlocks and provide synchronization between threads of a parallel program.

To calculate Thread Vulnerability Factor of a parallel application, we implement a thread-level Pintool to trace all instructions and memory operations of a parallel program. We ensure the synchronization of threads by using Pin's lock mechanism and inject our LVF and RVF calculation code into analysis routines protected by locks for each thread. Our Pintool provides TVF calculation of any parallel program running on a target architecture. We also extend register fault pin tool for parallel applications, which injects the fault into an architectural register and analyzes the impact of the fault in the application [54]. Figure 3.6 presents the analysis routine of our pin tool which inserts a bit flip fault to the specified bit of the specified register on the faulty core.

3.3.2. Experiments on the Multicore Architecture

We conduct our validation study on the real environment for the same set of benchmark applications selected for the simulation environment (Section 3.2.2). We

```
#include <stdio.h>
#include ‘‘pin.H’’
FILE * trace;
// Print a memory read record
VOID RecordMemRead(VOID * ip, VOID * addr){
    fprintf(trace,‘‘%p: R %p\n’’, ip, addr);
}
// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr){
    fprintf(trace,‘‘%p: W %p\n’’, ip, addr);
}
// Is called for every instruction and instruments reads and writes
VOID Instruction(INS ins, VOID *v){
    UINT32 memOperands = INS_MemoryOperandCount(ins);
        // Iterate over each memory operand of the instruction.
    for (UINT32 memOp = 0; memOp < memOperands; memOp++){
        if (INS_MemoryOperandIsRead(ins, memOp)){
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
                IARG_INST_PTR, IARG_MEMORYOP_EA, memOp, IARG_END);
        }
        if (INS_MemoryOperandIsWritten(ins, memOp)){
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
                IARG_INST_PTR, IARG_MEMORYOP_EA, memOp, IARG_END);
        }
    }
}
VOID Fini(INT32 code, VOID *v){fclose(trace);}
int main(int argc, char *argv[]){
    PIN_Init(argc, argv);
    trace = fopen(‘‘pinatrace.out’’, ‘‘w’’);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Figure 3.5. Pintool for printing addresses of all program memory reads and writes.

```
VOID InsertFault(CONTEXT* _ctxt, THREADID threadid)
{
    if(threadid == faultCore){
        GetFaultyBit(_ctxt, &faultReg, &faultBit);

        UINT32 old_val = PIN_GetContextReg(_ctxt, faultReg);
        faultMask = (1 << faultBit);
        UINT32 new_val = old_val ^ faultMask;
        PIN_SetContextReg(_ctxt, faultReg, new_val);

        PIN_RemoveInstrumentation();
        faultDone = 1;
        PIN_ExecuteAt(_ctxt);
    }
}
```

Figure 3.6. Fault insertion analysis routine.

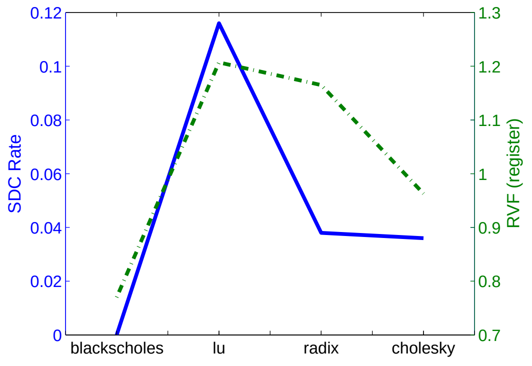
execute *blackscholes*, *LU*, *cholesky*, and *radix* applications on a 16-core architecture by utilizing different number of cores. The characteristics of our architecture are given in Table 3.1. Our experiments include both TVF calculation and fault injection phases for each application for different thread counts.

Table 3.1. Characteristics of our workstation environment.

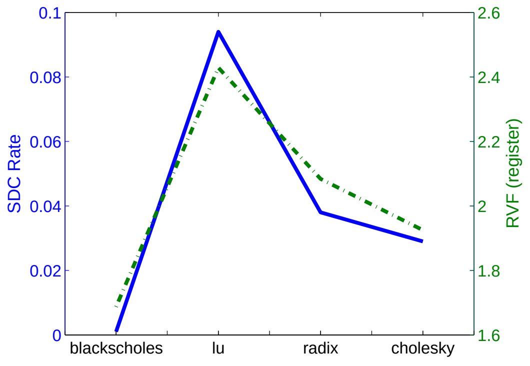
Processor	Dual Intel Xeon Processor E5-2680, 2.7 GHz, 20 MB cache, 1600 MHz memory, Eight-core
Hard disk	256GB SATA 1st SSD, 1TB SATA 7200 2nd HDD
Main memory	32GB (8x4GB) DDR3-1600
Operating system	Red Hat Enterprise Linux 6

Figure 3.7 presents both SDC rates and RVF values by considering the relative vulnerability of four selected applications. Our real environment tests give consistent results with our simulation results. Although the values for both cases are not the same for simulation and real environment case, the relative behavior is not very different.

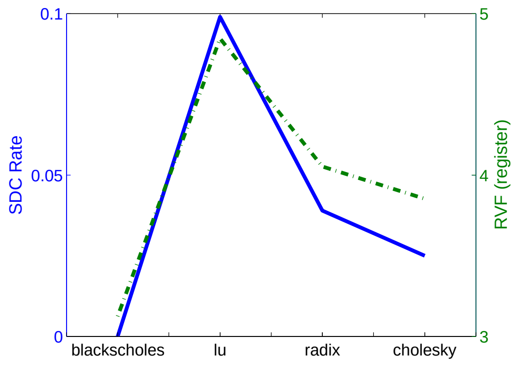
While *lu* has the largest RVF values and SDC rates, *blackscholes* has the smallest values. The order of applications' reliability is the same for both RVF values and SDC rates on our 4-core, 8-core, and 16-core utilization cases.



(a) 4 thread execution



(b) 8 thread execution



(c) 16 thread execution

Figure 3.7. Vulnerability values (SDC rate and RVF value) for 4 benchmark applications running on our 16-core architecture.

4. PERFORMANCE-RELIABILITY ANALYSIS OF MULTITHREADED APPLICATIONS

As illustrated in TVF-execution clock cycles curves (Figure 2.9), in general, performance of a multithreaded application increases as the number of cores increases in a multicore system; however the reliability, which is represented in terms of vulnerability of the application threads, tends to decrease. This behavior leads to a performance-reliability tradeoff for different core counts in the system. One can choose to work with lower performance but higher reliability for safety-critical applications. Such tradeoff may be exploited for different parallel implementations of an algorithm or different algorithms of a problem. By analyzing system behavior in terms of vulnerability and performance, one implementation or algorithm may be selected to achieve system need.

We conduct an analysis for multithreaded applications running on multicore architectures by considering performance and reliability characteristics of different versions [55]. We first present details of selected multithreaded applications for our analysis in Section 4.1. Then, results of performance-reliability tradeoff for different versions of the applications are given in Section 4.2.

4.1. Multithreaded Applications

To evaluate performance and reliability behaviors of multithreaded applications, we select three scientific applications; which are Fast Fourier Transform, Jacobi, and Water Simulation.

4.1.1. Fast Fourier Transform

Fast Fourier Transform (FFT) is an algorithm to compute discrete Fourier Transform (DFT) which is a mathematical transform requiring complex number calculations and used in various applications including time series, partial differential equations, and digital signal processing [5]. While DFT is a mathematical transform which requires complex number calculations, FFT is an algorithm to compute this transform by reducing its complexity. There are several forms of FFT algorithm; in this analysis, our focus is on one-dimensional, unordered, radix-2 FFT.

Algorithm 4.1 represents the sequential version of n point one-dimensional FFT algorithm. **X** and **Y** parameters are the input vector and output fourier transform, respectively. This algorithm divides DFT into smaller DFT computations along with multiplications by complex roots of unity, known as twiddle factors, where ω is the primitive nth root of unity in the complex plane which equals to $e^{2\pi\sqrt{-1}/n}$.

In the algorithm, the outer loop is executed $log\ n$ times, and the inner loop is executed n times for each iteration of the outer loop, which results in $\Theta(n\log n)$ time complexity.

The highest computation effort is required for the calculation of $R[i]$ by using $S[j]$ and $S[k]$. The pattern of combination of input array elements used in this calculation is represented by a butterfly network as shown in Figure 4.2. In a parallel algorithm, it is important to assign input elements into threads by considering communication cost, which affects both performance and reliability. We consider two parallel algorithms defined in [56]: binary-exchange and transpose algorithms which have different thread communication pattern.

In the binary-exchange algorithm, the array elements, which are denoted as their binary representations, are mapped to cores such that elements with indices having the same $d = logp$ most significant bits are mapped into the same core where p is the number of cores in the system. For 8-point FFT calculation on 4 cores, mapping is applied as follows: X[0] (000) and X[1] (001) elements are mapped into $Core_1$, X[2] (010) and X[3] (011) elements are mapped into $Core_2$, X[4] (100) and X[5] (101) elements are mapped into $Core_3$, X[6] (110) and X[7] (111) elements are mapped into $Core_4$.

```
procedure $serial\_FFT(X, Y, n)$
  $r \leftarrow logn$
  for $i \leftarrow 0$ to $n-1$ do
    $R[i] \leftarrow X[i]$
  end for
  for $m \leftarrow 0$ to $r-1$ do
    for $i \leftarrow 0$ to $n-1$ do
      $S[i] \leftarrow R[i]$
    end for
    for $i \leftarrow 0$ to $n-1$ do
      $j \leftarrow (b_0 ... b_{m-1} 0 b_{m+1} ... b_{r-1})$
      $k \leftarrow (b_0 ... b_{m-1} 1 b_{m+1} ... b_{r-1})$
      $R[i] \leftarrow S[j] + S[k] * \omega^{b_m b_{m-1} ... b_0 0 ... 0}$
    end for
  end for
  for $i \leftarrow 0$ to $n-1$ do
    $Y[i] \leftarrow R[i]$
  end for
```

Figure 4.1. The Cooley-Tukey algorithm for one-dimensional, unordered, radix-2 FFT [5].

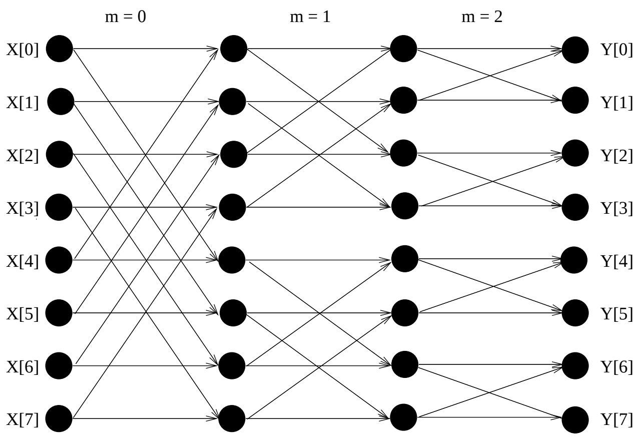


Figure 4.2. The pattern of combination of input array elements in an 8-point FFT computation.

The elements belong to different cores are combined during first d iterations (out of m iterations), while the elements belong to the same cores are combined in the remaining iterations. The communication between the cores only exists along these first d iterations. Figure 4.3 represents the combination pattern of array elements and the communication behavior of cores among iterations for 16-point FFT calculation on 4-cores system.

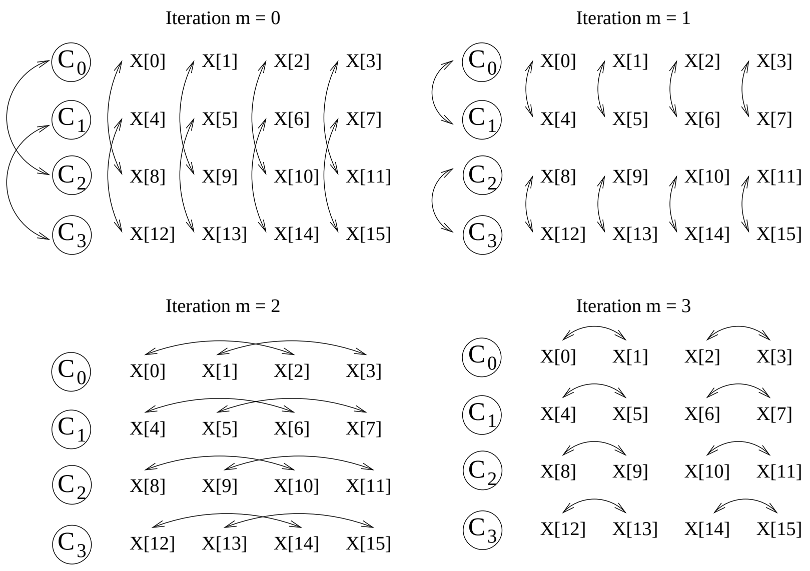


Figure 4.3. Binary-exchange algorithm for a 16-point FFT on four cores.

The transpose algorithm, which involves a matrix transposition operation, requires smaller amount of communication among cores. The input array with size n is arranged in an $\sqrt{n} \times \sqrt{n}$ two dimensional array in row major order where FFT calculation can be performed by applying FFT over the rows and then applying FFT over the columns. These array elements are mapped to p cores such that each core stores $\sqrt{n}/p$ rows. FFT over the rows requires no communication among the cores. After transposition, FFT over the rows (old columns) is computed to complete FFT operation. The communication between the cores is needed for only transposition operation. Figure 4.4 represents the combination pattern of array elements among iterations and shows that there is no communication between cores for 16-point FFT calculation on 4-cores system.

The binary-exchange algorithm, which requires more communication among its threads, performs well on parallel architectures with high communication bandwidth (e.g., chip multiprocessors which have processors on a single die) than the transpose algorithm, which has lower overhead due to the communication but spends more time to initiate the communication.

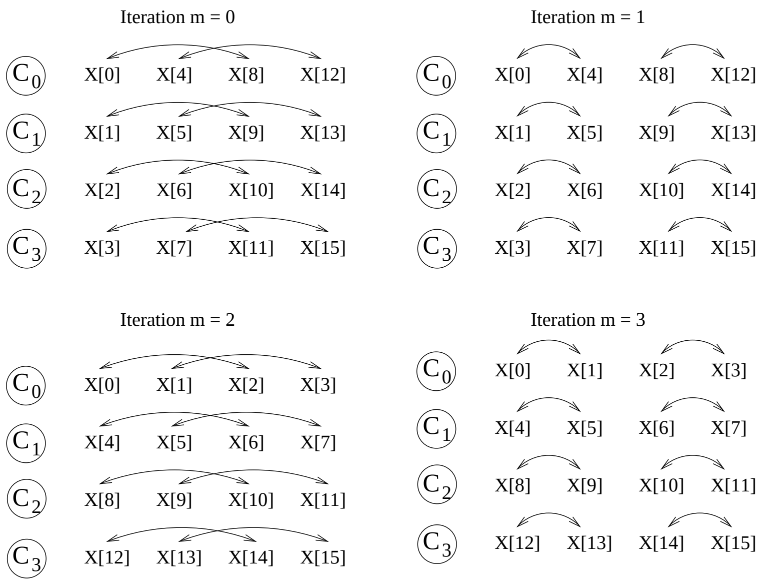


Figure 4.4. Transpose algorithm for a 16-point FFT on four cores.

4.1.2. Jacobi Kernel

Stencil computation represents a common kernel for engineering applications including multimedia processing, quantum dynamics, and electromagnetics [57]. The performance optimization of computations has been extensively studied and several code transformations have been developed to improve data locality in the calculations [58–60].

To illustrate the effect of the loop transformations, we consider 2-dimensional Jacobi code, which updates the contents of grid elements by using neighbor elements in two consecutive loop iterations. Our parallel implementation is simply parallelization of two loops in the calculation (Figure 4.5).

The loop transformations used in our performance-reliability analysis are *loop unrolling* (Figure 4.6) which reduces the loop overhead with smaller number of iter-

```
procedure jacobi(A, B, n)
  for $t \leftarrow 1$ to iterations do
    for $i \leftarrow 0$ to $n - 1$ do
      for $j \leftarrow 0$ to $n - 1$ do
        $B[i, j] \leftarrow (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) \times 0.2$
      end for
    end for
    for $i \leftarrow 0$ to $n - 1$ do
      for $j \leftarrow 0$ to $n - 1$ do
        $A[i, j] \leftarrow B[i, j]$
      end for
    end for
  end for
```

Figure 4.5. 2-D Jacobi code.

ations, *loop fusion* (Figure 4.7) which replaces multiple loops with a single one, and *loop interchange* (Figure 4.8) which exchanges the order of two iteration variables. We consider 512x512 grid size for Jacobi computations in our experiments.

4.1.3. Water Simulation

The Water Simulation is N-body molecular dynamics application, which simulates forces and potential energy of water molecules. The method of molecular dynamics is widely used to analyze the atomic structures in materials science, biochemistry, and biophysics [61].

The SPLASH-2 benchmark suite [39] has two parallel *pthread* versions of water simulation application: *Water-nsquared* and *Water-spatial*. Water-nsquared is an improved version of the original Water program in SPLASH [62], and computes force and potentials in $O(n^2)$. The computation is performed over a number of time steps by using a predictor-corrector method. It improves the original version by using a locking strategy in the updates, which stores a local copy of the particle accelerations as it

```
procedure $unrolling(A, B, n)$
  for $t \leftarrow 1$ $to$ $iterations$ do
    for $i \leftarrow 0$ $to$ $n-1$ do
      for $j \leftarrow 0$ $to$ $n-1$ $by$ $4$ do
        $B[i][j] \leftarrow (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) \times 0.2$
        $B[i][j+1] \leftarrow (A[i][j+1] + A[i-1][j+1] + A[i+1][j+1] + A[i][j] + A[i][j+2]) \times 0.2$
        $B[i][j+2] \leftarrow (A[i][j+2] + A[i-1][j+2] + A[i+1][j+2] + A[i][j+1] + A[i][j+3]) \times 0.2$
        $B[i][j+3] \leftarrow (A[i][j+3] + A[i-1][j+3] + A[i+1][j+3] + A[i][j+2] + A[i][j+4]) \times 0.2$
      end for
    end for
    for $i \leftarrow 0$ $to$ $n-1$ do
      for $j \leftarrow 0$ $to$ $n-1$ $by$ $4$ do
        $A[i][j] \leftarrow B[i][j]$
        $A[i][j+1] \leftarrow B[i][j+1]$
        $A[i][j+2] \leftarrow B[i][j+2]$
        $A[i][j+3] \leftarrow B[i][j+3]$
      end for
    end for
  end for
```

Figure 4.6. Unrolled 2-D Jacobi code.

```
**procedure** $fusion(A, B, n)$
  **for** $t \leftarrow 1$ *to* $iterations$ **do**
    **for** $i \leftarrow 0$ *to* $n - 1$ **do**
      **for** $j \leftarrow 0$ *to* $n - 1$ **do**
        **if** $(j == 0)$ **then**
          $B[i, j] \leftarrow (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) \times 0.2$
        **else if** $(j == n - 1)$ **then**
          $A[i][j-1] \leftarrow B[i][j-1]$
        **else**
          $B[i][j] \leftarrow (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) \times 0.2$
          $A[i][j-1] \leftarrow B[i][j-1]$
        **end if**
      **end for**
    **end for**
  **end for**
```

Figure 4.7. Fused 2-D Jacobi code.

```
**procedure** $interchange(A, B, n)$
  **for** $t \leftarrow 1$ *to* $iterations$ **do**
    **for** $j \leftarrow 0$ *to* $n - 1$ **do**
      **for** $i \leftarrow 0$ *to* $n - 1$ **do**
        $B[i, j] \leftarrow (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) \times 0.2$
      **end for**
    **end for**
    **for** $j \leftarrow 0$ *to* $n - 1$ **do**
      **for** $i \leftarrow 0$ *to* $n - 1$ **do**
        $A[i, j] \leftarrow B[i, j]$
      **end for**
    **end for**
  **end for**
```

Figure 4.8. Interchanged 2-D Jacobi code.

calculates and sends to the shared copy at the end. Water-spatial is a more efficient method and uses an $O(n)$ algorithm. It divides the molecules into a grid of cells (processors) and employs spatial locality to calculate inter-molecular forces by considering only molecules in nearby cells. Our experiments for water simulation are conducted for Water-nsquared and Water-spatial programs by considering 512 water molecules and default parameters provided by the SPLASH-2 benchmark.

4.2. Experimental Results

To evaluate performance and reliability of the multithreaded applications by measuring their execution times and thread vulnerability factors respectively, we execute our target applications in the Simics toolset [3]. We measure execution clock cycles and calculate TVF of each thread in target applications with respect to L1 caches, register file, and ALU units. Unless otherwise stated, we use equal local and remote vulnerability weights ($w_L = w_R = 0.5$). Our experiments are conducted by using the same number of threads as the core counts (assign one thread per core) in the target architecture.

Table 4.1 presents the vulnerability and execution time values of our benchmark applications for two, four, eight, and sixteen-thread (one thread per core) execution scenarios.

One can see from the *FFT* results that two parallel algorithms have different TVF values. As mentioned in Section 2.3, the *binary-exchange* algorithm has more communication whereas the *transpose* algorithm does not have much thread communication after transpose operation. However, the *transpose* algorithm spends its execution cycles to transposition which requires more local resource usage. The effect of this distinct characteristics of the algorithms can be observed on the local vulnerability values as well as on the remote vulnerability values. LVF values for *transpose* algorithm, which requires more computation, are larger than LVF values for *binary-exchange*, especially for register resources (e.g., binary-exchange 1.27, transpose 1.56 for the 4-core case, binary-exchange 2.53, transpose 3.13 for the 8-core case). Since these parallel algo-

Table 4.1. TVF values and execution time of our benchmark applications.

			FFT		WATER		JACOBIAN			
			binary-exchange	transpose	nsquared	spatial	original	fused	inter-changed	unrolled
2 core	LVF	ALU	0.61	0.61	0.46	0.48	0.92	0.95	0.92	0.96
		Register	0.62	0.76	0.84	0.91	0.51	0.51	0.51	0.51
		Memory	0.55	0.53	1.26	1.32	1.00	1.14	0.99	1.00
	RVF	ALU	0.44	0.45	0.43	0.42	0.90	0.91	0.90	0.94
		Register	0.37	0.38	0.51	0.52	0.40	0.41	0.40	0.39
		Memory	0.29	0.23	0.58	0.64	0.90	0.92	0.86	0.90
	Execution time		32.61	33.35	9.78	8.27	13.52	10.87	13.85	9.19
4 core	LVF	ALU	1.21	1.22	0.93	0.96	1.83	1.90	1.83	1.92
		Register	1.27	1.56	1.42	1.46	0.99	1.01	1.03	0.90
		Memory	1.00	1.06	2.36	2.69	2.00	2.28	1.98	2.00
	RVF	ALU	0.82	0.84	0.85	0.83	1.81	1.82	1.81	1.89
		Register	0.68	0.66	1.05	1.03	0.91	0.86	0.91	0.90
		Memory	0.52	0.41	1.01	1.20	1.82	1.84	1.73	1.82
	Execution time		16.32	16.68	5.46	4.49	6.80	5.42	6.93	4.63
8 core	LVF	ALU	2.43	2.45	1.86	1.92	3.67	3.80	3.67	3.84
		Register	2.53	3.13	2.63	2.77	2.02	1.99	1.89	2.06
		Memory	2.00	2.12	4.40	5.05	4.03	4.55	3.99	4.03
	RVF	ALU	1.66	1.63	1.69	1.64	3.66	3.64	3.63	3.80
		Register	1.16	1.19	2.08	2.05	1.78	1.87	1.87	1.85
		Memory	0.87	0.75	1.73	2.19	3.68	3.67	3.49	3.68
	Execution time		8.17	8.35	3.29	2.59	3.49	2.78	3.56	2.42
16 core	LVF	ALU	4.87	4.89	3.73	3.83	7.33	7.60	7.33	7.67
		Register	5.14	6.35	4.97	5.29	3.96	3.94	4.00	3.99
		Memory	2.86	4.23	8.55	9.73	8.13	9.08	8.06	8.13
	RVF	ALU	3.26	3.25	3.36	3.28	7.28	7.29	7.28	7.61
		Register	2.22	2.18	4.19	4.13	3.78	3.78	3.78	3.73
		Memory	1.64	1.43	3.15	4.33	7.47	7.34	7.10	7.46
	Execution time		4.09	4.18	2.35	1.69	1.99	1.51	2.02	1.45

rithms have different thread communication patterns, RVF values, which result from communication of threads in a multithreaded application, differ for these algorithms as well. The *binary-exchange* algorithm, which requires thread communication for more iterations, has larger RVF (especially for memory resource) values than the *transpose* algorithm which has thread communication only for one phase (before transposition) of the algorithm (e.g., binary-exchange 0.87, transpose 0.75 for the 8-core case and binary-exchange 1.64, transpose 1.43 for the 16-core case). On the other hand, the execution time values are not so distinct while *binary-exchange* performs well for all core cases.

Figure 4.9 represents the contribution of remote threads to the RVF value of the target thread for 4-core, 8-core, and 16-core execution cases of the binary-exchange algorithm. The darker the grid is, the higher the contribution is. Although there are some values for each grid, the smaller values are represented as no contribution. For instance, the value in the grid representing the contribution of $Thread_4$ to remote vulnerability of $Thread_6$ (the black colored) is 13437.5; while the value in the lower grid (the white colored), which represents the contribution of $Thread_3$, is 2.87. These values are measured before normalization which divides these values by the number of remote counts.

We exclude the contribution of the main thread which has complete input data and sends the specific portion to the related threads at the beginning of the execution. In the 8-core execution, which has the pattern of combination illustrated in Figure 4.2; $Thread_0$ reads data from $Thread_4$, $Thread_2$, and $Thread_1$ during first, second, and third iterations respectively. The impact of this pattern on RVF values is demonstrated in Figure 4.9, such that RVF value of $Thread_0$ for the binary-exchange algorithm is constructed by the values written by $Thread_1$ and $Thread_2$. Since $Thread_0$ reads data from the main thread in the first iteration ($m = 0$), the contribution of $Thread_4$ is not visible for this thread. However, one can see that $Thread_0$ reads remote data written by $Thread_2$ and $Thread_1$ in the following iterations ($m = 1$ and $m = 2$ respectively). The same scenario is valid for 4-core and 16-core execution cases. Since the data exchange occurs at the first $d = logp$ iterations and data at the first iteration is always read from the main thread; each thread communicates with $d-1$ threads which contribute its RVF value. While these threads are $Thread_1$ in the 4-core execution; $Thread_1$ and $Thread_2$ in the 8-core execution; $Thread_1$, $Thread_2$ and $Thread_4$ in the 16-core execution for $Thread_0$.

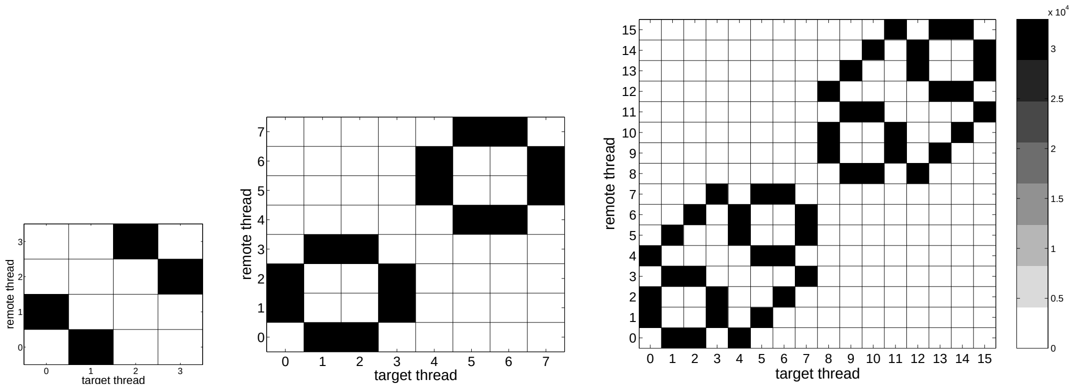


Figure 4.9. RVF contribution of each thread for binary-exchange algorithm.

On the other hand, RVF values of the threads in the transpose algorithm do not have such a certain pattern as shown in Figure 4.10, which represents the communication pattern of threads in the transpose algorithm. Since the values are very small representing lower communication between threads in the algorithm, their difference becomes diverse in the grid display. While the black colored grid in the second column for 8-core execution case represents the value 12.42, the gray colored grid located just below is 2.36.

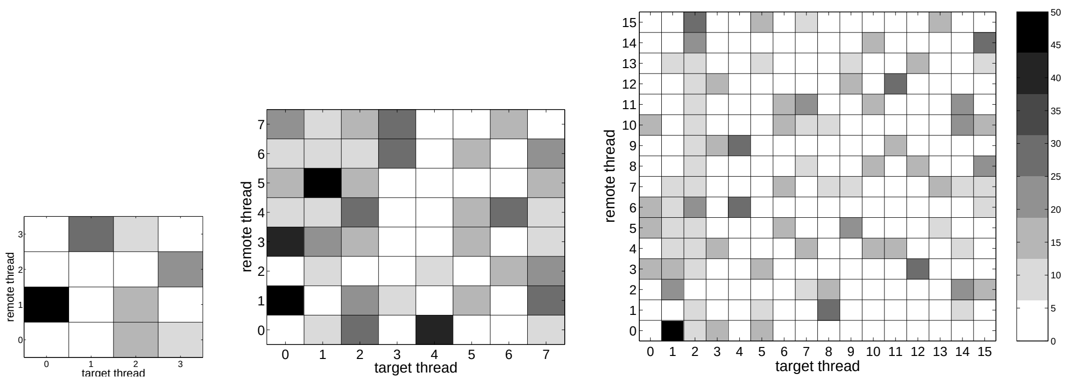


Figure 4.10. RVF contribution of each thread for transpose algorithm.

The results of the *Jacobi* kernel reveal that loop fusion and loop unrolling improves the performance of the code by eliminating loop overhead. On the other hand, loop interchange has no significant effect on the execution time. While the *fusion* increases the vulnerability values (for both local and remote) which results in a tradeoff between performance and reliability, the *loop unrolling*, which has the largest performance gain, does not affect the vulnerability if compared with the original version. The unrolled version has similar vulnerability results, the values are even the same for some cases (1.00, 2.00, 4.03 for memory resources in the 2-core, 4-core, and 8-core executions respectively).

The execution time values of the *water* simulation application are significantly different for two parallel versions. The difference becomes more clear as the core count rises due to the communication overhead of the *nsquared* algorithm. Although there is more communication overhead in the *nsquared* algorithm, RVF values that result from remote read operations are larger in the *spatial* algorithm. Since cache miss rates are smaller for *nsquared* version due to the lack of spatial locality, LVF values with respect to memory (cache) resources are smaller as well. Therefore, RVF values calculated by using local vulnerability become larger for the *spatial* version which demonstrates that the faster *spatial* algorithm is more vulnerable to soft errors with larger vulnerability values (e.g., nsquared 0.58, spatial 0.64 for the 2-core case and nsquared 1.01, spatial 1.20 for the 4-core case).

Since LVF values have been counted for RVF calculation (RVF value of one thread is calculated by adding TVF values of threads communicating with the target thread), only RVF values are considered as the reliability metric by ignoring very small LVF values that are not counted for RVF calculation. We also examine the vulnerability values for the resource which has the largest difference between different cases. For instance, RVF values with respect to ALU and register resources are not significantly different for two FFT algorithms (e.g., 2.21% higher in transpose algorithm in the 2-core execution for the ALU resource, 3.18% higher in binary-exchange algorithm in the 4-core execution for the register resource). On the other hand, the change in RVF values with respect to memory (cache) resources is large enough to evaluate reliability measure in tradeoff analysis. To evaluate performance-reliability tradeoff between different versions of our applications, we demonstrate the change (in percentage) for both vulnerability values and execution time of the applications (see Figure 4.11, Figure 4.12, and Figure 4.13). The plots provide the behavior of the applications in terms of both reliability and performance if we choose a specific version.

Figure 4.11 reveals that if we use the *transpose* algorithm instead of the *binary-exchange* algorithm, we should sacrifice approximately 3% performance but gain 20% reliability with lower RVF values in the 2-core case. This observation is valid for any number of cores. We can say that one may prefer the *transpose* algorithm rather than the *binary-exchange* algorithm to work with much higher reliability by accepting little amount of performance loss.

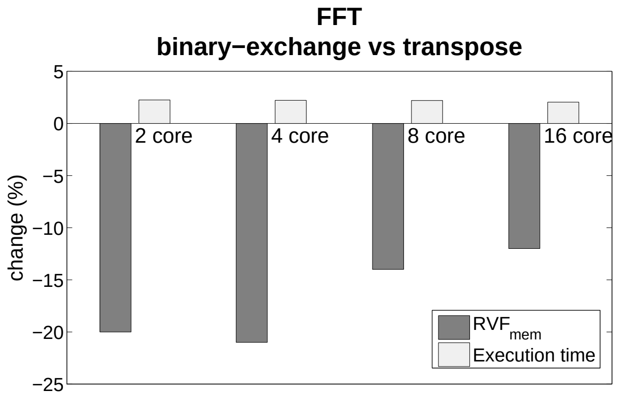


Figure 4.11. The change in percentage of RVF values and execution time for versions of FFT.

There is similar observation for the versions of the *Jacobi* codes, in that case the choice is clear for the performance. Since the *loop unrolling* gives the best performance among the other loop transformations, we evaluate the vulnerability and performance change if one prefers loop unrolling instead of other versions. Figure 4.12 presents the change in both RVF and execution time values for loop unrolling and other code versions (original, fused, and interchanged respectively). The vulnerability values do not differ much for each case. Specifically, the values are smaller than 5% even there is almost no change if we compare the original version with the unrolled one. On the other hand, the performance gain is obvious if the unrolled version is used. For instance, if one employs the unrolled version instead of the interchanged one, about 50% speedup is possible by sacrificing only 5% reliability.

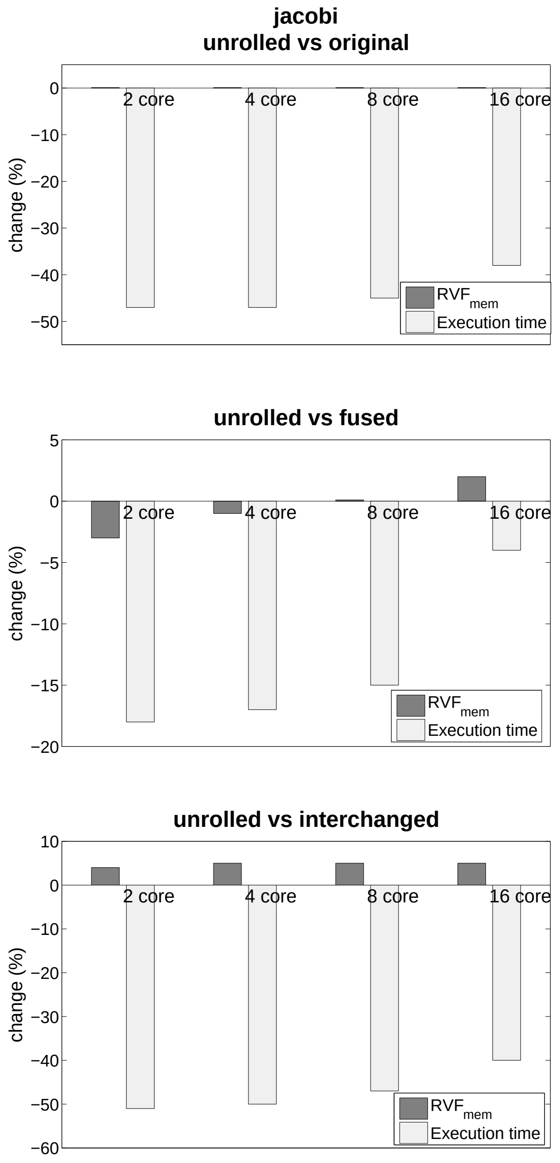


Figure 4.12. The change in percentage of RVF values and execution time for loop transformations of Jacobi code.

Although the choice is clear for two applications in terms of performance and reliability, *water* simulation application reveals more interesting results which yield a tradeoff between performance and reliability concerns. Figure 4.13 demonstrates this tradeoff between two different water simulation algorithms. While *spatial* has larger performance advantage, *nsquared* is more reliable with similar rates. For the 8-core execution, one should trade 27% performance gain with 21% reliability loss if he selects *spatial* algorithm. We cannot easily conclude that one version satisfies both performance and reliability constraints. While the safety-critical systems with higher reliability needs may prefer the *nsquared* version by sacrificing some performance, the systems for which the performance is the most crucial factor may opt for the *spatial* version, which has higher performance but is much more vulnerable to hardware errors. However, it is difficult to trade the vulnerability with performance for the systems which do not have evident performance and reliability needs.

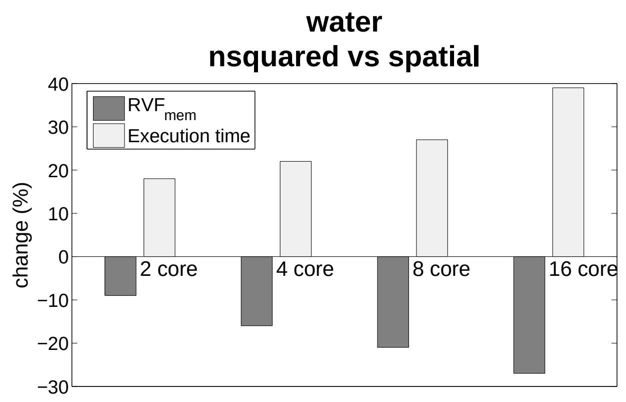


Figure 4.13. The change in percentage of RVF values and execution time for versions of water simulation.

5. RELIABILITY-AWARE CORE PARTITIONING FOR MULTICORE ARCHITECTURES

Executing multiple applications concurrently is an important way of utilizing the computational power provided by emerging chip multiprocessor (CMP) architectures. However, this multiprogramming brings a resource management and partitioning problem, for which one can find numerous examples in the literature. Most of the resource partitioning schemes proposed to date focus on performance or energy centric strategies.

In this chapter, we explore reliability-aware core partitioning strategies targeting CMPs based on performance and reliability characteristics of multi-application workloads. We first present an overview on core partitioning problem for multicore systems and related work in Section 5.1. Section 5.2 presents the system model targeted by our work and core partitioning schemes for multithreaded applications. The details and results from our experimental analysis are presented in Section 5.3.

5.1. An Overview on Core Partitioning

Chip multiprocessors (CMPs) are fast replacing conventional single-core machines in most application domains [6]. Major vendors have already shifted to CMP technology for their laptop, desktop and server processing units. CMPs provide both power and scalability gains when compared with single-core designs. CMPs are also preferable from a parallelism perspective as they enable thread level parallelism, in addition to instruction level parallelism.

The degree of parallelism extracted from an application is the primary factor that determines the potential performance gains that can be achieved from a multicore architecture. Intel has already prototyped 80-core Teraflop machine [63] and similar/larger designs have been discussed [64]. Since emerging CMPs target to provide large number of cores, executing multiple multithreaded applications at the same time is an efficient way of utilizing CMP architectures.

Core partitioning is a critical problem in the context of CMPs, which deals with how many cores to allocate to each application running simultaneously. A number of recent studies has addressed partitioning and/or scheduling related issues in CMPs where they mostly target on performance speedup or quality of service (QoS) [65, 66]. Since energy consumption has become a crucial problem in recent years [67], energy-delay product [68] has also been used as an optimization metric [69]. Energy efficiency has been considered for task partitioning problem in multicore architectures [70]. Another concern in the context of CMP architectures is the system reliability [71].

The easiest way to partition available cores is to allocate equal number of processors to each application in the workload. However, this strategy becomes unfeasible for the cases having different requirements. While the approach yields acceptable results for the workloads that highly require performance, it does not make sense to increase the number of allocated cores if we consider reliability related issues. Recent studies show that CMP architectures become more vulnerable as the number of cores increases [72, 73]. Since the reliability of the system decreases as the number of allocated cores increases, the safety-critical applications requiring high reliability cannot get benefit from additional cores in a CMP system. If performance gains obtained by additional cores are not very high, even the systems that do not have very high reliability concerns can prefer not to use the core and work with fewer cores than available to provide higher reliability by sacrificing little performance.

Resource partitioning strategies have been widely studied for parallel architectures including CMP systems. While some research aim to maximize performance, recent work focuses on both performance and power related issues. The policies also deal with partitioning of different resources in the system among threads of multiple multithreaded application workloads.

Liu *et al.* [74] proposes a processor partitioning strategy for distributed-memory multiprocessor systems. Their work deals with mesh-connected systems running multiple jobs concurrently and aims to improve system performance by minimizing communication cost. The technique is based on creating partitions for each job such that communication distance within each partition to be as small as possible. Moreover, a thread mapping strategy which tries to find out an optimum way to map threads of an individual job to the processors allocated to that job has been proposed. The strategy aims higher performance by considering communication among threads and thread adjacency in the network.

A Clustered Multi-Threaded (CMT) processor based on SMT architectures has been presented in [75]. The CMT approach partitions the execution units among processors in CMP architectures such that the application threads, which are assigned to the processor cores, utilize system resources. Thus the computation-intensive threads benefit from the resources and lightweight threads do not cause the reduction in utilization. Raasch *et al.* [76] also investigates the performance impact of resource partitioning polices on SMT architectures. Both static and dynamic approaches which deal with partitioning of microarchitectural resources have been examined and compared by presenting weighted-speedup values of multiple threaded workloads.

Several studies have focused on cache partitioning in CMP platforms [77–79]. Ravi [77] proposes a cache framework which provides priority-based management of shared cache structures among heterogeneous threads. The framework, constructed by considering priority assignment to applications according to their memory access behavior, allows higher priority applications to use more cache lines and lower priority applications to use fewer cache lines. Chang and Sohi [78] presents a cooperative cache partitioning strategy which considers multiple requirements including thrashing avoidance, fairness improvement, priority support and QoS guarantee. Power related issues as well as performance metrics have been considered for shared cache partitioning problem in chip multiprocessors [79]. The power-aware cache partitioning strategy based on a power-aware cache design improves energy efficiency and conducts a performance-power tradeoff analysis by working together with a performance-aware partitioning technique.

Energy-aware core partitioning techniques have been studied as well as strategies for cache partitioning on multicore architectures [69, 80]. Ding *et al.* [69] proposes a dynamic core partitioning scheme of which main objective is to decrease energy consumption while maintaining high performance. To be able to evaluate the efficiency of the scheme satisfying both requirements, energy-delay product (EDP) metric [68] which considers both execution time and energy consumption has been used. Moreover, weighted energy-delay product gain (W-EDPG) metric which represents the average improvement in EDP values of all multithreaded applications executing concurrently has been proposed to compare different schemes.

Srikantaiah *et al.* [81] examines processor partitioning and cache partitioning to be able to build an integrated partitioning strategy. The algorithm proposed by the scheme combines processor partitioning and cache partitioning iteratively, and aims to maximize fair speedup metric and maintain quality of service metric. The efficiency of different schemes including equal partitioning, implicit processor partitioning, proposed processor only partitioning, proposed cache only partitioning and integrated partitioning has been compared to be able to emphasize the effectiveness of the proposed strategy.

5.2. Reliability-Aware Core Partitioning

We conduct a core partitioning analysis for multiple multithreaded applications executing on the same CMP architecture. We propose and evaluate *reliability-aware* core partitioning schemes for multicore architectures [82]. We use TVF for the evaluation of the multithreaded applications' reliability while analyzing various core partitioning schemes. Our schemes consider the reliability of the system, which has a performance bound to satisfy the quality of service. The goal of our reliability-oriented partitioning scheme is to maximize the reliability of the system while distributing available cores to the multithreaded applications. Another scheme that we propose considers both the system performance and reliability, and partitions the residual cores (i.e., the cores remaining after satisfying the performance bounds) to maximize the value of the combined metric defined as Vulnerability-Delay Product (VDP). We also perform a simulation study with various workloads consisting of multiple multithreaded applications to evaluate our proposed partitioning schemes. To quantify system reliability, TVF metric is used as demonstrating the vulnerability of the multithreaded applications running on architectures having various number of cores. We compare the weighted speedup, the weighted reliability loss and the weighted vulnerability-delay product gain on the system for different partitioning schemes.

We consider a chip multiprocessor (CMP) system consisted of p cores and s applications (each of which can be multithreaded) running on the system, as illustrated in Figure 5.1.

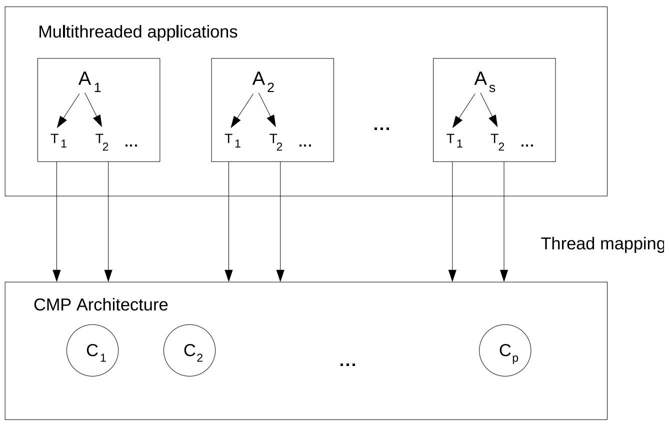


Figure 5.1. System architecture.

Each of these applications (i.e., its threads) can be mapped to a subset of available cores; but, no core is shared by threads that belong to different applications. In our experiments, we assume one-to-one assignment between threads and cores. Since the total number of threads cannot exceed the number of cores, we have the following constraint:

$$\sum_{i=1}^{s} p_i \leq p,$$

where p_i represents the number of threads for application i. That is, p_i cores of the system are reserved for p_i threads of the i^{th} application. Our goal is to determine the number of threads (the number of cores reserved) of each application running on the system.

While most of recent CMP-based studies target performance and/or energy, reliability is also becoming a first-class citizen. Motivated by this, we consider both reliability and performance to partition available cores for a set of multithreaded applications' threads. Although performance is measured by various metrics [78] which are based on execution cycles of an application, there has not been a common software centric metric for reliability.

The primary goal behind our work is to determine efficient core partitioning by determining the number of cores allocated to each application with the restriction of one application thread per core, based on performance and reliability metrics. We evaluate four different core partitioning schemes, which are *equal partitioning*, *reliability-oriented partitioning*, *performance-oriented partitioning*, *partitioning based on a combined metric*.

5.2.1. Equal Partitioning

In this scheme, the cores in the target CMP are equally partitioned among all applications in the system by assuming that there is no information on performance and reliability characteristics of the applications. This strategy partitions the processor cores (p) into a number of partitions that is equal to the number of applications (s) in the system such that each application runs with equal number of threads (p/s). Throughout this study, we use this equal core partitioning scheme as our *baseline strategy* against which we compare other approaches.

5.2.2. Reliability-Oriented Partitioning

In Chapter 2, we have observed that the reliability of multithreaded applications, measured in terms of TVF, decreases as the number of cores increases (see Figure 2.9

for an example). Consequently, if the reliability is the main concern, one needs to use as fewer cores as possible. On the other hand, there is an execution time bound for each application to be satisfied in a system having multiple applications. Many applications that target emerging CMPs require a guarantee of a certain level of performance which is referred as Quality of Service (QoS) [66]. In this scheme, we specify the execution time to be satisfied for each application and allocate the number of cores in the CMP architecture according to this minimum performance level requirement. Given that system has s different multithreaded applications and p processor cores, when each application i has p_i threads running on p_i cores, the total number of cores allocated to the workload is equal to $k = \sum_{i=1}^{s} p_i$. To maximize system reliability, we do not use remaining $(q = p - k)$ cores.

5.2.3. Performance-Oriented Partitioning

After allocating the processor cores by considering execution time bound on each application, this scheme attempts to achieve the highest performance gains by using remaining cores in the system, and is illustrated in Figure 5.2. Consider an 8-core CMP architecture and two multithreaded applications to be executed in the system. Normalized execution time values of the applications under various number of threads are given in Figure 2.3.

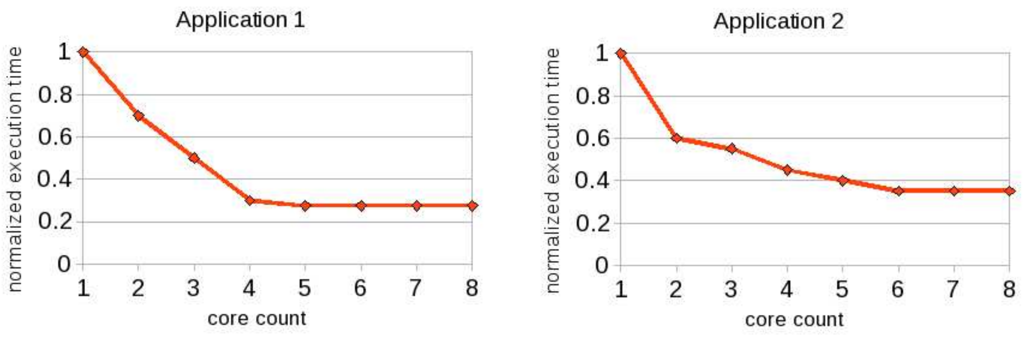


Figure 5.2. Example execution time/core count behavior of two applications.

Let us assume that the (normalized) execution time bounds are 0.5 and 0.6 time units for $Application_1$ and $Application_2$, respectively. As a consequence, we need to allocate at least 3 cores to $Application_1$ and 2 cores to $Application_2$. Then, we can distribute 3 remaining cores one by one by considering a workload wide performance metric (e.g., weighted speedup which represents the sum of entire applications' speedup values [83]). When additional core's effect on each application is examined, it is clear that allocating one more thread of $Application_1$ brings more speedup. If we give one more thread to $Application_1$ by increasing its thread number from 3 (running on 3 cores) to 4 (running on 4 cores), its execution time decreases from 0.5 to 0.3 time units. On the other hand, the execution time of $Application_2$ decreases only by 0.05 time units (0.6 to 0.55) if it is executed by 3 threads instead of 2 threads. After allocating a free core to $Application_1$, we look at the second free core's allocation decision. Although $Application_1$ scales well up to that point, its performance is not affected very much beyond 4 threads. In comparison, the execution time of $Application_2$ decreases continuously until 6 thread execution. Therefore, the remaining 2 cores in the system should be allocated to $Application_2$'s threads to achieve the maximum weighted speedup.

5.2.4. Partitioning Based on a Combined Metric

The goal behind this scheme is to distribute the remaining cores (after satisfying the performance bounds) across applications such that the value of a combined performance-reliability metric is maximized.

To achieve both high performance and reliability in the system, we need a combined metric which includes both execution cycles representing performance and TVF metric representing the vulnerability of the system to the errors. One of the first metrics that can be come up with is the product of the execution time and TVF as in Energy-Delay product [68]. This combined metric, referred to as Vulnerability-Delay product (VDP) in the rest of this chapter, can be calculated as follows:

$$\begin{aligned} VDP &= Vulnerability \times Delay \\ &= TVF \times Execution\ cycles. \end{aligned} \tag{5.1}$$

Both TVF and execution cycles should be minimized to improve the vulnerability-delay product of an application. As the number of threads in the application increases, execution cycles decrease but the TVF value increases conversely. Therefore, we should carry out a tradeoff analysis which requires to find an optimum point to be able to reduce the combined metric.

```
procedure $partition(n, workload)$
  for each $n$ cores to allocate do
    initialize maximum gain
    for each application in the workload do
      calculate gain for an additional core
      if there is no gain then
        continue with other application
      else if gain $>$ maximum gain then
        maximum gain $\leftarrow$ gain
      end if
    end for
    if there is no gain then
      terminate partitioning
    else
      allocate the core to the application with the maximum gain
    end if
  end for
```

Figure 5.3. Basic partitioning algorithm.

Figure 5.3 represents the algorithm skeleton used in our partitioning schemes. The partitioning strategies (including reliability-oriented, performance-oriented and the scheme maximizing both reliability and performance) aim to get the largest gain with allocation of one additional core. The only difference between the strategies is the target metric to optimize. For instance, performance-oriented scheme considers performance gain by using execution time vs core count curve (as in Figure 5.2), whereas the scheme maximizing both performance and reliability tries to maximize vulnerability delay product metric by using VDP vs the core count curve for the applications in the workload.

5.3. Experimental Evaluation

To evaluate our core partitioning strategies, we execute benchmark applications (from PARSEC and SPLASH-2) for different number of threads running on different number of cores by mapping one thread onto one core. Our experiments include 16 different executions for each application by varying the number of cores from 1 to 16, simulated on Simics simulator.

Execution clock cycles are measured and Thread Vulnerability Factor of each thread in target applications are calculated with respect to L1 caches, register file and ALU units. Since the combination of those values for different resources is not much feasible, we consider the highest value as the value of TVF. The local vulnerability factor (LVF) values have been counted in remote vulnerability factor (RVF) calculation [34], since the RVF value of a thread is calculated by adding the TVF (both LVF and RVF) values of threads communicating with target thread. The values which have not counted in RVF calculation of another thread are considered as LVF term. Since the LVF values are significantly smaller than the RVF values, we consider only the RVF values as the reliability metric, which is referred as the TVF term in this section.

Figure 5.4 plots the normalized execution clock cycles of benchmark applications for different number of cores. Most of the applications (*barnes*, *blackscholes*, *canneal*, *fmm* and *streamcluster*) except those which have heterogeneous threads, scale reasonably well as the number of cores (threads running on the cores) increases, due to their data-parallel characteristics. Two applications including *swaptions*, which has two sets of threads running different code portions for each core, and *water-spatial*, which has non-homogeneous threads having different tasks, i.e., one thread differs from the other threads, have diverse behavior. *Cholesky* application also shows different trend for larger number of cores. Since these three applications are not-so-well scalable, we call them as non-scalable applications in the rest of the chapter.

The reliability trend represented by TVF values vs core counts is shown in Figure 5.5. It can be observed that the reliability of the multithreaded applications decreases by increasing the number of cores, which is opposite to the performance increase (There are two exceptions including *swaptions* and *water-spatial* where TVF values show different behavior for some cases.) This behavior reflects the tradeoff between performance and reliability, which can be explored by analysis of plots.

Figure 5.6 presents the vulnerability-delay product which represents both relative reliability and performance of multithreaded applications. This plot is constructed by calculating the combined metric value for each core case.

5.3.1. Workload Construction

In our experiments, we construct workloads consisting of multiple multithreaded applications by considering the performance scalability of the applications. Our workloads, having 3 applications selected from benchmark suites, are composed of either scalable, non-scalable or mix of two kinds of applications. Since we have 3 applications (*swaptions*, *water-spatial* and *cholesky*) which do not scale as the number of cores increases, only one of the workloads includes only non-scalable applications. On the other hand, 10 different workloads are possible which have different combination of scalable applications. Each other workload is constructed by including mix of scalable and non-scalable applications which have either two scalable and one non-scalable applications or two non-scalable and one scalable applications. There have been a total of 56 workloads, comprising all possible combinations.

To represent workload mixes, we use letter combinations of benchmark applications denoted by ‘H’, ‘B’, ‘C’, ‘F’, ‘T’, ‘K’, ‘S’, ‘W’ letters for *barnes*, *blackscholes*, *canneal*, *fmm*, *streamcluster*, *cholesky*, *swaptions*, *water-spatial* applications respectively. For instance, ‘KSW’ stands for the workload which consists of all non-scalable applica-

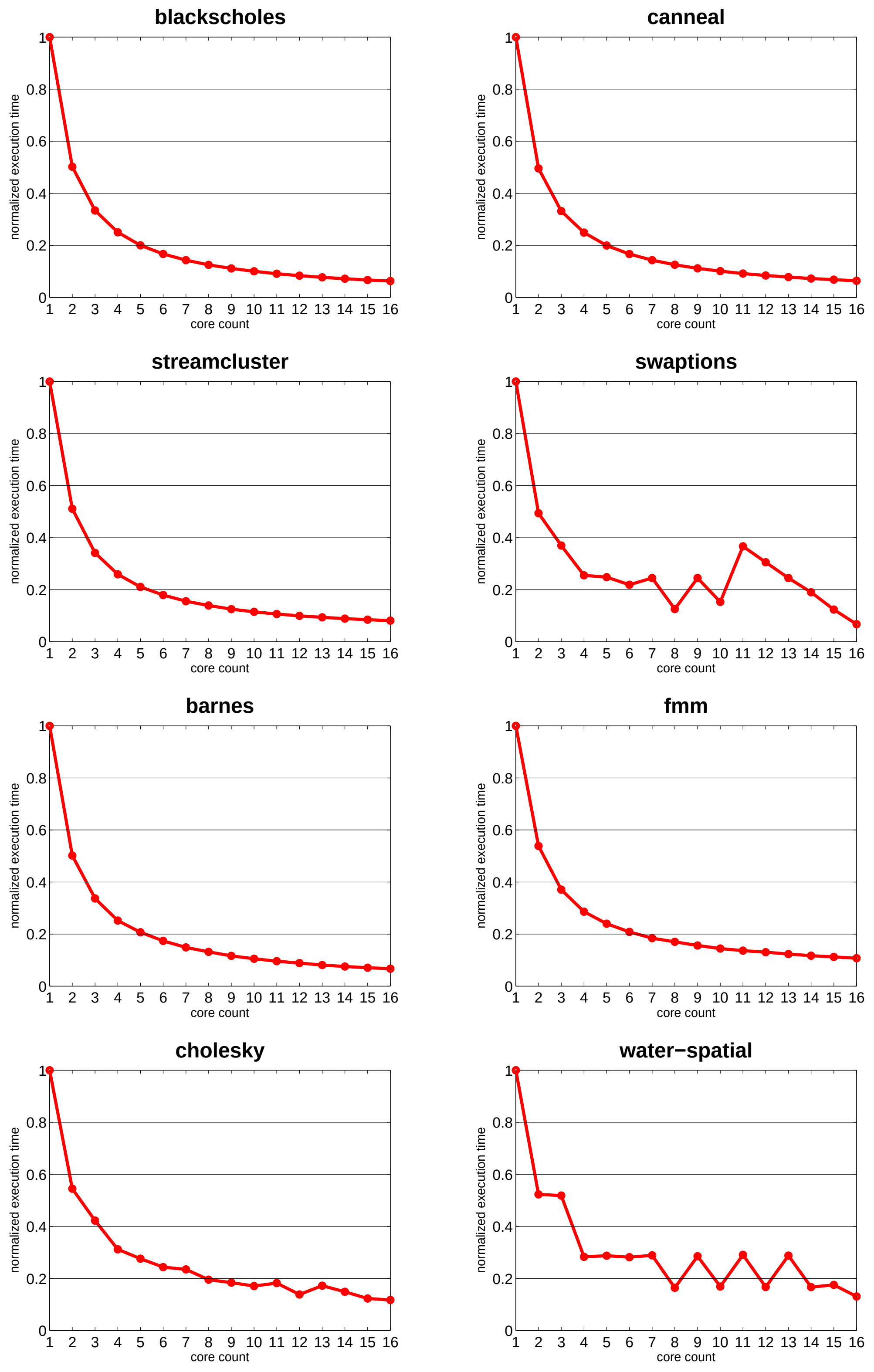


Figure 5.4. Normalized execution time of our benchmark applications.

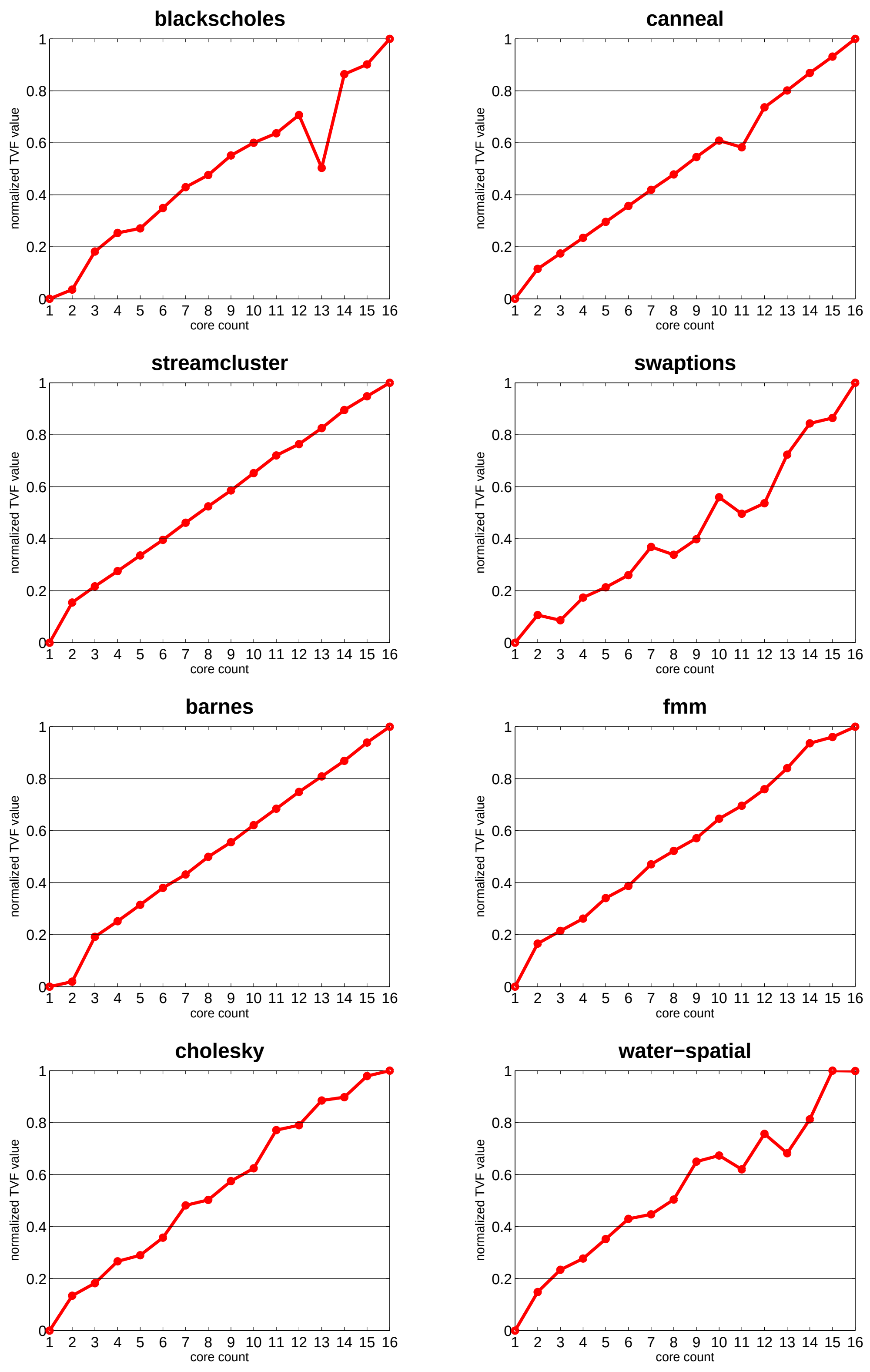


Figure 5.5. Normalized TVF values of our benchmark applications.

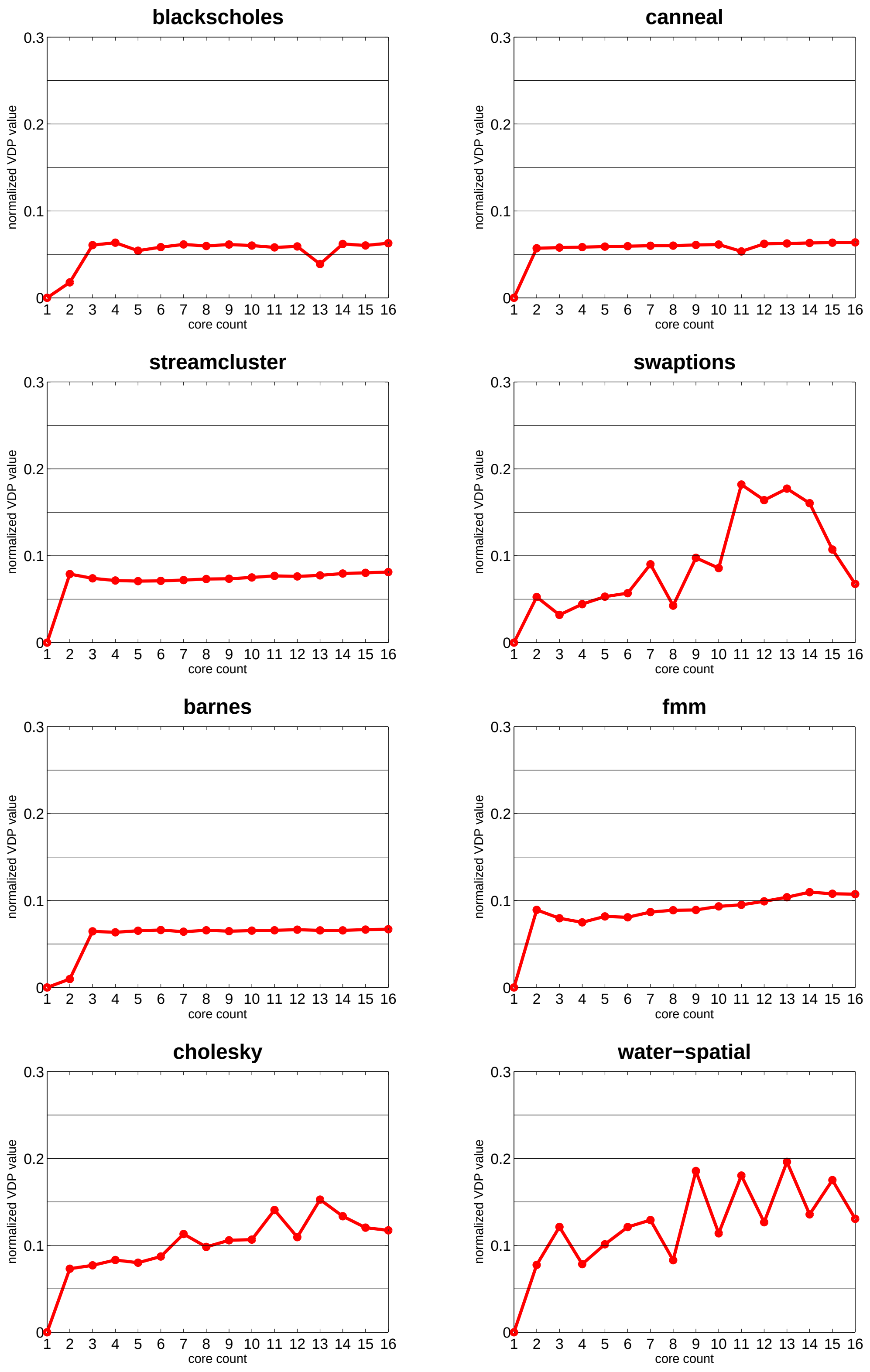


Figure 5.6. Normalized VDP values of our benchmark applications.

tions *cholesky*, *swaptions* and *water-spatial* applications; 'BCF' stands for the workload with scalable applications including *blackscholes*, *canneal* and *fmm*; 'WHT' stands for the workload with one non-scalable *water-spatial* application and two scalable *barnes*, *streamcluster* applications.

5.3.2. Performance and Reliability Metrics

The comparison of the core partitioning schemes is based on execution clock cycles and thread vulnerability factor values which represent the system's performance and reliability respectively. To evaluate the performance gain in the multi-application system, the Weighted Speedup metric (WS) of the workload for a given scheme is defined as the sum of application speedup values [81, 83],

$$WS(scheme) = \sum_{i=1}^{s} \frac{IPC_{A_i}(scheme)}{IPC_{A_i}(base)}, \tag{5.2}$$

where s is the number of co-runner applications in the system, IPC is instruction per cycle which represents measure of speedup. We consider equal core partitioning scheme as the *base* scheme for our experiments. We use normalized weighted speedup which is equal to $NWS = WS/s$, where s is the number of applications.

The reliability metric used in the core partitioning analysis is similar to weighted speedup. We define the metric *Weighted Reliability Loss* of a given scheme as the sum applications' TVF ratios,

$$WRL(scheme) = \sum_{i=1}^{s} \frac{TVF_{A_i}(scheme)}{TVF_{A_i}(base)}. \tag{5.3}$$

As in the previous case, we consider the normalized weighted reliability loss value, which is equal to $NWRL = WRL/s$. Since our objective is to increase the performance gain and reduce the reliability loss, we aim higher values for normalized weighted speedup (NWS) metric and smaller values for normalized weighted reliability loss (NWRL) metric.

We consider VDP metric to analyze our results with respect to both performance and reliability and define the metric *Weighted Vulnerability-Delay Product Gain* of a given partitioning scheme as the sum applications' VDP ratios,

$$WVDPG(scheme) = \sum_{i=1}^{s} \frac{VDP_{A_i}(base)}{VDP_{A_i}(scheme)}. \tag{5.4}$$

We consider the normalized weighted vulnerability-delay product gain value, which is equal to $NWVDPG = WVDPG/s$. Our goal is to get the highest values of NWVDPG metric which demonstrates the efficiency of the scheme for both performance and vulnerability.

5.3.3. Evaluating Partitioning Schemes

We compose workloads each having 3 benchmark applications among the 8 applications as mentioned in Section 5.3.1; and we apply different partitioning schemes to map the available cores onto these applications' threads. After applying partitioning schemes and determining complete core partition, we compare the performance and reliability results of the alternative schemes. To analyze different core partitioning schemes, we assume that we have 21 available cores in the system and our aim is to partition these cores for 3 multithreaded applications for each case. We also assume that the QoS values are guaranteed by providing 3 cores for each application in the workloads, that is, 3 cores are assigned to each application in the workload before applying our partitioning schemes.

5.3.3.1. Performance-Reliability Tradeoff Analysis. We apply partitioning schemes for all workloads and evaluate weighted-speedup, weighted reliability loss and weighted vulnerability-delay product gain metrics with respect to equal partitioning scheme. Table 5.1 represents arithmetic, harmonic and geometric means among all workloads with different partitioning schemes. The weighted speedup, weighted reliability loss and weighted vulnerability-delay product gain values are calculated by considering all applications in the workload, and then normalized by dividing by the number of applications. While the scheme maximizing reliability achieves the highest reliability gain (as can be expected), the highest speedup is achieved by the scheme maximizing performance in average. One may conduct performance-reliability tradeoff analysis of core partitioning strategies by using performance gain and reliability loss values. If the performance is the most critical criteria for the workload, performance-oriented scheme gives the highest weighted-speedup values. However, the performance gain is not significant and simple equal partitioning may be the choice. On the other hand, the reliability-oriented scheme provides significantly better reliability loss values (60% gain to equal partitioning). It may be reasonable to employ this scheme in safety-critical workloads where some performance can be sacrificed for the sake of reliability. Moreover, the scheme maximizing both performance and reliability, which also yields better results for vulnerability, may be a good choice for the systems in which the reliability is important but smaller loss are tolerable with some performance gain. The weighted vulnerability-delay product gain (NWVDPG) may also helpful to consider both vulnerability and performance issues.

Table 5.1. NWS, NWRL and NWVDPG mean values among 56 workloads for core partitioning schemes.

Partitioning Scheme	Arithmetic mean			Geometric mean			Harmonic mean		
	NWS	NWRL	NWVDPG	NWS	NWRL	NWVDPG	NWS	NWRL	NWVDPG
Reliability Oriented	0.5036	0.4178	1.3090	0.5022	0.4160	1.2785	0.5008	0.4141	1.2501
Performance Oriented	1.0968	0.9711	1.2240	1.0946	0.9706	1.2145	1.0924	0.9701	1.2048
Based on Combined Metric	0.6338	0.4920	1.3980	0.6279	0.4866	1.3696	0.6221	0.4811	1.3425

After evaluating all workloads, we pick 10 workloads by including different combinations based on the scalability classification to use in our detailed evaluation. We include 1 workload (KSW) consisting only non-scalable (not-well scalable) applications, 3 workloads (BCT, HCF, HBF) consisting only scalable applications and 6 mixed workloads (BHS, BTS, CFW, HKW, TKS, TSW).

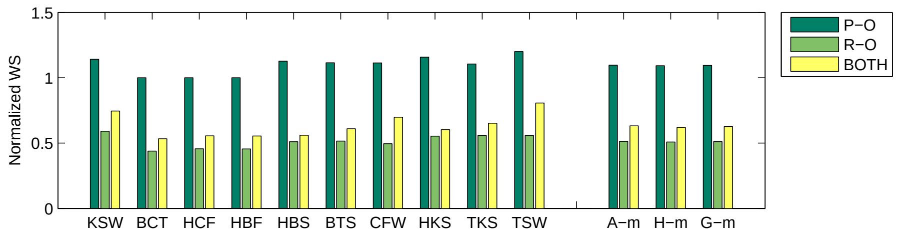


Figure 5.7. Normalized weighted-speedup values of partitioning schemes to equal core partitioning for selected 10 workloads.

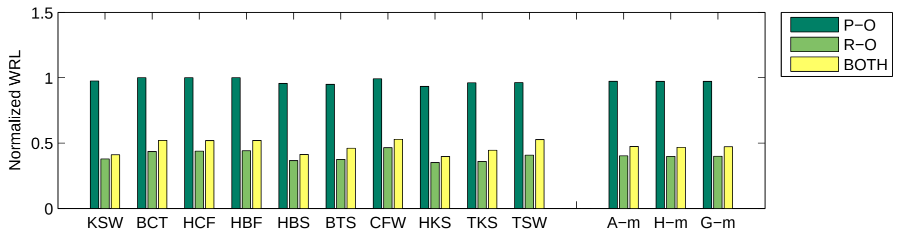


Figure 5.8. Normalized weighted-reliability loss values of partitioning schemes to equal core partitioning for selected 10 workloads.

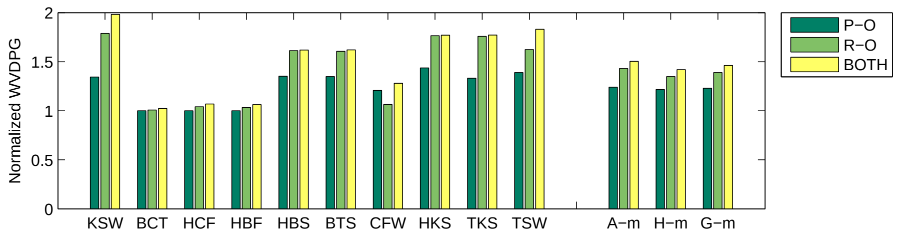


Figure 5.9. Normalized weighted-vulnerability-delay product gain values of partitioning schemes to equal core partitioning for selected 10 workloads.

While Figure 5.7 presents normalized weighted speedup for these three-application workloads with different partitioning schemes (‘P-O’, ‘R-O’ and ‘BOTH’ stand for performance-oriented scheme, reliability-oriented scheme and the scheme maximizing both respectively), Figure 5.8 plots the reliability loss values for the same workloads partitioned by the same schemes. The figures include per workload values as well as arithmetic (A-m), harmonic (H-m) and geometric (G-m) mean values. It is observed that the trend in metric values is similar to the mean values evaluated for all possible workload combinations, that is, *reliability-oriented* scheme against the scheme *based on combined metric* gives better (smaller) reliability loss (NWRL) values but reduces performance gain badly while *performance-oriented* scheme increases speed-up (NWS) by increasing reliability loss values in a relatively acceptable rate. Moreover, reliability loss values are not very different between *reliability-oriented* scheme and the scheme *based on combined metric* even the same for some cases (HKW workload).

However, the specific values exhibit a certain level of diversity for different workload sets with different scalability characteristics. The workloads consisting of only scalable programs do not exploit *performance-oriented* scheme, that is, the weighted-speedup metric equals to 1 for each case. If equal core partitioning was used instead of performing performance gain analysis, the final partition would be the same since the applications scale for the same rate. On the other hand, the workloads which have mostly not-so-well scalable applications yield better performance gain results for *performance-oriented* scheme. The performance gain against equal partitioning can reach 15%-20% (KSW, TSW respectively) for these workload types.

To evaluate the effect of our partitioning strategies on both performance and vulnerability, we present the normalized weighted vulnerability-delay product gain (NWVDPG) values in Figure 5.9. It is observed that the scheme maximizing the combined metric yields the highest gain in the vulnerability-delay product values. Since *reliability-oriented* scheme improves the vulnerability more than the performance gain that *performance-oriented* scheme provides, it results in larger NWVDPG values.

5.3.3.2. Detailed Workload Analysis. To better understand the steps of our partitioning approach, we now focus on two example workloads and analyze in detail the behavior of the different partitioning schemes regarding these workloads.

Assuming that the specified QoS values are guaranteed by providing 3 cores for each application in a three-application workload; there are residual 12 cores on a 21-core CMP system. The gain maximization analysis starts with these available 12 cores in the system.

- Our first workload, HCF, consists of 3 scalable benchmark applications including *barnes*, *canneal* and *fmm*.
  - (i) *Reliability-Oriented.* Since the vulnerability values increases as the number of cores increases for each application in the HCF workload, any of additional 12 cores are not used to maximize reliability of the system.
  - (ii) *Performance-Oriented.* Since all applications in this workload are scalable, there are performance gains for each application with an additional core. To be able to apply performance-oriented partitioning, it is necessary to examine the amount of performance gain for each application and decide allocation of additional cores with the highest gain. If we investigate the execution time plots, we can see that the structure of the curves is very similar to each other and it is not difficult to say that applications take each additional core in order. To follow allocation order, we can provide preliminary partitioning steps as follows: If we increase the core count by one (3 to 4), our performance gain is 0.0849, 0.0828 and 0.0846 in *barnes*, *canneal* and *fmm* applications respectively. Therefore, performance-oriented scheme assigns first additional core to *barnes* application. Since the execution time curves become flatter as the number of the cores increases, the gain for one more additional core (4 to 5) is smaller (0.0450) for *barnes*. However, other two applications in the workload can still benefit from an additional core which provides execution on 4 cores instead of 3. Larger gain (0.0840) comes from *fmm* application which takes the second of 12 additional cores. The algorithm continues in the same manner and ends with 7 cores assigned

for each application in the workload. The final result is the same as equal partitioning scheme.

(iii) *Based on combined metric.* To evaluate this scheme, we should look at values of our combined metric for target applications as illustrated in Figure 5.6. For the first core assignment case, we can see that the VDP (Vulnerability Delay Product) of *canneal* application increases (from 3 to 4 cores) while the value decreases for other two applications. Therefore we should assign the core to one with the highest decrease in VDP value to maximize both performance and reliability. Since VDP gain is 0.0012 and 0.0047 for *barnes* and *fmm* applications respectively, we assign the first available core to *fmm*. The second core allocation is suitable only for *barnes* application since other two applications have larger VDP values for an additional core. If we try to assign the other core to an application, we can see that the VDP value for all three applications increases for another core allocation (*barnes* and *fmm* from 4 to 5, *canneal* from 3 to 4). Therefore, in this case, the best option is not to allocate any other core to the applications in the system. We have 11 allocated cores; 4 cores to *barnes*, 4 cores to *fmm* and 3 cores to *canneal*.

- Our second workload, BTS, consists of two scalable benchmark applications (*blackscholes*, *streamcluster*) and one non-scalable application (*swaptions*).

(i) *Reliability-Oriented.* The scenario is the same as HCF workload for reliability-oriented scheme. Since the vulnerability values of three applications in the BTS workload increases as the number of cores increases, any of additional 12 cores are not used to maximize reliability of the system.

(ii) *Performance-Oriented.* The performance maximization scheme attempts mapping 12 available cores to the applications by considering the highest performance gain. The performance plots demonstrate that the most suitable application for the first core is *swaptions* since it provides the highest performance gain (from 3 to 4 core, 0.1146 decrease in execution time). The speedup is 0.0819 for *streamcluster* and 0.0836 for *blackscholes*. However, reserving one more core for *swaptions* does not cause larger gain (0.0070) than the benefit of other two applications. Therefore, we should assign the remaining 11 cores to *blackscholes* (5 cores) and *streamcluster* (6 cores) applications in order as in the first workload case. Thus we partition 21 cores such that 4 cores to *swaptions*, 8 cores to *blackscholes* and 9 cores to *streamcluster*.

(iii) *Based on combined metric.* The vulnerability delay product plots (Figure 5.6) for *blackscholes* and *swaptions* applications in this workload demonstrate that there is no VDP gain for the additional core if we assign 4 cores instead of 3 cores. Since *blackscholes* and *swaptions* has larger VDP values for 4 core counts instead of 3 core execution and only *streamcluster* application gets benefit from the additional core, the first core is assigned to *streamcluster* to minimize the combined metric. The vulnerability delay product gain continues with one more additional core for *streamcluster* application. Therefore, we can assign the next core to this application. However, its VDP keeps decreasing until 5 core execution. Beyond that point, there is no reduction on VDP values in the workload and we do not use additional cores after allocating 2 cores to *streamcluster* application. This scheme gives 3 cores to *blackscholes*, 5 cores to *streamcluster* and 3 cores to *swaptions*.

5.3.4. Sensitivity Analysis

To study the effect of several factors in our core partitioning strategies, we conduct a sensitivity analysis which includes four different initial core assignments, three different number of applications in the workload and three different number of cores in the system.

5.3.4.1. Initial Core Assignment. Since Quality of Service (QoS) [66] requirements for the applications running on CMP architectures differ, it is important to study the effect of initial core allocation which guarantees a certain level of performance for our partitioning schemes. We perform a sensitivity analysis that employs different initial core assignments. In the previous experiments, we start our partitioning with 3 core assignment for each application in the workload. We also apply our partitioning schemes by assigning initially 2 (Figure 5.10) and 4 (Figure 5.11) core for each application. Another case (Figure 5.12) is to assign 3 cores, 4 cores and 5 cores for the first, second and third application in the workload, respectively. We assume that the number of available cores is 21 and our workloads have 3 applications. We evaluate performance and reliability metrics for each case. It is observed that as the number of cores assigned initially (given as quality of service) increases performance gain (weighted speedup metric) increases as well for *reliability-oriented* scheme and the scheme *based on combined metric*. However, the reliability loss (weighted reliability loss metric) is affected badly since more allocated cores make the system more vulnerable. *Performance-oriented* partitioning scheme is not affected by the initial core assignments. If we assign different number of cores to each application, it does not affect the performance and reliability results in any partitioning scheme unless there is the same total number of initial core assignments (Figure 5.11, 5.12). The metric values are similar for 4 cores for each application and 3, 4, 5 cores for the applications in the workload, each case has 12 cores assigned initially.

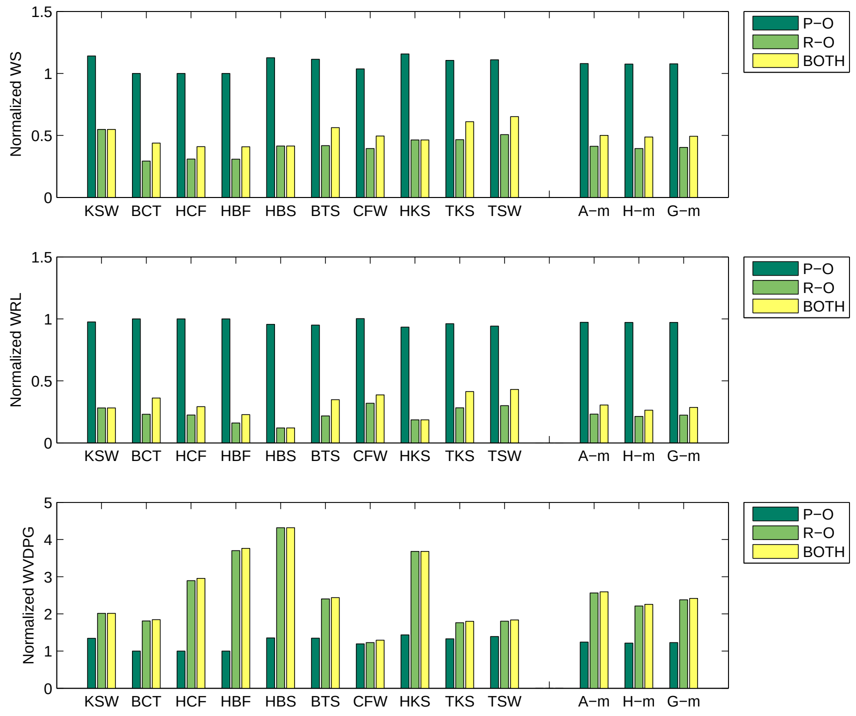


Figure 5.10. NWS, NWRL and NWVDPG values of partitioning schemes to equal core partitioning for selected 10 workloads with 2 core initial assignment.

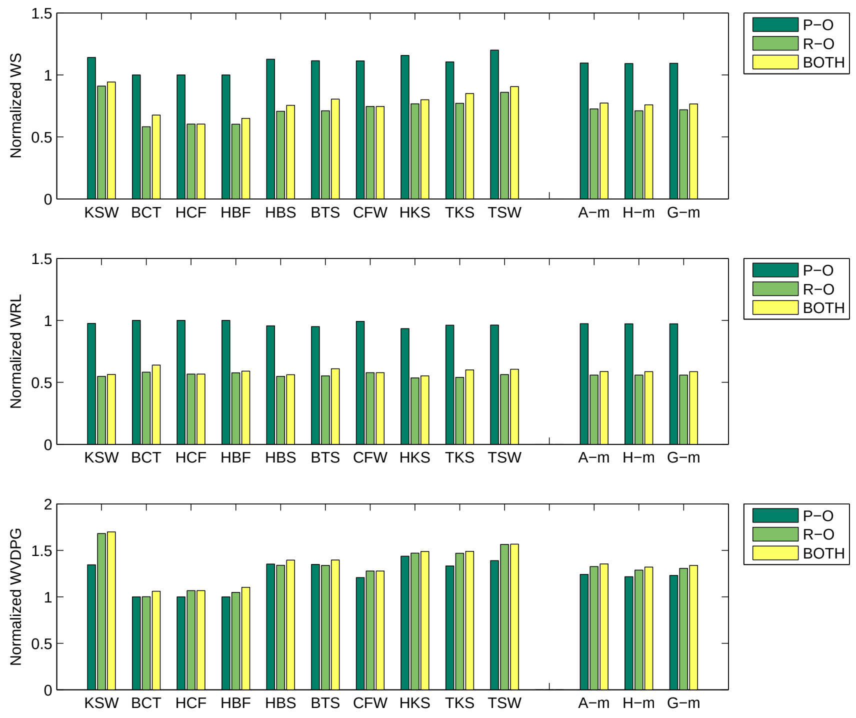


Figure 5.11. NWS, NWRL and NWVDPG values of partitioning schemes to equal core partitioning for selected 10 workloads with 4 core initial assignment.

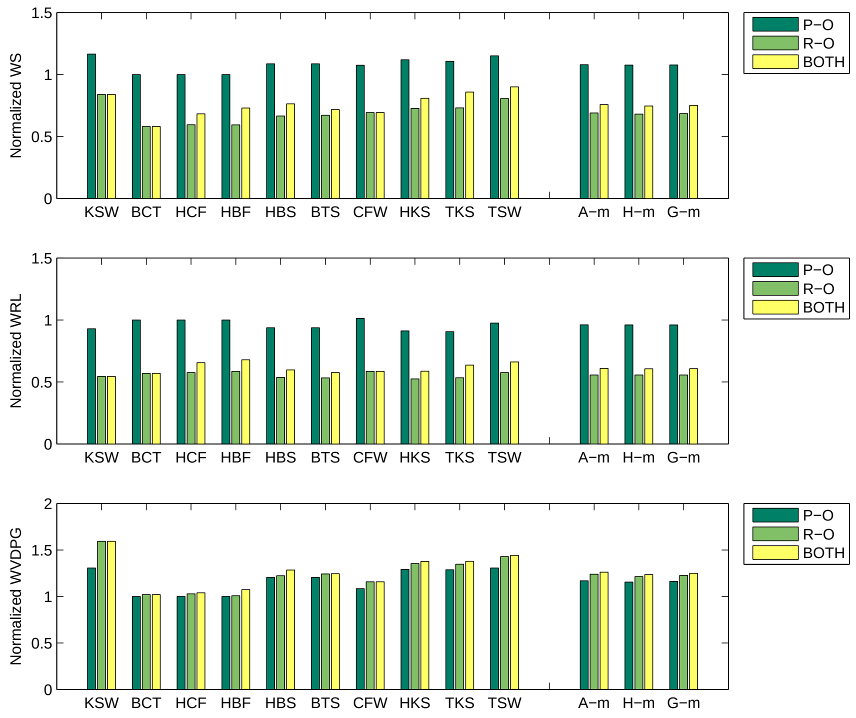


Figure 5.12. NWS, NWRL and NWVDPG values of partitioning schemes to equal core partitioning for selected 10 workloads with 3, 4, 5 core initial assignment.

5.3.4.2. The Number of Applications in the Workload. The number of applications per workload is important for core partitioning problem. To evaluate the effects of the number of applications on our results, we also conduct experiments with two-application and four-application workloads. We use our selected 10 workloads and again assume that 3 cores are assigned for each application initially. To keep equal core partitioning, we allocate 20 cores (both 2 and 4 applications are evenly partitioned to 20 cores) instead of 21. Therefore, equal core partitioning scheme allocates 10 cores and 5 cores for each application respectively. We first exclude one application from each benchmark and evaluate our partitioning strategies. Figure 5.13 demonstrates the performance and reliability values of new workload set. Since our initial assignment requires smaller number of cores to be allocated, *reliability-oriented* scheme and the scheme *based on combined metric* yield better results for reliability loss but both reduces performance gain. On the other hand, the effectiveness of the *performance-*

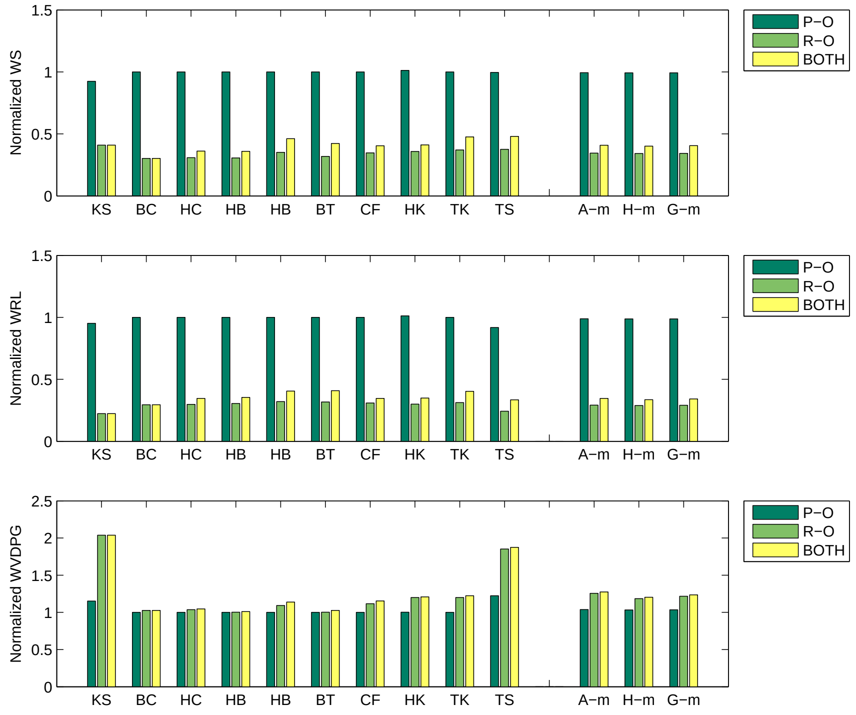


Figure 5.13. NWS, NWRL and NWVDPG values of partitioning schemes to equal core partitioning for 2-application workloads.

oriented scheme disappears when using the two-application workloads. The scheme approaches to equal core partitioning results since we select non-scalable application

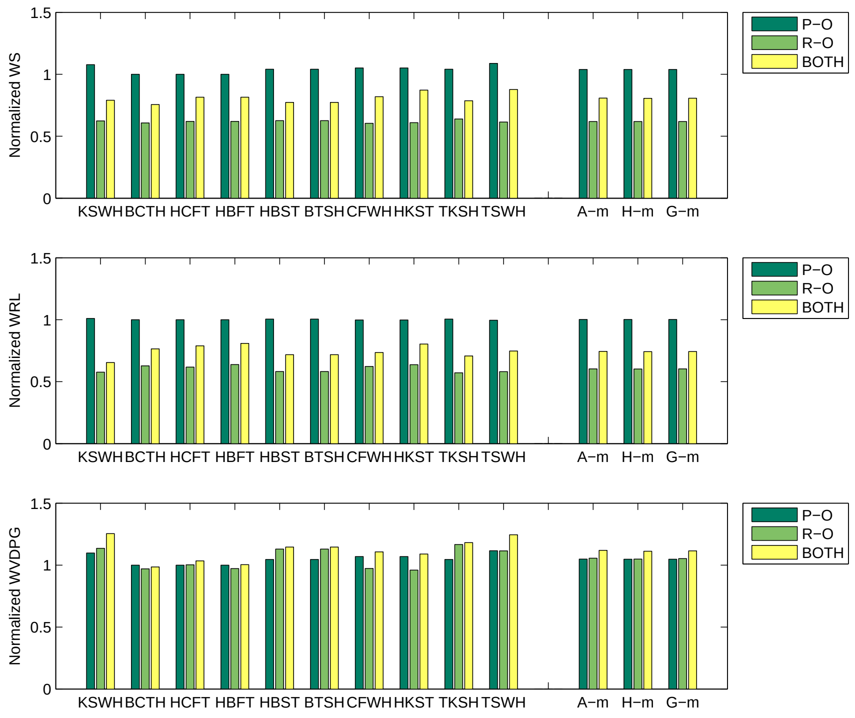


Figure 5.14. NWS, NWRL and NWVDPG values of partitioning schemes to equal core partitioning for 4-application workloads.

to remove from the workloads. Three-application workloads (HBS, BTS, CFW) consisting of two scalable and one non-scalable benchmarks become two-application workloads (HB, BT, CF) that contain only scalable applications and the partitioning results of these two-application workloads are the same as equal partitioning case which yields no weighted-speedup. For instance, our three-application workload *BTS* has 8, 9 and 4 cores under the *performance-oriented* scheme; however, two-application workload *BT* that is constructed by removing non-scalable *swaptions* has 10 core allocation for each application under the same scheme. This results show that the characteristics of the applications is as important as the number of applications in the workload sets. If we excluded scalable *streamcluster* benchmark instead of *swaptions*, our *performance-oriented* scheme would allocate 16 cores to *blackscholes* which is scalable but only 4 cores to non-scalable *swaptions*. Thus we could obtain both performance (11%) and reliability (20%) gain against equal core partitioning. We also conduct experiments for four-application workloads which are constructed by including one more application to our 10 workloads used in the previous experiments. Figure 5.14 demonstrates the weighted metric values for four-application sets. Since we initially allocate more cores for one extra application, our reliability loss metric is affected negatively but the performance gain becomes larger for *reliability-oriented* scheme and the scheme *based on combined metric*. *Performance-oriented* scheme again behaves similar to equal core partitioning scheme since the applications (*barnes* and *streamcluster*) included are scalable for each case. 15%-20% performance gains obtained by the scheme are lost for four-application workloads.

5.3.4.3. The Number of Cores in the System. Since the number of cores is a key parameter in a CMP system, we analyze the effect of the number of cores by conducting experiments for the systems with different core counts. Our simulation analysis includes 24-core and 18-core multicore architectures. We use the same 10 workloads consisting of 3 applications with the same initial assignment. Figure 5.15 and Figure 5.16 demonstrate the performance and reliability values of the 18-core system and 24-core system respectively. The results for *performance-oriented* scheme are not significantly different since additional cores provide the same amount of performance gain and reliability loss if compared to the base equal partitioning scheme. On the other hand, *reliability-oriented* scheme and the scheme *based on combined metric* affect the performance and reliability metric values as the number of cores alters in the system. The schemes, which consider the vulnerability, aim to allocate minimum number of cores in the system to maximize the system reliability. Essentially, the final assignment for these schemes are the same for the 18-core, 21-core and 24-core cases. However, the metric values differ due to the different assignments in the base equal partitioning scheme. While the proposed schemes are compared to the case which assigns 6 cores for each application in the 18-core system, the analysis for the 21-core and 24-core systems considers the 7 cores and 8 cores assignment for each application. Therefore, the weighted speedup tends to decrease as the number of cores increases in the system. Moreover, the weighted reliability loss metric has the highest values for the 24-core system which compares the metric with the equal partitioning scheme that assigns 8 cores to each application. NWVDPG metric that combines both the performance and vulnerability has the same trend for the *reliability-oriented* scheme and the scheme

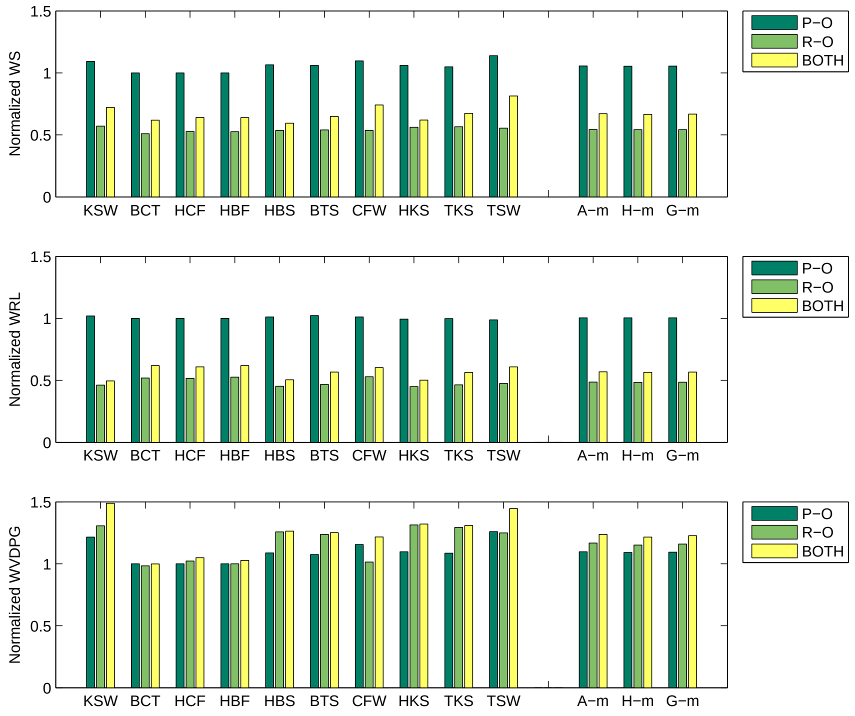


Figure 5.15. NWS, NWRL and NWVDPG values of partitioning schemes to equal core partitioning for 18-core system.

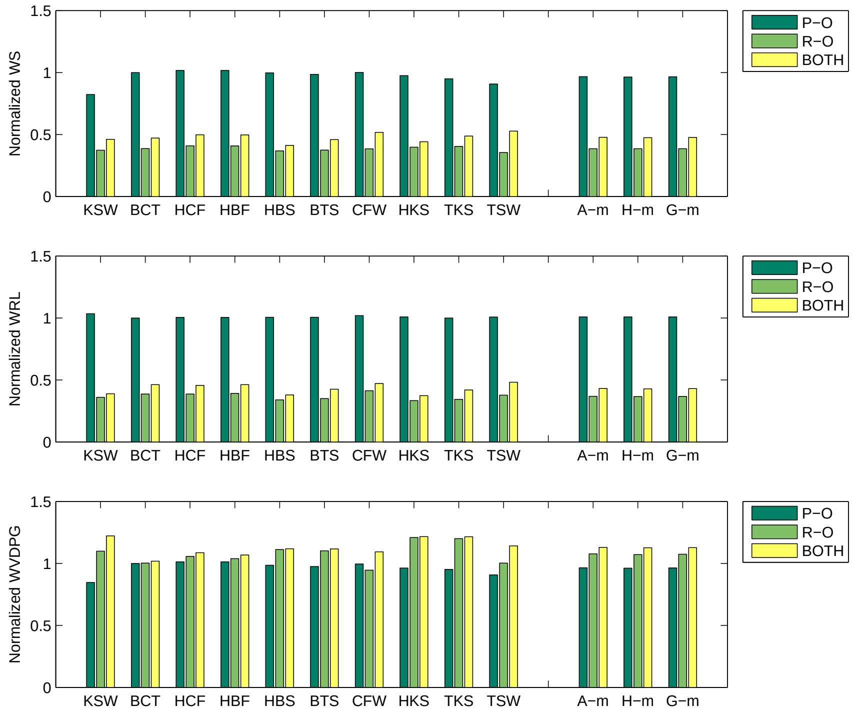


Figure 5.16. NWS, NWRL and NWVDPG values of partitioning schemes to equal core partitioning for 24-core system.

based on combined metric, it increases as the number of cores decreases in the system.

6. PARTIAL FAULT TOLERANCE BASED ON THREAD-LEVEL VULNERABILITY ASSESSMENT

To deal with soft errors and raise the fault tolerance level in the system, the replication of system resources has been used at both hardware and software level. Since the redundancy causes performance degradation and resource consumption, it is required to explore partial redundancy techniques which replicate the most vulnerable parts of the code. The redundancy level of user applications depends on user preferences and may be different for the users with different requirements.

In this chapter, we propose a user-assisted reliability assessment tool based on critical thread analysis for redundancy in parallel architectures [84]. We present introduction to fault tolerance techniques and related work in the literature in Section 6.1. Section 6.2 emphasizes the motivation behind our partial fault tolerance mechanism based on critical thread analysis. Section 6.3 represents our user-assisted reliability assessment mechanism by providing critical thread and critical region analysis, and the effect of redundancy on the system vulnerability is examined in Section 6.4. The benchmark applications used in our evaluations is given in Section 6.5, and results from our experimental analysis are presented in Section 6.6.

6.1. An Overview on Fault Tolerance Techniques

Redundancy, as a fault tolerance technique, is the replication of hardware and/or software components of a system by targeting to increase reliability [16, 21, 22, 85]. While the replication of hardware components is a method for reliability [15, 16], there have been software redundancy methods which execute the program code redundantly and compare the results at the end of the execution [20, 21, 86, 87].

In SRT (Simultaneous and Redundantly Threaded) method, two threads (two copies of the application) redundantly execute the same instructions in an SMT core [16].

Master (leading) thread executes the instructions but it updates LVQ (load value queue), which is a special memory unit, instead of memory. Slave (trailing) thread executes the same instructions, then reads value from LVQ and checks the results. Although SRT provides high fault-coverage rate, it results in high performance degradation since the leading and trailing threads compete for the resources in the system.

The fault tolerance mechanisms have been proposed for multithreaded applications as well as serial programs [88–92]. The program code is replicated as in the single-threaded case, but the atomic operations are synchronized between master and slave threads to provide correct execution.

Since the redundancy causes performance degradation and resource consumption, the replication of whole program may not be efficient and practical. Therefore, partial redundancy techniques based on the selective replication of instructions in a program have been proposed for higher performance and acceptable reliability [33, 93–97].

Partial explicit redundancy (PER) distinguishes execution phases as Single Execution Mode (SEM) and Redundant Execution Mode (REM) [93]. While only the main thread executes in SEM, the redundant thread executes with the main thread by considering IPC characteristics in REM phase. It provides both high soft error-coverage and low performance degradation due to partial redundancy scheme. Slick also proposes a partial redundant threading mechanism based on SRT processor [98]. It uses a set of predictors to estimate the output of the master thread without re-execution. Since the instructions, whose output has been predicted, do not need to be executed redundantly, the performance degradation of full redundancy scheme is reduced. Soundararajan *et al.* [94] also proposes a selective redundancy mechanism which selects a set of instructions for redundancy to provide maximum performance and minimum vulnerability based on a greedy heuristic dealing with the constraints. Silva *et al.* [52] proposes a partial redundancy scheme for stream processing applications. Their scheme is based on application quality analysis by considering an application-specific output score function.

Instruction-level redundancy schemes duplicate the instructions in a program at the compile time and compare the results of the replicated instructions at run-time [20, 21]. To reduce the cost of the full redundancy, there have been partial redundancy techniques which analyzes how the instructions affect the final application output [26, 33, 99]. Instruction-level fault tolerance configurability (ILCOFT) technique provides different protection levels for different instructions in an application by specifying the critical instructions which affect the output at most. ILCOFT-enabled system uses Instruction Vulnerability Factor (IVF) metric to determine the protection level of each instruction. Kumar *et al.* [100] also proposes an instruction-level partial redundancy method to reduce both performance loss and energy consumption. They define self-checking instructions, which do not need replication for fault tolerance, and reduce instruction redundancy level by implementing self-checking in redundant multi-threading scheme.

There have been redundancy techniques for fault tolerance of shared-memory multithreaded applications as well as single-threaded applications [88–90]. Sanchez *et al.* [89] proposes a new design for simultaneous and redundantly threaded method to reduce performance degradation of atomic operations. Atomic operations in parallel applications are handled by synchronizing master and slave threads in the redundant execution. An efficient redundancy scheme to deal with communication latency and nondeterministic ordering of communication events is proposed by mining available redundancy in the program execution [90].

Chen *et al.* [101] proposes a reliability-oriented approach for the computation of embedded array-intensive applications executing on chip multiprocessor systems. The approach uses idle processors to improve reliability by executing on them duplicates of the computation performed by the active processors. They consider performance, energy, and reliability issues for the redundant system. [102] also proposes partial code and data duplication schemes for embedded chip multiprocessors by considering memory overhead of the redundant execution.

6.2. Motivation

In CMP architectures, each additional core used for replication increases the system reliability for most of the cases. On the other hand, each additional core induces power consumption and affects energy efficiency badly. Moreover, potential performance improvement of additional processor core may be obstructed by consuming the core for the replication. To provide performance, reliability and energy efficiency, it is essential to use a few cores for the optimum reliability requirement. If the system tolerates some vulnerability, partial replication, which uses some of the cores, may be a reasonable choice.

Furthermore, the fault tolerance of applications running on multicore systems may be performed based on user preferences. Different user requirements may determine how much vulnerability is tolerable for the parallel execution. While some users may prefer to execute their applications in a system with a larger number of processors to provide higher reliability, others may tolerate vulnerability by considering higher hardware expenses. It is also possible for the user to define specific replication limits to be tolerated in terms of both cost and performance. Therefore, the redundancy identification process requires feedback from the user of an application and it is practical to apply partial redundancy by taking into account user preferences.

By considering vulnerability tolerance of the system and user-specific requirements, it is essential to analyze the target application and the system to determine the critical parts to be replicated. In a parallel program, it is reasonable to apply partial redundancy by considering the replication of the critical thread(s) affecting the system vulnerability mostly and to use idle cores for redundant threads' execution in an efficient way (in terms of performance, power and reliability). Additionally, thread-level redundancy may be extended by exploring critical regions of individual threads and executing redundantly only those regions to reduce redundancy overhead.

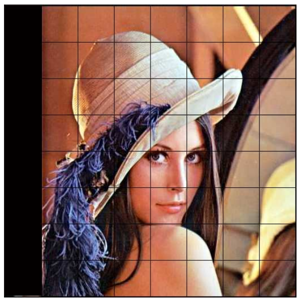
Soft errors may cause data corruption during program execution as well as program execution termination. Output corrupting faults have different severity, and it depends on the corruption magnitude and corruption location for an error to be important [103].

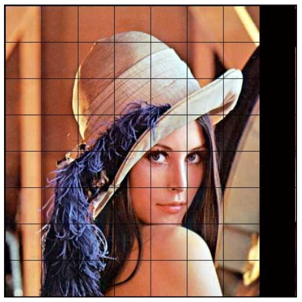
In a parallel application, there are multiple threads running concurrently each responsible for individual task. While some multithreaded applications exhibit data-parallel characteristics in which data is partitioned among threads, the others have multiple distinct tasks assigned for each thread. If the final output is composed of the partial results obtained by different threads, a possible error in one thread causes the final output corruption. However, partial data corruption may not have the same effect with the entire corruption for some applications.

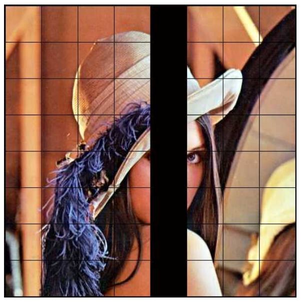
Identifying the critical thread in an application is more clear for some domains including image processing and computer graphics. Due to large amount of data to be processed on an image processing application, parallel processing reduces computation time by partitioning data among multiple threads [104]. In general, each thread works on pixel values by considering local region of the entire image in an image processing application. Since the output is formed with the partial operations provided by distinct threads, the correct execution of each thread is important for the correct final result. However, data corruption can be tolerable if the error occurs in different parts of the resulting image [105].

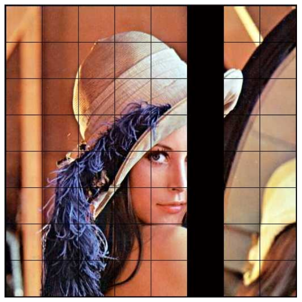
The characteristics of input data may affect the fault tolerance identification and the impact of input image is significant for image processing applications. Assume that, the image given in Figure 6.1 is partitioned column-wise among 8 threads for a filtering process of an image processing application. If an error occurs on threads that process the first or the last column, fault in the resulting image is tolerable since the error corrupts only the background of the image. However, the failure of the other threads (e.g., Thread 5 or Thread 6) is more critical due to loss of the important part of the image. Therefore, we should replicate more critical threads to provide higher fault tolerance.

In general, a parallel computer vision framework consists of several tasks in a


(a) Thread 1 corruption


(b) Thread 8 corruption


(c) Thread 5 corruption


(d) Thread 6 corruption

(e) Original

Figure 6.1. Sample data distribution.

pipelined structure. First of all, data acquisition and preprocessing phase is performed. Then compute-intensive tasks with multiple data processing are parallelized among threads. An error occurrence in the first phase is carried out by one or more threads; which may seriously affect the overall execution since the other threads use output data of this step due to the pipelined structure. Therefore, threads that are responsible for data preprocessing become the most critical threads for redundancy.

As can be seen in computer vision applications, all threads in a parallel application may not have the same vulnerability and reliability impact on the execution; therefore, it will be more practical to replicate only the most critical threads to provide redundancy in the system. Since users have different requirements for an application execution, and the redundancy level is identified by the users, a partial fault tolerance mechanism based on critical thread analysis should be managed based on user preferences. To the best of our knowledge, there is no user-assisted thread-level vulnerability assessment tool in the literature.

6.3. Thread-Level Reliability Assessment

This section presents our user-assisted reliability assessment tool design details and partial replication analysis algorithms.

6.3.1. System Design and Overview

Our reliability assessment tool consists of two components as illustrated in Figure 6.2:

- *User:* The user provides input to the analysis phase by specifying the number of cores to be used for redundant execution of a given parallel application. The number of cores indicates the maximum number of cores that is required by the user for reliability improvement. Then, the user executes the target application with redundant execution of threads and/or regions by considering direction advised by the analysis phase.
- *Analysis:* Our critical thread/region analysis consists of a simulation environment, based on Simics toolset [3]. The decision unit is a front-end component, which gets reliability preferences of users (in the form of number of cores for replications), and it initiates the execution of the target application on the simulator. Our critical thread/region analyzer (which is a Simics module) works in parallel with the application, and collects information about the execution threads at runtime. It calculates vulnerability value and criticality degree value (explained in Section 6.3.2) of each thread; and it performs region level analysis by considering the synchronization points to determine partial thread replications. At the end of the execution, analyzer decides the most critical threads and/or thread regions. It reports the criticality results to the decision unit of our tool. Finally, our decision unit combines user preferences and criticality results to make a suggestion on redundant execution of the application.

As an example, assume that a user has three available cores for replication of an application. Our decision unit may advise the replication of three most critical

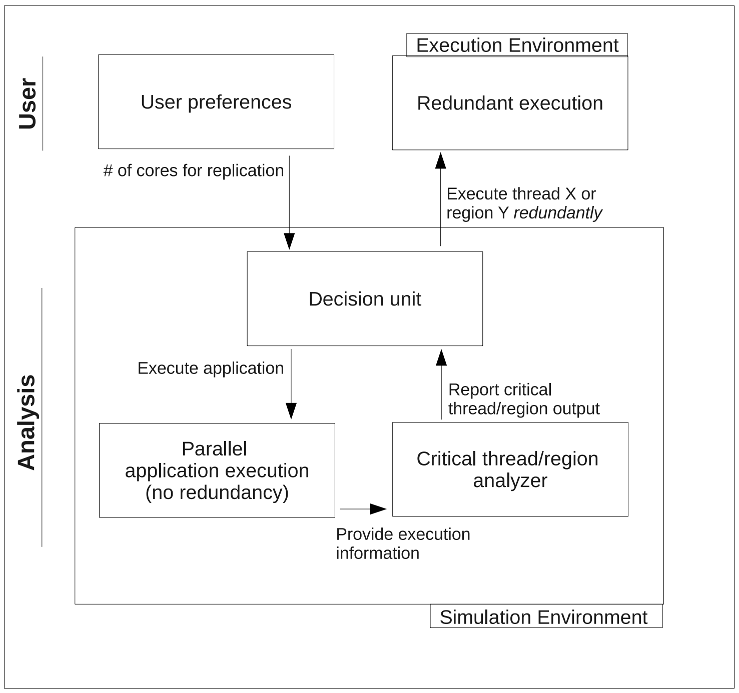

Execution Environment
User
User preferences
Redundant execution
# of cores for replication
Execute thread X or region Y *redundantly*
Decision unit
Execute application
Report critical thread/region output
Analysis
Parallel application execution (no redundancy)
Critical thread/region analyzer
Provide execution information
Simulation Environment

Figure 6.2. Flow of our reliability assessment tool.

threads if they are highly critical based on the analysis of our critical thread analyzer in our assessment tool; or it may suggest that two cores should be used for replication but third one may not be essential, which can be used for performance improvement or powered off for energy efficiency. The decision unit may also advise region level replication if it is more appropriate than thread replication.

6.3.2. Critical Thread Replication

To decide the most critical thread for redundancy, we consider both thread vulnerability and interactions between threads.

6.3.2.1. Thread Vulnerability. Although the thread having the largest TVF value is the most vulnerable one to transient errors, the redundancy of this thread may not be efficient for the vulnerability of the system.

The redundant execution of the thread that has the largest LVF value increases the system reliability by decreasing the vulnerability of the thread. Since the thread is more vulnerable to soft errors locally, it becomes more appropriate for redundant execution than the other alternatives, since by having higher probability of a soft error hits. However, RVF value of one thread does not provide relevant information about the redundancy of the thread since it has only the vulnerability of the other threads in the application. On the other hand, considering only the local term for the redundancy is not efficient for a multithreaded application due to the communication behavior of the threads.

6.3.2.2. Thread Interactions. In general, a thread that affects the other threads via remote memory write operations is critical for the system vulnerability since an error in this thread probably causes a failure on the other dependent threads, which read the erroneous data from the faulty thread. Therefore, it seems to be efficient for redundancy analysis to discover thread that has the *most remote writes*, i.e., the thread with the highest out-degree value. Additionally, *depth of a thread* and *number of writes* to the same destination thread are the other concerns that are utilized for retrieving thread interactions of our analyzer.

In a multithreaded application, in which multiple threads can communicate to each other along the execution, the vulnerability of one thread induced by the other threads should also be considered. Since the threads communicate via shared-memory, an error affecting one thread may cause failure in another thread, which reads the erroneous data written previously. Errors can propagate even if the memory system or the last level cache is highly reliable. Once the data written is faulty, ECC or any method will not be able to do anything.

Figure 6.3 represents a *thread interaction graph* (TIG) for the threads in a multithreaded application, which illustrates the communication of the threads in a timeline. In this application, T_2 frequently writes data which is read by the other threads. If an error hits T_2 and it calculates an erroneous data, the other threads may yield incorrect results using the wrong value. Since T_2 has the largest out-degree, its failure badly affects the system vulnerability by causing the reliability loss for all threads in the application.

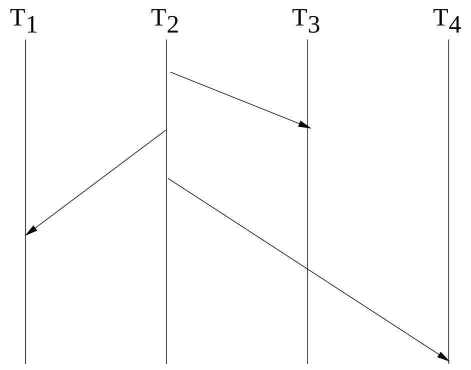


Figure 6.3. Thread interaction graph with four threads.

There are several thread communication patterns in parallel applications. Figure 6.4 represents a thread behavior of an 8-thread execution, where T_1 writes data read by other two threads (T_2 and T_4). While T_2 and T_4 reads data directly from T_1, all other threads (T_3 via T_2; T_5, T_6, T_7 and T_8 via T_4) in the execution may be affected by T_1 indirectly. Therefore, a failure in T_1 is critical for all the other threads in the application. If T_1 had more depth (which is 2 for this case), the importance of this thread would be much larger. On the other hand, T_4 has many outgoing edges which indicate that many threads may be affected directly if an error hits T_4. The effect of out-degree and the depth may depend on the thread interaction graph of the given application.

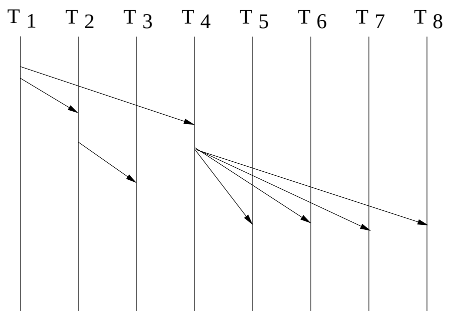


Figure 6.4. Thread behavior of an 8-thread application.

During the execution of a parallel application, one thread may write data and it is read by a specific thread multiple times. Each remote write operation may affect the remote thread directly, since an error causing incorrect calculation also corrupts the remote thread. Although we should consider each write operation separately, it is also possible to take into account only the last write.

Figure 6.5 represents an example communication scenario in which T_2 writes data read by T_1 for three times (X, Y and Z values). An error hit at the code segment marked with "a" possibly corrupts X value. The corruption of Y value is possible in case of an error which hits the code segment "b". Z value may be corrupted due to an error on the code segment "c", similarly. Moreover, Y and Z values are dependent on the code segment of T_1 which includes "a" and "a+b" respectively. Since we consider the code segment "a+b+c" for the last write operation (Z value), it is not necessary to count the previous remote write operations (X and Y values) for the critical thread analysis. Although these multiple write operations increase the dependency of T_1 on T_2, they do not enhance the importance of T_2 for redundancy analysis. The same Z value is also read by T_3, and T_2 becomes more critical since it affects two different threads. However, the remote write operations on T_1 and T_3 have the same effect on the critical thread analysis in our tool.

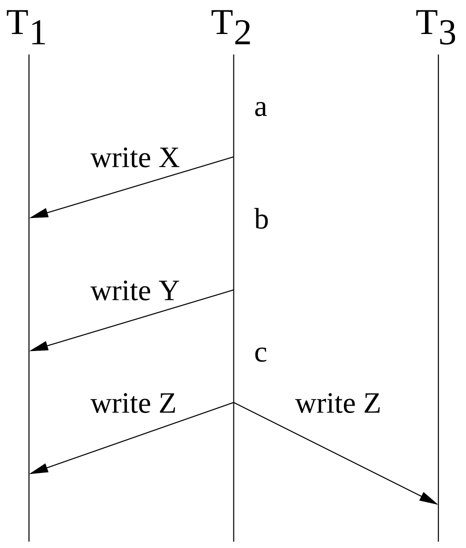


Figure 6.5. An example thread interaction graph which has multiple remote write operations of a single thread.

6.3.2.3. Critical Thread Identification Algorithm. Our thread-level vulnerability assessment tool considers both local behavior of each individual thread and thread interactions to determine the critical thread for redundancy in a parallel application.

The analyzer (given in Figure 6.2) tracks memory load/store operations during program execution. It logs store operations for each thread and evaluates load operations by considering the interactions between threads. We calculate LVF value of each thread during execution and we also keep track of thread interactions.

The *criticality degree* of each node in the thread interaction graph is calculated by using two components, which are *direct criticality degree* and *indirect criticality degree.* The former one represents the criticality induced from the write/read relationship between two threads. If one thread alters a memory location and another thread reads that data, the writer thread directly affects the reader thread and becomes more critical due to its effect on the reader thread. This interaction leads to an increase on direct criticality degree value. Since the vulnerability is represented by LVF, we increase value of direct criticality degree of the writer thread by LVF amount of the thread. On the other hand, the *indirect criticality degree* represents the criticality propagated from previous write/read relationships of the writer thread. All threads having affected the writer thread has an indirect effect on the reader thread. This causes a slight increase in indirect criticality degree value of the threads due to the weighted sum of direct and indirect criticality degree values.

We track memory operations in our target parallel application running on a multicore architecture and store them on a *hashmap*, where each entry of the hashmap contains a memory location and the number of the thread that writes to the given location. Whenever a store operation occurs, a new entry, for representing thread that performs the store operation, is added to the hashmap. If the map already contains this memory location, the entry is updated by the current thread number.

Algorithm 6.6 presents criticality degree calculation procedure for load operations. The direct and indirect criticality degree values are stored in *direct_degree* and *indirect_degree* matrices, respectively. If the memory location in the load operation has been altered by another thread previously, the writer thread criticality degree value is incremented due to its effect on the reader thread. In Algorithm 6.6, the $[tid_{out}][tid_{in}]$ entry of *direct_degree* matrix, which stores total vulnerability effect of the writer threads on the reader threads, is updated. Additionally, the *indirect_degree* matrix is modified by analyzing all threads. For both direct and indirect criticality computations, there are three distinct structures of each resource including ALU, *register*, and *memory*. To reduce the effect of the indirect threads, the vulnerability values are multiplied by a *weight* term which is a predefined value in the range [0..1].

```
procedure on_store_operation(tid, location, hashmap)
  add (hashmap, location, tid)
procedure on_load_operation(tid, location, hashmap)
  $tid_{in} \leftarrow tid$
  $tid_{out} \leftarrow$ search (hashmap, location)
  if $tid_{in} <> tid_{out}$ then
    $direct\_degree[tid_{out}][tid_{in}] += LVF(T_{tid_{out}})$
    //Update direct criticality values
    for each thread $tid_X$ do
      $indirect\_degree[tid_X][tid_{in}] +=$
      $weight \times (direct\_degree[tid_X][tid_{out}] + indirect\_degree[tid_X][tid_{out}])$
      //Update indirect criticality values
    end for
  end if
```

Figure 6.6. Algorithm for calculating direct and indirect criticality degree values of threads.

After running the multithreaded application and collecting statistics on the behavior of threads, the most critical thread or threads for redundancy is determined based on the Algorithm 6.7. The algorithm considers either criticality degree matrices of vulnerability factors based on a predefined threshold value, ϵ. If maximum number of remote memory write operations is larger than ϵ, then degree metrics are utilized to determine the critical thread. If the remote write operations are not large enough, local vulnerability values are used since local behavior of threads determines the critical thread in the application. We assign the threshold value by comparing values from different applications. The majority of three resources, which are ALU, memory, and registers determines the most critical thread of the given application. As an example, if T_1 is selected by both ALU and memory, and T_2 is selected by register unit as the critical thread, then our algorithm returns the T_1 as the most critical thread.

```
procedure $evaluate(n, threshold)$
  $MCT \leftarrow$ the most critical thread
  $RC \leftarrow \max(remote\ count[tid_n])$
  if $RC > threshold$ then
    $CT\_ALU \leftarrow \max_n(degree_{ALU}[tid_n])$
    $CT\_mem \leftarrow \max_n(degree_{memory}[tid_n])$
    $CT\_reg \leftarrow \max_n(degree_{register}[tid_n])$
  else
    $CT\_ALU \leftarrow \max_n(LVF_{ALU}[tid_n])$
    $CT\_mem \leftarrow \max_n(LVF_{memory}[tid_n])$
    $CT\_reg \leftarrow \max_n(LVF_{register}[tid_n])$
  end if
  $MCT \leftarrow majority(CT\_ALU, CT\_mem, CT\_reg)$
```

Figure 6.7. Algorithm for determining critical thread of an application.

6.3.2.4. An Example Execution. Figure 6.8 represents an example thread interaction graph for an 8-thread execution. For this execution scenario, T_2 reads a value which calculated by T_1 previously. Then T_2 writes a value which will be read by T_3. T_4 produces data; and the threads other than T_1 and T_2 read these data values along their execution. T_1 is the thread with the largest depth which writes to T_2 directly, and T_3 indirectly. LVF_1, LVF_2 and LVF_4 values denote the local vulnerability factor values in the course of remote write operation for threads T_1, T_2, and T_4, respectively.

We may store remote write/read relationships in a matrix structure where the rows represent the *writer* threads and the columns represent the *reader* threads (see Table 6.1). When a thread, T_1, remotely writes a value which is read by another thread,

T_2, the entry in the first row and the second column is filled with the vulnerability value that T_1 has at the time of write operation; i.e., $LVF_1 = LVF(T_1)$. This entry indicates that T_1 affects the vulnerability of T_2 by the specified value. The other remote write operation (i.e., from T_2 to T_3) is more interesting since there is an indirect communication between T_1 and T_3. Firstly, the cell of the second row and the third column in Table 6.1 is filled with LVF_2 values for direct communication between T_2 and T_3. This operation also leads to a new entry at the cell of the first row and the third column, due to an indirect communication between T_1 and T_3. Since the effect is not direct and the possibility of that T_1 impacts T_3 is not large as the effect on T_2, we may reduce the vulnerability value by a constant rate (w). If the vulnerability is multiplied with a number between 0 and 1, we may reduce the effect of the indirect remote thread. The other remote write operations by T_4 to other threads also are added to the related locations of the matrix. At the end of the execution, each row of the table (Table 6.1 for thread interaction graph given in Figure 6.8) represents the criticality degree of the corresponding thread.

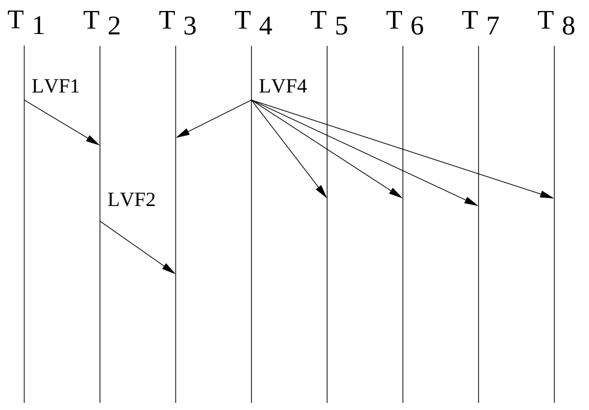


Figure 6.8. A thread interaction graph with 8 threads.

6.3.3. Critical Region Replication

Since the redundancy introduces synchronization overhead between replicated threads, it may be more preferable not to replicate the complete code of a thread if the reliability gain is not significant; one or more critical regions of the thread can be replicated, instead.

Table 6.1. Matrix for direct and indirect criticality degree values for critical thread analysis.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
T_1	-	$LVF1$	$w \times LVF1$	-	-	-	-	-
T_2	-	-	$LVF2$	-	-	-	-	-
T_3	-	-	-	-	-	-	-	-
T_4	-	-	$LVF4$	-	$LVF4$	$LVF4$	$LVF4$	$LVF4$
T_5	-	-	-	-	-	-	-	-
T_6	-	-	-	-	-	-	-	-
T_7	-	-	-	-	-	-	-	-
T_8	-	-	-	-	-	-	-	-

For critical region replication, we track communication points at thread codes and determine the code region that contributes most to the criticality of the thread. The TIG of an application given in Figure 6.9 represents remote write operations of threads in an execution, where a number above an edge represents an interaction between two threads. Our critical thread analyzer clearly concludes that the most critical thread for redundancy is T_2 due to its plenty of remote write operations; and it suggests the replication of T_2 for this figure. However, full replication of thread 2 may not be necessary. Since most of the remote write operations occur until the *interaction 5*, the replication of the thread code up to this point may be adequate for fault tolerance.

To determine the critical code region of a given thread, we perform criticality degree calculations at the end of each synchronization point during program execution. After gathering thread-level criticality degree values at distinct points, we compare the values of consecutive points. If the values are not significantly different, we may conclude that the replication of the last region is not crucial and suggest the partial replication of the thread by excluding the ineffective region. As an example, if the criticality degree values between the last two write operations (*interaction 5* and *interaction 6* in Figure 6.9) are not much different, we may say that the redundancy of the code region after *interaction 5* does not provide significant gain.

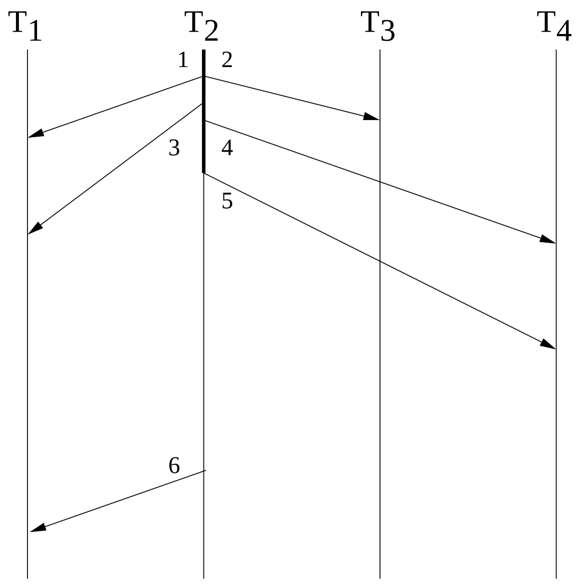


Figure 6.9. A TIG example to represent synchronization of thread regions.

6.4. Vulnerability of Redundant Computations

6.4.1. Vulnerability Evaluation

To illustrate the effect of the selective thread redundancy on the system reliability, we extend Thread Vulnerability Factor (TVF) which measures the vulnerability of multithreaded applications running on multicore architectures. Since TVF metric does not consider thread redundancy, we extend it to evaluate the thread vulnerability for redundant execution. First of all, we should evaluate how TVF is affected by redundant computations and how to calculate TVF if there are redundant threads. Since the local term of the thread vulnerability (i.e., LVF term) represents the vulnerability of the thread induced by the code itself, its value decreases if this is a redundant copy of the thread. While the redundancy increases the reliability of the redundant thread, it increases the reliability of the other threads as well. The remote term of the thread vulnerability (RVF) represents the vulnerability impact of the threads that interact with the target thread and is calculated by considering the vulnerability of remote threads. Specifically, RVF for T_2 in Figure 6.8 can be calculated as follows:

$$RVF(T_2) \quad = \quad TVF(T_1)$$

$$= [\,w_L \times LVF(T_1)\,] + [\,w_R \times RVF(T_1)\,]$$

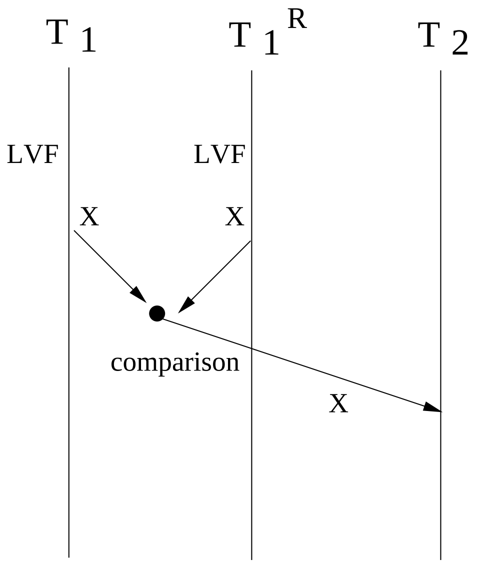


Figure 6.10. TIG for thread replication case.

Assume that T_1^R is the replica of T_1 in Figure 6.10. Then the vulnerability (LVF) of the remote thread T_1 decreases; therefore, LVF term should be reduced in the above calculation. If we multiply the local term by itself, its value decreases and the effect of the redundancy appears in the vulnerability of the dependent thread T_2. RVF for T_2 in the redundant execution of T_1 becomes:

$$\begin{aligned} RVF(T_2) &= TVF(T_1) \\ &= [\,w_L \times (LVF(T_1) \times LVF(T_1))\,] + [\,w_R \times RVF(T_1)\,] \end{aligned}$$

Since LVF term takes value in range [0..1], the local vulnerability of T_1 (i.e., the partial vulnerability of the remote vulnerability of T_2) is reduced. These reductions on the vulnerability of threads that read data from the redundant thread result in the reduction on the overall system vulnerability which is calculated by augmenting TVF of each thread in the application.

For the partial thread replication case, we calculate remote values with the same consideration. We reduce remote values only for the replicated code regions, while the calculation is the same for non-redundant parts.

6.4.2. A Case Study

To evaluate critical thread identification and the vulnerability computation of a redundant system, we execute a synthetic application with the thread interaction graph given in Figure 6.8. In our application, the remote write operations consist of simple array element calculations. The amount of data for each communication is fixed, that is, the number of elements written and read for each interaction is equal. Since the synchronization part in the application is not related to our analysis, we exclude the barrier code which provides atomic operations in the parallel application. We also do not consider remote write operations that are less than a predetermined threshold value, due to the simulation inconsistencies. We gather vulnerability data for three architectural resources including register, memory, and ALU. The reduction factor for indirect remote writes is taken as 0.8 in our experiments.

The result of the critical thread analysis for this application is given in Table 6.2, which includes only threads' criticality degrees. We also provide the number of remote write operations (given in remote count column) to point out the out-degree value, without concerning vulnerability concept. Based on Table 6.2, the most critical thread

Table 6.2. Metric values for critical thread analysis of synthetic application.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
ALU	689.104	399.537	0.000	1839.236	0.000	0.000	0.000	0.000
register	775.859	430.016	0.000	2235.174	0.000	0.000	0.000	0.000
memory	1033.348	474.736	0.000	2457.000	0.000	0.000	0.000	0.000
remote count	1800	1001	0	5001	0	0	0	0

for redundancy is T_4 which has the largest out-degree values for all resources (register, memory and ALU). T_4 has also the largest number of remote write operations as seen in the last row.

After we determine the most critical thread for redundancy in the application, we gather TVF values for the redundant case. Table 6.3 represents the vulnerability values of each thread in the synthetic application. We evaluate the redundant cases for T_4 which is the largest criticality degree values for our critical thread analysis and T_1 which is the largest depth (i.e., its depth is equal to 2) in the dependency structure. Although our analysis recommends us to replicate T_4, we also evaluate vulnerability values for T_1 redundancy in order to compare the results of different cases. Table 6.3 demonstrates that T_3, which has the most remote read operations (due to direct read from T_2 and T_4, and indirect read from T_1), has the largest remote vulnerability values for all resources. Since T_1 and T_4 have no dependent threads, their remote vulnerability factor values equal to zero. The vulnerability values for the other threads are similar to each other due to the same number of remote read operations.

The effects of the redundant threads on the vulnerability values are given in Table 6.3. We calculate the vulnerability values as explained previously by reducing the vulnerability effect of the redundant thread. If one thread is replicated, the remote vulnerability values of the threads that read data written by this thread decrease since the this thread's code reliability is improved by means of redundant execution. While the vulnerability of all threads except T_2 is decreased for the redundancy of T_4, only T_1 and T_2 have smaller vulnerability values for the redundancy of T_1. Thus total system vulnerability, which is equal to the sum of thread vulnerabilities, decreases by larger amount for T_4 redundant case.

6.5. Benchmark Applications

In our experimental analysis, we perform critical thread evaluation for a variety of parallel programming patterns and thread behavior. We select four main classes of patterns including *task parallel*, *divide and conquer*, *geometric decomposition*, and *pipeline* programming model [106]. We evaluate our analysis on parallel benchmark applications from PARSEC [38] and SPLASH-2 [62] suites by choosing applications that exhibit different patterns.

Table 6.3. RVF values of redundant executions of synthetic application.

		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	$Total$
No redundancy	ALU	0.000	0.191	0.239	0.000	0.184	0.184	0.184	0.184	1.166
	Register	0.000	0.215	0.272	0.000	0.223	0.224	0.224	0.224	1.382
	Memory	0.000	0.287	0.313	0.000	0.245	0.246	0.246	0.247	1.584
Redundant=4	ALU	0.000	0.191	0.182	0.000	0.068	0.068	0.068	0.068	0.645
	Register	0.000	0.216	0.211	0.000	0.100	0.100	0.101	0.101	0.829
	Memory	0.000	0.291	0.252	0.000	0.121	0.122	0.122	0.123	1.031
Redundant=1	ALU	0.000	0.074	0.210	0.000	0.184	0.184	0.184	0.184	1.020
	Register	0.000	0.094	0.242	0.000	0.223	0.224	0.224	0.224	1.231
	Memory	0.000	0.172	0.285	0.000	0.246	0.246	0.247	0.247	1.443

- *Task parallel:* The thread tasks are balanced and there is no data dependencies between tasks (*blackscholes*, *swaptions* from PARSEC).
- *Divide and conquer:* The task is divided into subtasks, the solutions to the subtasks are then combined to give a solution to the original task (*radix* from SPLASH-2).
- *Geometric decomposition:* The problem is decomposed into smaller chunks operated in parallel, the solutions is composed of updates to local chunks and boundaries of chunks which induces data sharing between neighboring threads (*barnes*, *fft*, *lu* from SPLASH-2).
- *Pipeline:* There is data flow between coarse grained tasks and it is executed on pipeline stages (*canneal* from PARSEC).

6.6. Experimental Results

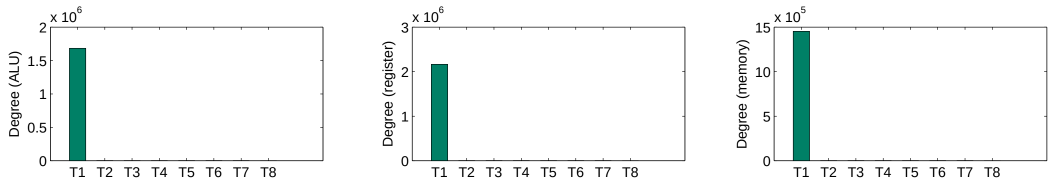
We execute our benchmark applications on an 8-core multicore architecture by mapping one thread onto one core in order to evaluate our vulnerability analysis. Our experimental analysis consists of two phases including *critical thread evaluation* and *critical region evaluation*. We consider the replication of the execution codes (both thread and region) in our redundancy experiments, with the assumption of improving the system reliability. In this work, we do not deal with redundancy levels (duplicate, triplicate) that may have distinct effects on the reliability.

6.6.1. Evaluating Critical Thread Replication

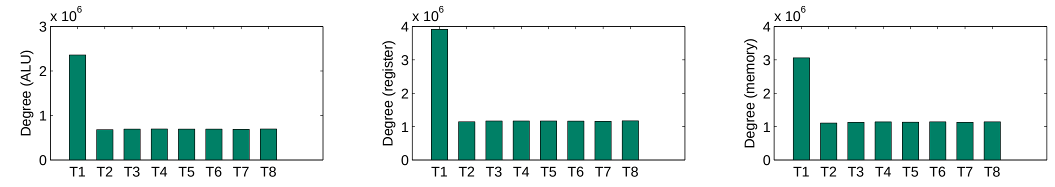
For evaluating our critical thread assessment tool, it requires critical thread analysis, replication and validation phases, which are summarized at the following subsections.

<u>6.6.1.1. Critical Thread Analysis.</u> An application is executed by mapping one thread onto one core and memory operations are tracked to construct dependency structures for critical thread analysis. At the end of the execution, we figure out the most critical thread(s) for replication, and collect statistics to calculate the vulnerability (TVF) of the execution. Figure 6.11 presents values of criticality degree terms of each thread for ALU, register, and memory resources, for 8 applications selected from benchmark suites. When the results are examined, the criticality degree values are evidently larger for one thread (T_1) in *blackscholes*, *canneal* and *swaptions*. Since *blackscholes* and *swaptions* have task-parallel characteristics, their threads have similar tasks with no communication along their execution. Therefore the threads other than the first thread have similar criticality degree values which are not too large due to the lack of communication. Only the first thread, which distributes the input data to other threads, has large criticality degree values for each resource. *Canneal* application, which exhibits pipeline pattern, has large communication between its threads. However, the threads have similar criticality degree values due to the homogeneous work. Again the first thread having the input data has the largest criticality degree value for all the resources. For *blackscholes*, *canneal* and *swaptions*, the first thread should be selected for replication if we want to increase the reliability with minimum number of replications. Our analyzer suggests exactly one thread replication to the user, and it advises not to use any other processor core for reliability improvement.

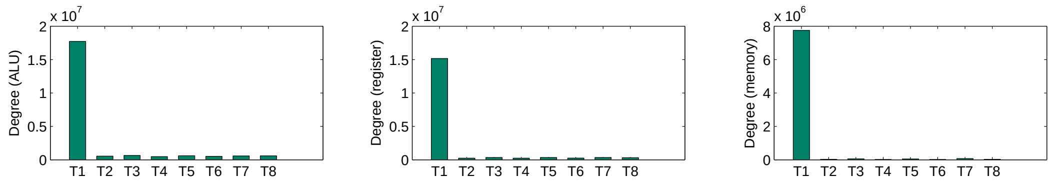
Since *FFT* threads exchange data along their executions, the criticality degree values are similar to each other. It is probable that the replication of any thread results



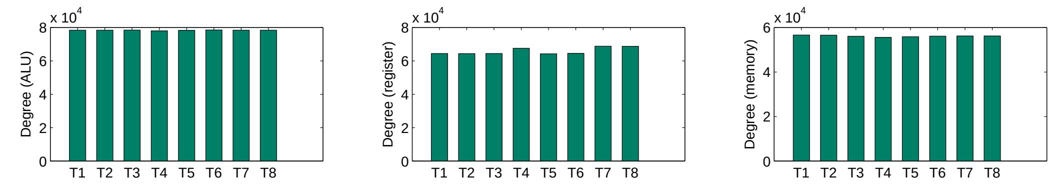
(a) blackscholes



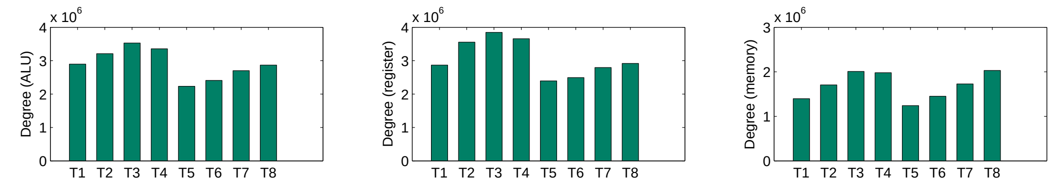
(b) canneal



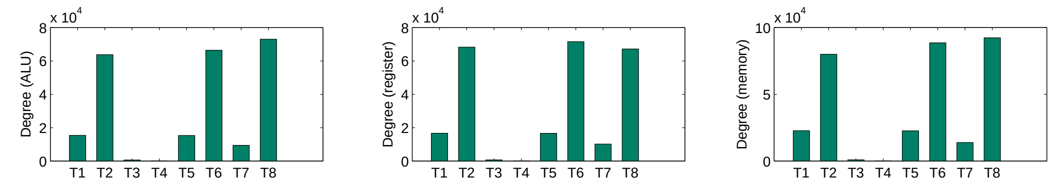
(c) swaptions



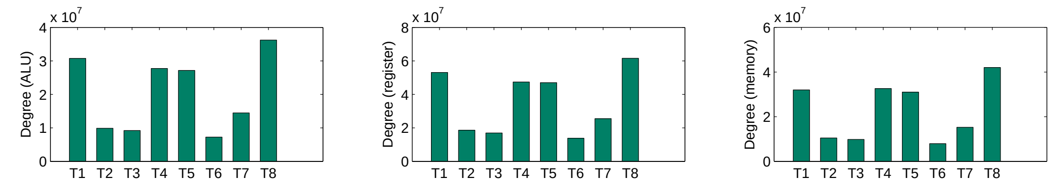
(d) FFT



(e) LU



(f) radix



(g) barnes

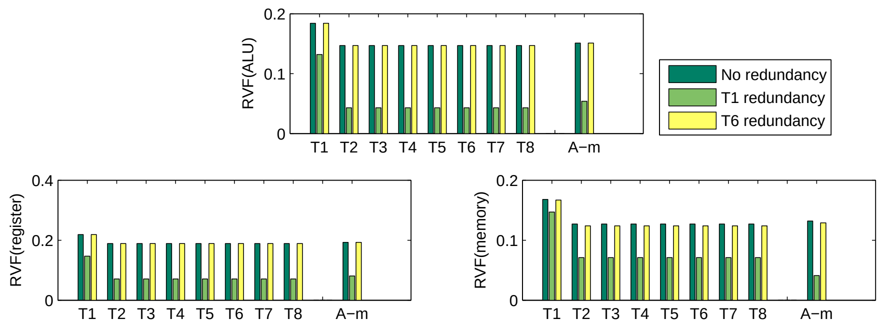
Figure 6.11. Metric values for benchmark applications.

in the same amount of reliability gain. We cannot select the thread for redundancy with our critical thread analysis. We may suggest to the user to use as much as possible cores for replication to improve system reliability.

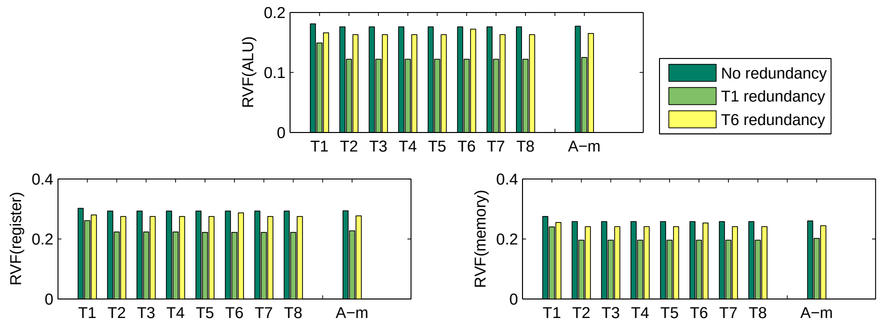
On the other hand, the difference between the criticality degree values of application threads for *LU*, *radix* and *barnes* is more apparent. Since the application threads have diverse characteristics, the critical thread analysis becomes more essential for these applications. While *LU* threads have more similar criticality degree values, *radix* and *barnes* threads exhibit diverse values. If we have resources for only one thread replication, we select T_3 for *LU* and T_8 for *radix* as well as *barnes* in order to decrease the vulnerability in a most efficient way.

An inference on the least critical thread can be stated by using our critical thread analysis. Since each replication causes additional resource and performance cost, it is also critical to decide the thread which may not be replicated. *Radix* application figures show that T_3 and T_4 have almost no effect on the other threads. If we do not execute these threads redundantly, the reliability does not change significantly due to the lack of remote effects of these threads.

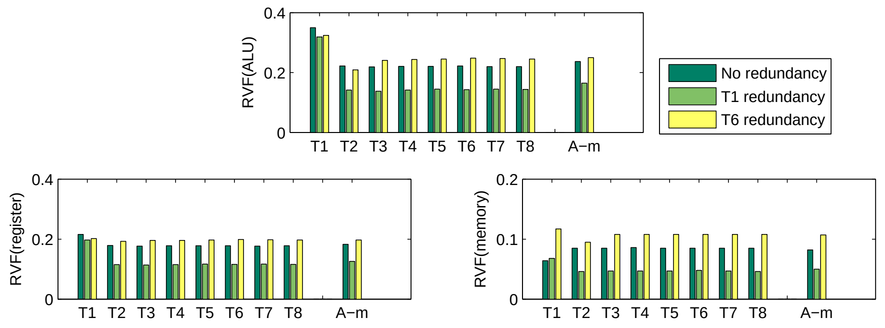
6.6.1.2. Critical Thread Replication. After discovering the most critical thread(s) for redundancy, we re-execute our applications by a redundant copy of the most critical thread (if any) and calculate the vulnerability values by considering the effect of the redundant copies. In this work, we do not deal with synchronization of redundant copies and do not consider performance issues. We assume that redundant threads decrease the vulnerability and partial redundancy based on critical thread analysis reduces the vulnerability with smaller performance degradation. To validate and demonstrate the efficiency of our analysis, we also include experiments of no redundancy and the replication of other threads in the application. We compare the vulnerability values for different replication cases. Our executions include at most one thread replication for each case. Figure 6.12 and Figure 6.13 present the vulnerability values (RVF for ALU, register and memory resources) for redundant cases of PARSEC and SPLASH-2 appli-



(a) blackscholes



(b) canneal



(c) swaptions

Figure 6.12. The vulnerability values for redundant cases of PARSEC applications.

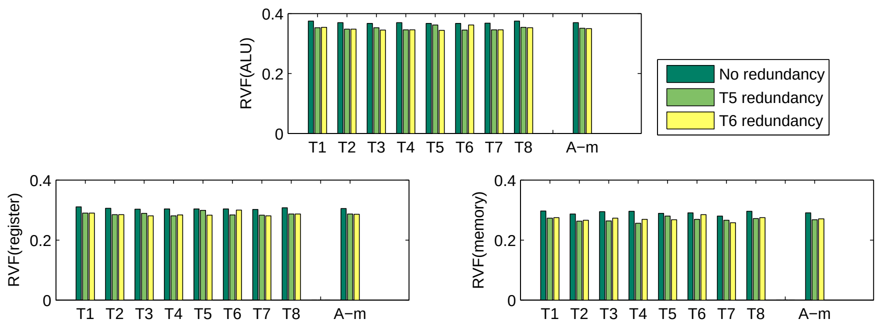
cations respectively. We include the normal execution case with no redundancy and two additional scenarios with one thread replication for each application. The figures demonstrate the remote vulnerability factor values for each thread as well as the arithmetic mean (shown with A-m bars).

Since T_1 is the most critical thread for *blackscholes*, *canneal* and *swaptions* applications, we include the first thread replication which demonstrates the vulnerability decrease in case of the most critical thread replication. We also execute a randomly selected thread (T_6) redundantly as a second scenario; and calculate the vulnerability values for ALU, register and memory resources. Figure 6.12 demonstrates that while T_1 replication increases the reliability significantly, there is almost no effect of T_6 replication in the vulnerability values.

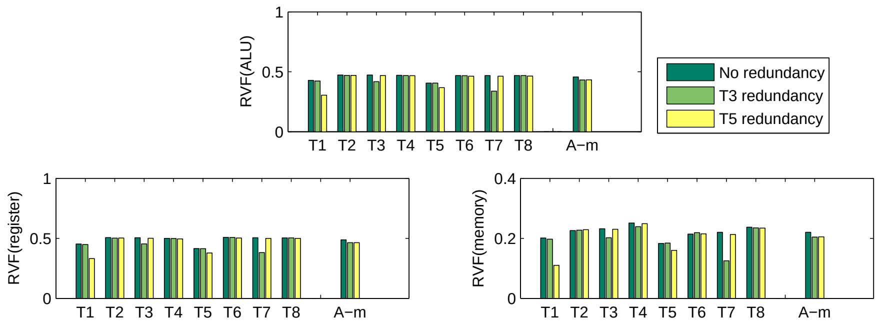
Since none of FFT threads exhibit distinct characteristics for critical thread analysis, we select two threads (T_5 and T_6) randomly for redundancy in order to evaluate the effect of any thread's replication on the vulnerability. Figure 6.13(a) demonstrates that the vulnerability values decrease for redundant cases, where the difference is not significant in average.

While LU has more diverse communication behavior between its threads, the average reliability improvement for T_3 (the most critical thread) replication is not much different compared to T_5 (the least critical thread) replication. T_3 is the most critical thread, but the difference between the criticality degree values are not much significant. Therefore, the effect of the most critical thread replication is not clear for LU application. While the distinct vulnerability values for individual threads differ, the mean values are similar.

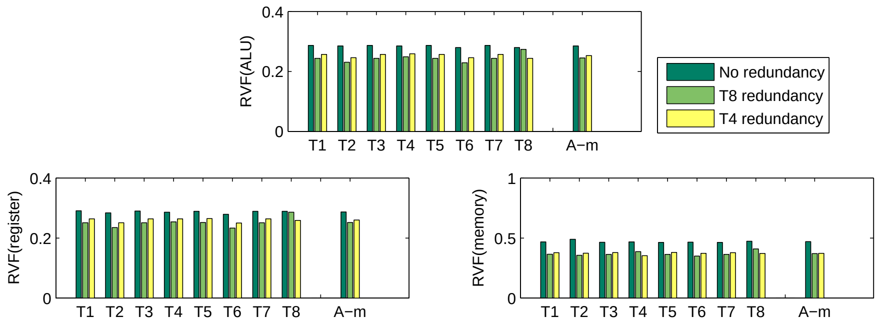
The most interesting results that demonstrate the effect of critical thread analysis belong to *radix* and *barnes* applications. T_8 is the most critical thread for both applications. The replication of the most critical thread decreases the vulnerability values for each resource significantly. We include the replication of the least critical threads in order to emphasize the difference between the partial replication cases. Although



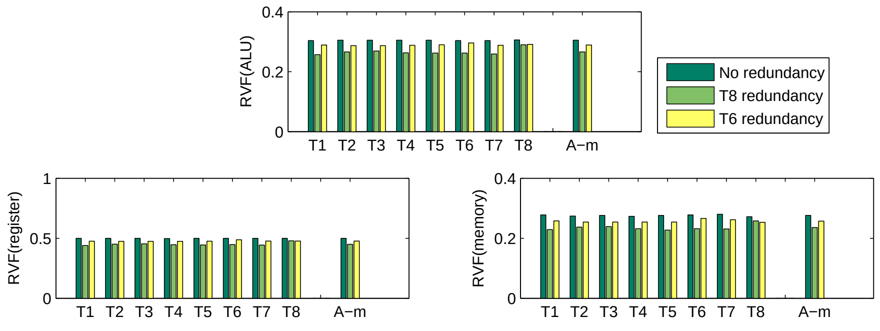
(a) FFT



(b) LU



(c) radix



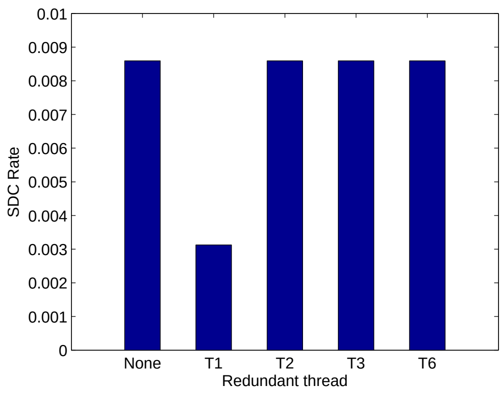
(d) barnes

Figure 6.13. The vulnerability values for redundant cases of SPLASH-2 applications.

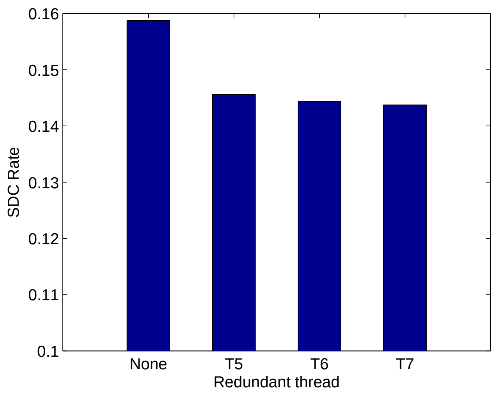
the replication of thread T_4 for *radix* and thread T_6 for *barnes* causes vulnerability decrease for some threads, the overall reliability improvement is smaller than the case of replicating the most critical thread. These results demonstrate that if we replicate the thread other than the most critical one, the decreases at vulnerability values would not be so high. Therefore, the thread with the highest criticality degree value should be selected, if there are limited resources for full redundancy.

6.6.1.3. Validation of Critical Thread Replication. We execute selected benchmark applications on our fault-injection framework (given in Chapter 3) to validate our critical thread-based fault tolerance scheme. SDC (Silent Data Corruption) errors are subtle form of errors considered in our study, which include both self-thread errors and fault propagation errors in a parallel program execution. SDC rate is utilized as a metric to compare results, which is the fraction of the injected faults that results in unacceptable outputs [50]. We do not classify the data corruptions as acceptable or unacceptable, we assume all data corruptions are unacceptable. To analyze the output errors and detect data corruption, the application output should be deterministic and easy to compare. Therefore, we select a subset of applications that have exact results from the Parsec and Splash-2 benchmarks for our fault injection experiments including *blackscholes*, *LU*, *FFT*, and *radix*. Figure 6.14 represents the SDC rates of applications for different redundant cases. We include no redundancy case as well as the replication of the most critical and the least critical threads evaluated at Section 6.6.1.2. While *blackscholes* has significantly lower SDC rates for the most critical thread replication (T_1), *LU* and *FFT* redundant cases have relatively similar results. It is also observed that the replication of T_8 and T_6 which have the largest criticality degree values (see Figure 6.11) for *radix* application reduces the SDC rates more than the replication of T_3 and T_4 which have the smallest criticality degree values.

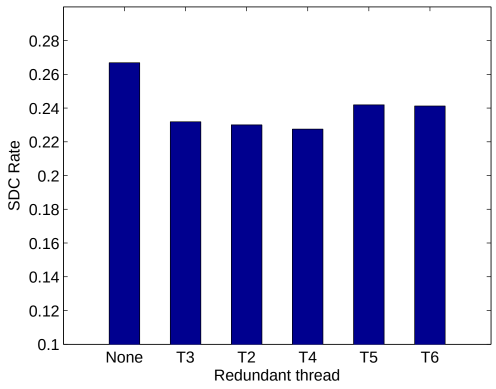
In another experiment, SDC rates are computed by varying the number of redundant threads (See Figure 6.15). We start with no redundant case and include one thread replication by considering the criticality degree value. For example, we assume the replication of T_8, the replication of both T_8 and T_6, the replication of T_8, T_6, and



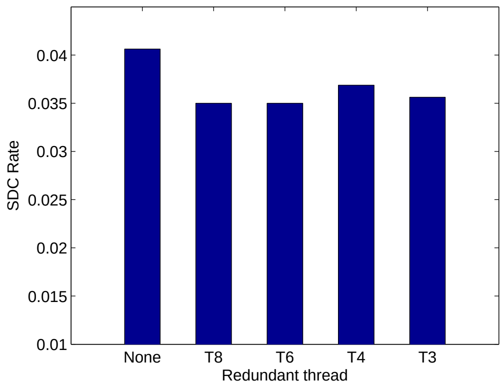
(a) blackscholes



(b) fft

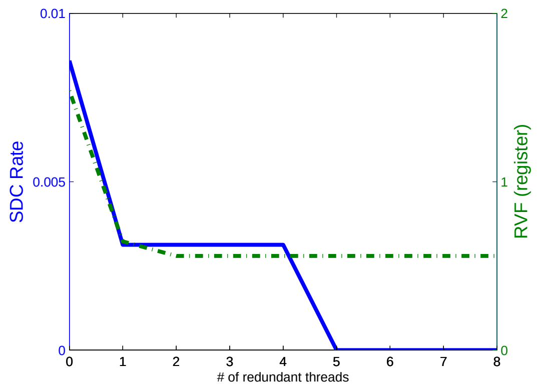


(c) lu

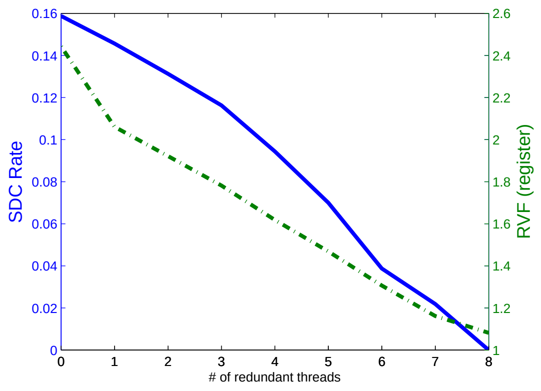


(d) radix

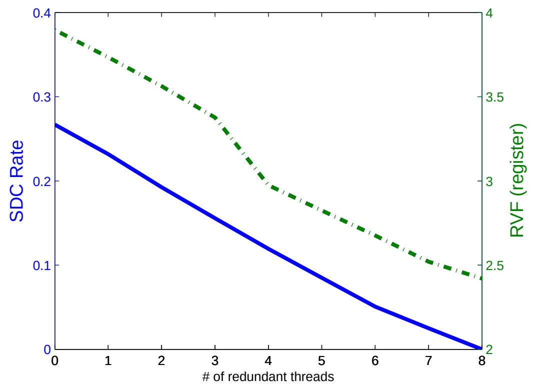
Figure 6.14. SDC rates for redundant cases.



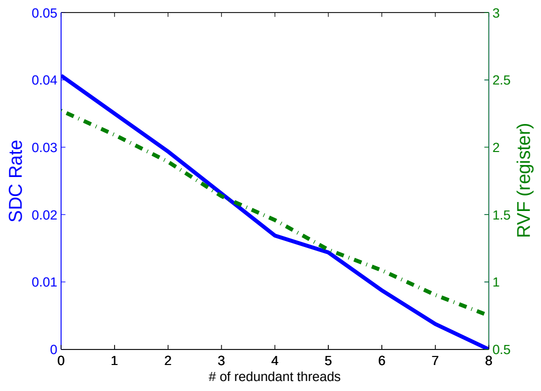
(a) blackscholes



(b) fft



(c) lu



(d) radix

Figure 6.15. Vulnerability values (SDC rate and RVF value) vs number of redundant threads.

T_2 for *radix* application for the case of one, two, and, three thread replications, respectively. We construct the complete graph by including one more thread replication, and end with full redundancy which results in zero SDC rate. We also conduct experiments for the replication of different number of threads and calculate the vulnerability values by considering the redundancy effect on vulnerability values. The similar results for our RVF values are represented in Figure 6.15. These results validate our partial replication scheme based on critical thread evaluation.

6.6.2. Evaluating Critical Region Replication

We extend our critical thread analysis by considering synchronization points of the applications. We consider execution steps as thread codes divided by these points in our critical region evaluation. The applications that has the behavior of geometric decomposition pattern are more appropriate for critical region analysis due to much communication between threads and many synchronization points. Therefore, we select LU and FFT for evaluating critical region replications. The first application for our critical region analysis is LU, which has 19 synchronization points in SPLASH-2 implementation. Since the first three intervals do not have any criticality degree values, we include only the last 16 regions for our analysis. We calculate criticality degree values at the end of each 16 execution steps by considering register, ALU, and memory resources. Since register resource has the largest values, we use the results of the that resource in our analysis. We construct graphs for distinct threads in Figure 6.16 to illustrate the criticality degree values at the end of each execution step represented by the synchronization points in the code. As presented in our previous critical thread analysis, the most critical threads for redundancy in LU application were T_2, T_3, and T_4. While the criticality degree values of these threads do not differ significantly, we may select T_3 for critical thread replication if we have only one available core. However, critical region analysis allows us to replicate different regions from different threads. The most critical regions of the most critical threads (T_2, T_3, and T_4) are execution intervals between 10-11, 5-6, and 6-8 (among 16 intervals) respectively. If the user has only one core to use for redundancy, we may advise the selective redundancy for dif-

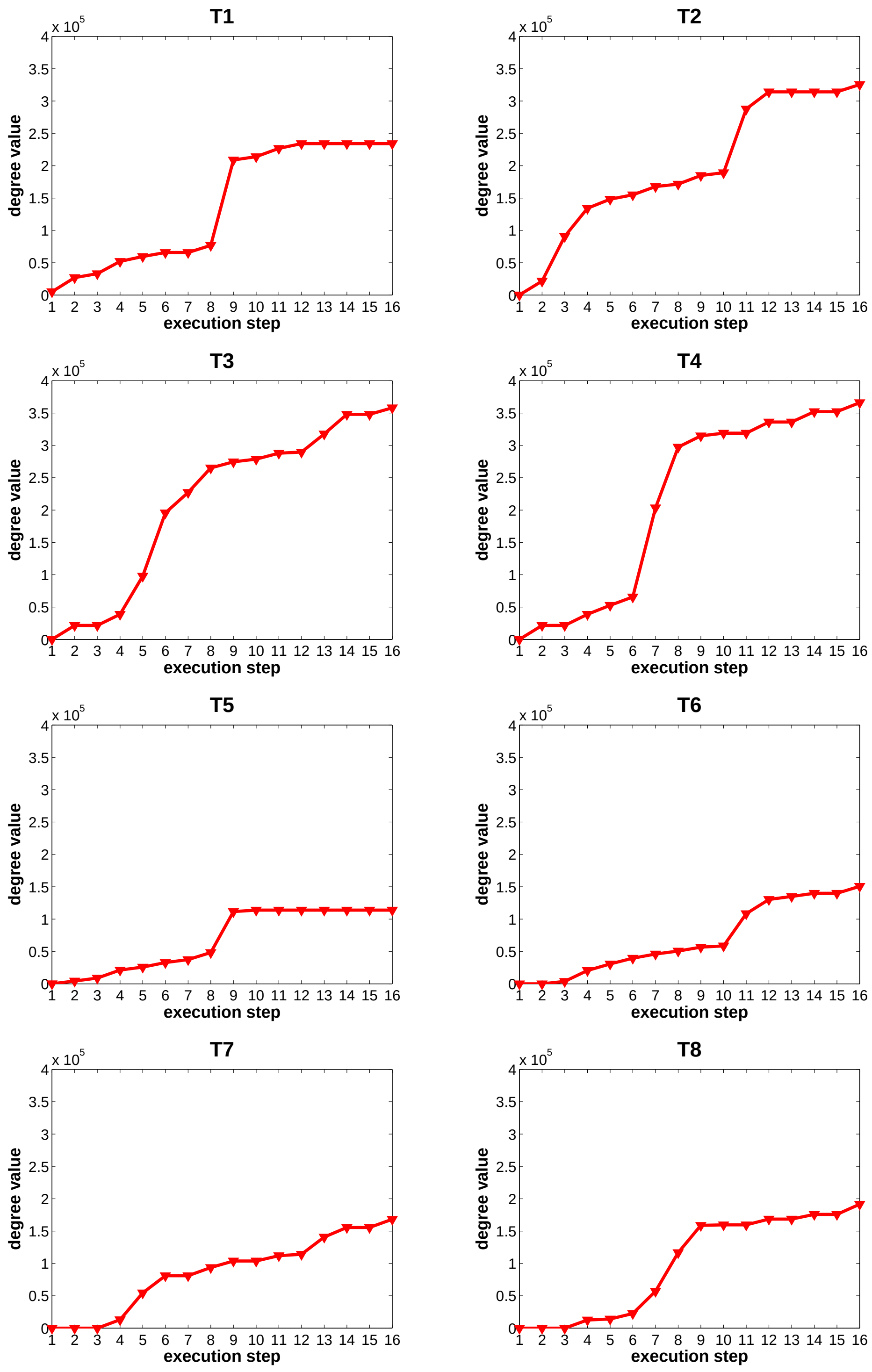


Figure 6.16. Metric values of LU execution steps for distinct threads.

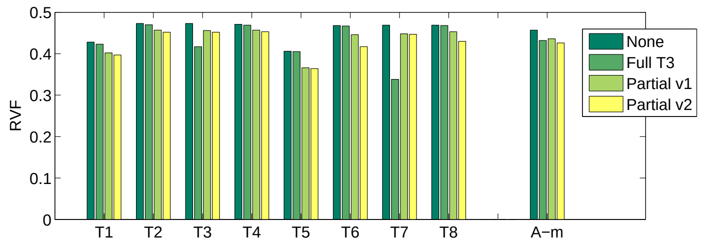


Figure 6.17. Vulnerability values for partially redundant cases of lu application.

ferent time intervals. While T_2 is executed redundantly between 10-11 execution steps, T_4 is replicated at 6-8 execution steps. The interval between 5-6 execution step is more critical for T_3, and the redundancy level for other steps may be determined by similar observations. If the synchronization overhead for redundant threads is not tolerable for performance, no redundancy may be applied for some intervals due to slight increase in vulnerability.

To illustrate the effect of critical region replication on the vulnerability of LU, we execute the application for different redundancy cases and calculate vulnerability values based on our critical region replication technique. We gather values for both one critical thread replication and partial replication of selective threads for LU application. Figure 6.17 represents RVF values for four cases including no replication, full replication of thread T_2, and two partial thread replication cases (stated as *Partial v1* and *Partial v2*). For the case of *Partial v1* replication, we replicate the most effective three regions from three most critical threads obtained from our critical region analyzer, which are the replication of T_2 between 10-11 execution steps, the replication of T_3 between 5-6 execution steps, and the replication of T_4 between 6-8 execution steps. We do not execute any redundant code in any other regions. We include the redundancy of relatively less effective regions for *Partial v2* case. Since we assume that we have one available core for the redundant execution, all regions from different threads have been replicated to increase the reliability of the application. We select the replicated thread by considering its contribution at the region, and compare the values for the eight threads. For LU application, we determine following replications: the replication of T_2 between 1-5 and 10-11 execution steps, the replication of T_3 between 5-6 execution

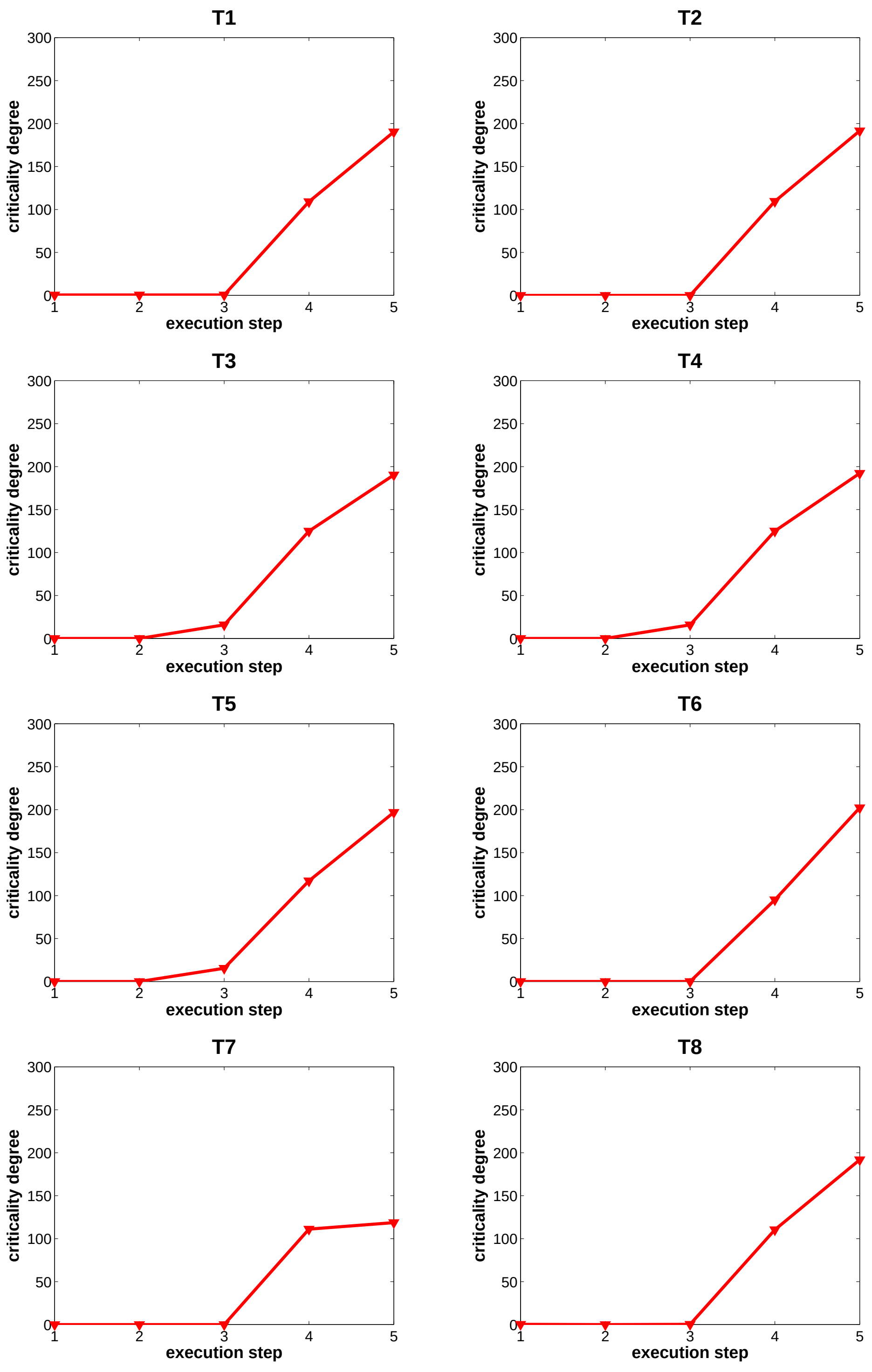


Figure 6.18. Metric values of FFT execution steps for distinct threads.

steps, and the replication of T_4 between 6-10 and 11-16 execution steps. As illustrated in Figure 6.17, *Partial v2* case, which considers both the critical region redundancy from the critical threads and the utilization of the available core for the redundancy, has the smallest vulnerability values. While *Partial v1* case does not present the best case for the vulnerability, it may be a choice to avoid from the performance loss that is induced by the synchronization overhead of the replicated threads.

Another benchmark application for our analysis is *FFT*, which we do not conclude any critical thread identification for full thread replication. The execution steps also have similar behavior for *FFT* threads (see Figure 6.18). While the first two intervals do not contribute the criticality significantly, the last two intervals (especially interval 3-4) are effective for all threads. We may advise to replicate only the code region between 3-4 if the synchronization overhead is very crucial.

7. CONCLUSIONS AND FUTURE WORK

The main focus of this thesis is to quantify reliability of multithreaded applications and to propose reliability-oriented solutions to map multithreaded applications on CMP architectures.

We propose a novel reliability metric, *Thread Vulnerability Factor (TVF)*, which represents the vulnerability of multithreaded applications to soft errors on multicore architectures. It takes into account remote vulnerability factor which is reflected from related threads as well as its local vulnerability factor. Since the TVF includes the vulnerability resulted from the dependency of the threads, it gives more effective and detailed information about the vulnerability of the multithreaded applications. Our results indicate that TVF values tend to increase with increasing number of cores, which means the system becomes more vulnerable as the core count rises.

We validate our proposed metric with both fault injection based experiments on Simics simulator and executions on a real multicore architecture. The fault injection based experiments demonstrate that a vulnerability comparison between multiple parallel applications based on TVF analysis yields similar results with a costly fault injection experimental study. We can use TVF metric to evaluate the relative vulnerability of multithreaded applications efficiently. Sample runs on a real multicore architecture provides consistent results with our simulation-based TVF evaluation.

Additionally, we present a performance-reliability analysis of different multithreaded applications running on multicore architectures. While the performance of the applications is measured by execution clock cycles, we use TVF metric to evaluate the relative reliability of multithreaded applications. Using these values gathered from experimental evaluation, we conduct a performance-reliability tradeoff analysis which compares different parallel versions of an application in terms of reliability as well as performance. We repeat this on three different applications. Our results indicate that the choice is clear for *FFT* and *Jacobi Kernel*. The transpose algorithm for *FFT* calculation results in less than 5% performance loss while the vulnerability increases by 20% compared to binary-exchange algorithm. One can prefer transpose algorithm for better reliability by sacrificing little performance. The unrolled *Jacobi* code reduces execution time up to 50% by not effecting vulnerability values. However, the tradeoff is more interesting for *Water Simulation* application. While *nsquared* version reduces the vulnerability values significantly, it increases execution time with similar rates compared to *spatial* version. One should trade performance with reliability to choose one version of the application.

As part of this thesis, we propose and evaluate reliability-aware core partitioning schemes of multicore architectures for multiple multithreaded applications. To consider both high performance and reliability objectives, a novel metric called Vulnerability-Delay product (VDP) is presented as part of this study. The VDP metric is a combined metric that includes both execution time for representing the performance and TVF for representing the vulnerability of the system to the soft errors. Experimental study performed on various workloads validates our core partitioning schemes.

A thread-level vulnerability assessment tool is presented by considering user preferences in this thesis. We propose a novel critical thread identification algorithm to determine critical thread and/or critical region of a thread in a multithreaded application. Our analysis evaluates the application threads of a parallel program by considering their criticality in the execution and selects the most critical thread or threads to be replicated. The most critical thread in a parallel program is the thread that affects the other threads via remote memory write operations, since an error in a thread probably causes a failure also on the dependent threads. Moreover, we extend the evaluation by exploring critical regions of individual threads and execute redundantly only those regions to reduce redundancy overhead. Our experimental evaluation indicates that the replication of the most critical thread improves the system reliability more than the replication of any other thread.

As a future work, a vulnerability aware thread scheduling strategy can be proposed in order to reduce the residence time of shared data between threads of a multithreaded application on a multicore system in order to increase the reliability of the computation by decreasing the possible data corruption on shared caches [107]. DAG (Direct Acyclic Graph) based structures, where nodes represent threads and edges represent thread communications, may be useful by adding vulnerability related weights that represent the reliability values. The scheduling algorithm may try to optimize the graph-based structures with the aim of minimizing application vulnerability to soft errors [108].

Another direction for future work is to consider energy and power constraints in addition to reliability for parallel computer architectures. Data reliability may be investigated in an energy-efficient fashion in the presence of soft errors. Vulnerability-energy tradeoff analyzes may be possible to optimize system reliability [109, 110]. As investigated reliability-aware systems in this thesis, energy-aware computing may be an extension for highly reliable computer architectures [111, 112].

REFERENCES

1. Weaver, C., J. Emer, S. S. Mukherjee and S. K. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor", *31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

2. Sridharan, V. and D. R. Kaeli, "Eliminating Microarchitectural Dependency from Architectural Vulnerability", *Proceedings of IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

3. S.Magnusson, M.Christensson, J.Eskilson, D.Forsgren, G.Hallberg, J.Högberg, F. Larrson, A. Moestedt and B. Werner, "Simics: A Full System Simulation Platform", *IEEE Computer*, Vol. 35, No. 2, pp. 50–58, 2002.

4. Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", *Proceedings of the conference on Programming language design and implementation*, 2005.

5. Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation*, Vol. 19, No. 90, pp. 297–301, 1965.

6. Olukotun, K., B. A. Nayfeh, L. Hammond, K. Wilson and K. Chang, "The case for a single-chip multiprocessor", *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 1996.

7. Hayes, J. P., I. Polian and B. Becker, "An Analysis Framework for Transient-Error Tolerance", *Proceedings of the 25th IEEE VLSI Test Symposium (VTS)*, 2007.

8. Shivakumar, P., M. Kistler, S. Keckler, D. Burger and L. Alvisi, "Modeling the

Effect of Technology Trends on the Soft Error Rate of Combinational Logic", *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2002.

9. Goloubeva, O., M. Rebaudengo, M. S. Reorda and M. Violante, *Software-Implemented Hardware Fault Tolerance*, Springer, 2006.

10. Pradhan, D. K., *Fault-Tolerant Computer System Design*, Prentice Hall, 1996.

11. Neumann, J. V., "Probabilistic logics and the synthesis of reliable organisms from unreliable components", *Automata Studies*, Vol. 34, pp. 43–99, 1956.

12. Brown, W., J. Tierney and R. Wasserman, "Improvement of Electronic-Computer Reliability through the Use of Redundancy", *IEEE Transactions on Electronic Computers*, Vol. 10, No. 3, pp. 407–416, 1961.

13. Vaidya, N. and D. Pradhan, "Fault-Tolerant Design Strategies for High Reliability and Safety", *IEEE Transactions on Computers*, Vol. 42, No. 10, pp. 1195–1206, 1993.

14. Koren, I. and S. Su, "Reliability analysis of N-modular redundancy systems with intermittent and permanent faults", *IEEE Transactions on Computers*, Vol. 28, No. 7, pp. 514–520, 1979.

15. Austin, T. M., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", *Proceedings of 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1999.

16. Reinhardt, S. K. and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading", *27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

17. Mukherjee, S. S., M. Kontz and S. K. Reinhardt, "Detailed Design and Eval-

uation of Redundant Multithreading Alternatives", *29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

18. Wells, P. M., K. Chakraborty and G. S. Sohi, "Mixed-Mode Multicore Reliability", *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

19. Ungsunan, P. D., C. Lin, X. Kong and Y. Gai, "Improving Multi-Core System Dependability with Asymmetrically Reliable Cores", *International Conference on Complex, Intelligent and Software Intensive Systems*, 2009.

20. Oh, N., P. P. Shirvani and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors", *IEEE Transactions on Reliability*, Vol. 51, No. 1, pp. 63–75, 2002.

21. Reis, G. A., J. Chang, N. Vachharajani, R. Rangan and D. I. August, "SWIFT: Software Implemented Fault Tolerance", *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2005.

22. Reis, G. A., J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems", *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2005.

23. Mukherjee, S. S., C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor", *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

24. Sridharan, V. and D. R. Kaeli, "Quantifying Software Vulnerability", *Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies (WREFT)*, 2008.

25. Yan, J. and W. Zhang, "Compiler-guided Register Reliability Improvement

Against Soft Errors", *Proceedings of 5th ACM international conference on Embedded software (EMSOFT)*, 2005.

26. Borodin, D., B. B. Juurlink and S. Vassiliadis, "Instruction-Level Fault Tolerance Configurability", *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, 2007.

27. Nair, A. A., L. K. John and L. Eeckhout, "AVF Stressmark: Towards an Automated Methodology for Bounding the Worst-Case Vulnerability to Soft Errors", *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

28. Walcott, K. R., G. Humphreys and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state", *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, 2007.

29. Li, X., S. V. Adve, P. Bose and J. A. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors", *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008.

30. Zhang, W., "Computing Cache Vulnerability to Transient Errors and Its Implication", *Proceedings of 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2005.

31. Zhang, W., "Computing and Minimizing Cache Vulnerability to Transient Errors", *IEEE Design and Test*, Vol. 26, No. 2, pp. 44–51, 2009.

32. Shrivastava, A., J. Lee and R. Jeyapaul, "Cache Vulnerability Equations for Protecting Data in Embedded Processor Caches from Soft Errors", *Proceedings of conference on Languages, compilers, and tools for embedded systems (LCTES)*, 2010.

33. Borodin, D. and B. B. Juurlink, "Protective Redundancy Overhead Reduction

Using Instruction Vulnerability Factor", *Computing Frontier (CF)*, 2010.

34. Oz, I., H. R. Topcuoglu, M. Kandemir and O. Tosun, "Quantifying Thread Vulnerability for Multicore Architectures", *Proceedings of 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, 2011.

35. Oz, I., H. R. Topcuoglu, M. Kandemir and O. Tosun, "Thread vulnerability in parallel applications", *Journal of Parallel and Distributed Computing*, Vol. 72, No. 10, pp. 1171–1185, 2012.

36. Culler, D., J. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware/-Software Approach*, Morgan Kaufmann, 1999.

37. *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, `http://www.intel.com/design/intarch/manuals/243191.htm`, accessed at March 2012.

38. C.Bienia, S.Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", *Proceedings of international conference on Parallel architectures and compilation techniques (PACT)*, 2008.

39. Woo, S. C., M. Ohara, E. Torrie, J. P. Singh and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995.

40. Bienia, C., S. Kumar and K. Li, "PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors", *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

41. Clark, J. A. and D. K. Pradhan, "Fault Injection", *Journal Computer*, Vol. 28, No. 6, pp. 47–56, 1995.

42. Arlat, J., M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications", *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, pp. 166–182, 1990.

43. Gunnetlo, O., J. Karlsson and J. Tonn, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", *International Symp. Fault-Tolerant Computing*, 1989.

44. Karlsson, J., J. Arlat and G. Leber, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", *IEEE International Working Conference Dependable Computing for Critical Applications*, 1995.

45. Hsueh, M.-C., T. K. Tsai and R. K. Iyer, "Fault Injection Techniques and Tools", *Journal Computer*, Vol. 30, No. 4, pp. 75–82, 1997.

46. Han, S., K. Shin and H. Rosenberg, "DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment", *Computer Performance and Dependability Symposium*, 1995.

47. Kanawati, G. A., N. A. Kanawati and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System", *IEEE Transactions on Computers*, Vol. 44, No. 2, pp. 248 – 260, 1995.

48. Stott, D. T., B. Floering, D. Burke, Z. Kalbarczyk and R. K. Iyer, "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors", *In Proceedings of the IEEE International Computer Performance and Dependability Symposium*, 2000.

49. Chandra, R., R. M. Lefever, K. R. Joshi, M. Cukier and W. H. Sanders, "A Global-State-Triggered Fault Injector for Distributed System Evaluation", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 7, pp. 593–605,

2004.

50. Hari, S. K. S., M.-L. Li, P. Ramachandran, B. Choi and S. V. Adve, "mSWAT: low-cost hardware fault detection and diagnosis for multicore systems", *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

51. Wang, N. J., A. Mahesri and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection", *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

52. Jacques-Silva, G., B. Gedik, H. Andrade, K.-L. Wu and R. K. Iyer, "Fault injection-based assessment of partial fault tolerance in stream processing applications", *Proceedings of the 5th ACM international conference on Distributed event-based system*, 2011.

53. Bach, M. M., M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil and A. Tal, "Analyzing Parallel Programs with Pin", *Journal Computer*, Vol. 43, No. 3, pp. 34–41, 2010.

54. *Fault Analysis Using Pin*, `http://www.cs.virginia.edu/kim/publicity/pin/tutorials/ISCA33/index.htm`, accessed at November 2012.

55. Oz, I., H. R. Topcuoglu, M. Kandemir and O. Tosun, "Performance-reliability tradeoff analysis for multithreaded applications", *Design, Automation, and Test in Europe (DATE)*, 2012.

56. Gupta, A. and V. Kumar, "The Scalability of FFT on Parallel Computers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 8, pp. 922–932, 1993.

57. Dursun, H., K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano and P. Vashishta, "A Multilevel Parallelization Framework for High-

Order Stencil Computations", *Proceedings of International Euro-Par Conference on Parallel Processing*, 2009.

58. Krishnamoorthy, S., M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev and P. Sadayappan, "Effective automatic parallelization of stencil computations", *Proceedings of Programming language design and implementation (PLDI)*, 2007.

59. Kamil, S., K. Datta, S. Williams, L. Oliker, J. Shalf and K. Yelick, "Implicit and explicit optimizations for stencil computations", *Proceedings of Workshop on Memory system performance and correctness*, 2006.

60. Renganarayana, L., M. Harthikote-Matha, R. Dewri and S. Rajopadhye, "Towards Optimal Multi-level Tiling for Stencil Computations", *Proceedings of Parallel and Distributed Processing Symposium*, 2007.

61. Frenkel, D. and B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications*, Academic Press, 2001.

62. Singh, J. P., W. Weber and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory", *Technical Report, Stanford University*, 1991.

63. *Teraflops Research Chip*, `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`, accessed at December 2011.

64. Borkar, S., "Thousand core chips: a technology perspective", *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007.

65. Bitirgen, R., E. Ipek and J. F. Martínez, "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach", *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2008.

66. Guo, F., Y. Solihin, L. Zhao and R. Iyer, "A Framework for Providing Quality of Service in Chip Multi-Processors", *Proceedings of 40th IEEE/ACM International*

Symposium on Microarchitecture (MICRO), 2007.

67. Eyerman, S. and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads", *IEEE Micro*, Vol. 28, No. 3, pp. 42–53, 2008.

68. Gonzalez, R. and M. Horowitz, "Energy Dissipation In General Purpose Microprocessors", *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 9, pp. 1277–1284, 1996.

69. Ding, Y., M. Kandemir, M. J. Irwin and P. Raghavan, "Dynamic Core Partitioning for Energy Efficiency", *Proceedings of the 6th Workshop on High-Performance, Power-Aware Computing (HPPAC)*, 2010.

70. Chen, J.-J. and L. Thiele, "Platform synthesis and partitioning of real-time tasks for energy efficiency", *Journal of Systems Architecture*, Vol. 57, No. 6, pp. 573–583, 2011.

71. Pan, A., O. Khan and S. Kundu, "Improving Yield and Reliability of Chip Multiprocessors", *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.

72. Bosilca, G., R. Delmas, J. Dongarra and J. Langou, "Algorithm-based fault tolerance applied to high performance computing", *Journal of Parallel and Distributed Computing (JPDC)*, Vol. 69, No. 4, pp. 410–416, 2009.

73. Soundararajan, N., A. Sivasubramaniam and V. Narayanan, "Characterizing the soft error vulnerability of multicores running multithreaded applications", *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems, (SIGMETRICS)*, 2010.

74. Liu, H., W.-M. Lin and Y. Song, "An efficient processor partitioning and thread mapping strategy for mesh-connected multiprocessor systems", *Proceedings of the ACM symposium on Applied computing (SAC)*, 1997.

75. El-Moursy, A., R. Garg, D. Albonesi and S. Dwarkadas, "Partitioning Multi-Threaded Processors with a Large Number of Threads", *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.

76. Raasch, S. E. and S. K. Reinhardt, "The impact of resource partitioning on SMT processors", *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

77. Ravi, I., "CQoS : A Framework for Enabling QoS in Shared Caches of CMP Platforms", *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, 2004.

78. Chang, J. and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors", *Proceedings of the International Conference on Supercomputing*, 2007.

79. Kamil, K., F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu and M. Valero, "Power and Performance Aware Reconfigurable Cache for CMPs", *The Second International Forum on Next Generation Multicore/Manycore Technologies*, 2010.

80. Yi-Hung, W., Y. Chuan-Yue, K. Tei-Wei, H. Shih-Hao and C. Yuan-Hua, "Energy-efficient real-time scheduling of multimedia tasks on multi-core processors", *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, 2010.

81. Srikantaiah, S., R. Das, A. K. Mishra, C. R. Das and M. Kandemir, "A case for integrated processor-cache partitioning in chip multiprocessors", *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

82. Oz, I., H. R. Topcuoglu, M. Kandemir and O. Tosun, "Reliability-Aware Core Partitioning in Chip Multiprocessors", *Journal of Systems Architecture*, Vol. 58, No. 3-4, pp. 160–176, 2012.

83. Snavely, A. and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

84. Oz, I., H. R. Topcuoglu, M. Kandemir and O. Tosun, "User-Assisted Thread-Level Vulnerability Assessment Tool", *submitted to IEEE Transactions on Computers, 2013.*

85. Reis, G. A., J. Chang, N. Vachharajani, R. Rangan, D. I. August and S. S. Mukherjee, "Software-controlled fault tolerance", *ACM Transactions on Architecture and Code Optimization*, Vol. 2, No. 4, pp. 366–396, 2005.

86. Wang, C., H. seop Kim, Y. Wu and V. Ying, "Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection", *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2007.

87. Shye, A., J. Blomstedt, T. Moseley, V. J. Reddi and D. a. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures", *IEEE Transactions on Dependable and Secure Computing*, Vol. 6, No. 2, pp. 135–148, 2009.

88. Sanchez, D., J. L. Aragon and J. M. Garcia, "REPAS: Reliable Execution for Parallel ApplicationS in Tiled-CMPs", *15th International Euro-Par Conference*, 2009.

89. Sanchez, D., J. L. Aragon and J. M. Garcia, "Extending SRT for parallel applications in tiled-CMP architectures", *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009.

90. Hyman, R., K. Bhattacharya and N. Ranganathan, "Redundancy Mining for Soft Error Detection in Multicore Processors", *IEEE Transactions on Computers*, Vol. 60, No. 8, pp. 1114–1125, 2011.

91. Ozturk, O., "Improving chip multiprocessor reliability through code replication", *Computers and Electrical Engineering*, Vol. 36, pp. 480–490, 2010.

92. Rashid, M. and M. Huang, "Supporting highly-decoupled thread-level redundancy for parallel programs", *International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

93. Gomaa, M. A. and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection", *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2005.

94. Soundararajan, N., A. Parashar and A. Sivasubramaniam, "Mechanisms for Bounding Vulnerabilities of Processor Structures", *Proceedings of the 34th annual international symposium on Computer Architecture (ISCA)*, 2007.

95. Feng, S., S. Gupta, A. Ansari and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap", *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, 2010.

96. Vera, X., J. Abella, J. Carretero and A. González, "Selective replication: A lightweight technique for soft errors", *ACM Transactions on Computer Systems (TOCS)*, Vol. 27, No. 4, pp. 40–70, 2009.

97. Reddy, V. K., E. Rotenberg and S. Parthasarathy, "Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance", *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2006.

98. Parashar, A., A. Sivasubramaniam and S. Gurumurthi, "SlicK: slice-based locality exploitation for efficient redundant multithreading", *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2006.

99. Borodin, D., B. B. Juurlink, S. Hamdioui and S. Vassiliadis, "Instruction-Level

Fault Tolerance Configurability", *Journal of Signal Processing Systems*, Vol. 57, No. 1, pp. 89–105, 2009.

100. Kumar, S. and A. Aggarwal, "Self-checking instructions: reducing instruction redundancy for concurrent error detection", *Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT)*, 2006.

101. G.Chen and M.Kandemir, "Energy-aware computation duplication for improving reliability in embedded chip multiprocessors", *Asia and South Pacific Conference on Design Automation*, 2006.

102. G.Chen, M.Kandemir and I.Kolcu, "Memory-Conscious Reliable Execution on Embedded Chip Multiprocessors", *International Conference on Dependable Systems and Networks (DSN)*, 2006.

103. Shye, A., T. Moseley, V. J. Reddi, J. Blomstedt and D. A. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance", *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

104. Braunl, T., S. Feyrer, W. Rapf and M. Reinhardt, *Parallel Image Processing*, Springer, 2001.

105. Sheaffer, J. W., D. P. Luebke and K. Skadron, "The Visual Vulnerability Spectrum: Characterizing Architectural Vulnerability for Graphics Hardware", *Proceedings of Eurographics/ACM Graphics Hardware 2006 (GH)*, 2006.

106. Mattson, T. G., B. A. Sanders and B. L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2004.

107. Kadayif, I., H. Sen and S. Koyuncu, "Modeling soft errors for data caches and alleviating their effects on data reliability", *Journal Microprocessors and Microsystems*, Vol. 34, No. 6, pp. 200–214, 2010.

108. Bosilca, G., A. Bouteiller, A. Danalis, T. Hèrault, P. Lemarinier and J. J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing", *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011.

109. Li, L., V. Degalahal, N. Vijaykrishnan, M. Kandemir and M. J. Irwin, "Soft error and energy consumption interactions: a data cache perspective", *International symposium on Low power electronics and design (ISLPED)*, 2004.

110. Jeyapaul, R. and A. Shrivastava, "Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors", *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011.

111. Zhao, H., M. Kandemir and M. J. Irwin, "Exploring Performance-Power Tradeoffs in Providing Reliability for NoC-Based MPSoCs", *International Symposium on Quality Electronic Design (ISQED)*, 2011.

112. Ozturk, O., M. Kandemir, M. J. Irwin and S. H. K. Narayanan, "Compiler Directed Network-on-Chip Reliability Enhancement for Chip Multiprocessors", *Conference on Languages, compilers, and tools for embedded systems (LCTES)*, 2010.