# SSFT: SELECTIVE SOFTWARE FAULT TOLERANCE

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Tuncer Turhan

January, 2014

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Özcan Öztürk(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Bedir Tekinerdoğan

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

# ABSTRACT

# SSFT: SELECTIVE SOFTWARE FAULT TOLERANCE

Tuncer Turhan
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Özcan Öztürk
January, 2014

As technology advances, the processors are shrunk in size and manufactured using higher density transistors which makes them cheaper, more power efficient and more powerful. While this progress is most beneficial to end-users, these advances make processors more vulnerable to outside radiation causing soft errors which occur mostly in the form of single bit flips on data. For protection against soft errors, hardware techniques like ECC (Error Correcting Code) and Ram Parity Memory are proposed to provide error detection and even error correction capabilities. While hardware techniques provide effective solutions, software only techniques may offer cheaper and more flexible alternatives where additional hardware is not available or cannot be introduced to existing architectures. Software fault detection techniques -while powerful- rely mostly on redundancy which causes significant amount of performance overhead and increase in the number of bits susceptible to soft errors. In most cases, where reliability is a concern, the availability and performance of the system is even a bigger concern, which actually requires a multi objective optimization approach. In applications where a certain margin of error is acceptable and availability is important, the existing software fault tolerance techniques may not be applied directly because of the unacceptable performance overheads they introduce to the system. Our technique Selective Software Fault Tolerance (SSFT) aims at providing availability and reliability simultaneously, by providing only required amount of protection while preserving the quality of the program output. SSFT uses software profiling information to understand application's vulnerabilities against transient faults. Transient faults are more likely to occur in instructions that have higher execution counts. Additionally, the instructions that cause greater damage in program output when hit by transient faults, should be considered as application weaknesses in terms of reliability. SSFT combines these information to eliminate the instructions from fault tolerance, that are less likely to be hit by transient errors or cause errors in program output. This approach reduces power consumption

and redundancy (therefore less data bits susceptible to soft errors), while improving performance and providing acceptable reliability. This technique can easily be adapted to existing software fault tolerance techniques in order to achieve a more suitable form of protection that will satisfy different concerns of the application. Similarly, hybrid and hardware only approaches may also take advantage of the optimizations provided by our technique.

*Keywords:* Software Fault Tolerance, Software Fault Injection, Software Profiling for Reliability, Reliability, Multi objective optimization: Reliability and Availability.

# ÖZET

# SEÇİMSEL YAZILIM HATA TOLERANSI

Tuncer Turhan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doç. Dr. Özcan Öztürk
Ocak, 2014

Teknolojik gelişmelerle birlikte, işlemciler boyut olarak daha küçültülüyor ve üretim sürecinde daha sık ve küçük boyutlu transistorler kullanılarak üretiliyorlar. Bu üretim süreci işlemcileri daha ucuz, daha güç tasarruflu ve daha güçlü kılıyor. Bu süreç son kullanıcı için son derece faydalı olmasına karşın, bu süreç işlemcileri dış ortamdan kaynaklı radyasyona karşı daha zayıf kılıyor ve bunun sonucunda, genellikle veri üstünde tek bir bitin değer değiştirmesi formunda oluşan, yumuşak hatalar oluşuyor. Yumuşak hatalara karşı uygulamaların güvenilirliğini arttırabilmek adına, literatürde ECC (Hata Düzeltme Kodu) özellikli veya Parite özellikle hafıza gibi donanımsal hata tolerans teknikleri geliştirilmiştir. Donanımsal hata tolerans teknikleri etkili çözümler üretmesine karşın, donanımsal altyapının bulunmadığı ya da var olan sisteme eklenmesi mümkün olmadığı durumlarda, yazılımsal hata toleransı teknikleri daha ucuz ve esnek bir alternatif sunabilir. Yazılımsal Hata Toleransı teknikleri, güçlü bir alternatif olmasına rağmen, genellikle yedekleme mantığına dayalı çalıştıklarından, performans düşüşüne ve hataya neden olmaya açık olan bit sayısını arttırmaktadır. Uygulama güvenilirliğinin bir endişe olduğu sistemlerde, performans ve erişiliebilirlik daha büyük bir sistem endişesi ve gereksinimi durumunda olduğundan, bu çoklu objektifli bir yaklaşım gerektirmektedir. Belirli ölçüde bir hatanın kabul edilebilir olduğu ve erişilebilirliğin önemli olduğu uygulamalarda, sisteme getirdikleri performans yükünden ötürü, literatürdeki yazılımsal hata toleransı tekniklerini olduğu gibi kullanılamayabilir. Bu noktada bizim tekniğimiz seçimsel yazılım hata toleransı (SYHT) erişilebilirlik ve güvenilirliği eş zamanlı sağlamayı hedefler. SYHT bunu, uygulamanın sadece ihtiyaç duyduğu ölçüde hata toleransı kullanarak ve uygulamanın ürettiği verilerde kaliteyi koruyarak sağlamaktadır. SYHT yazılım profil bilgisini kullanarak, uygulamanın yumuşak hatalara karşı hassasiyetlerini anlamaya çalışır. Yumuşak hatalar yüksek sayıda çalışan komut satırlarında oluşmaya meyillidir. Ayrıca, yumuşak hatalara maruz kaldığında, uygulama tarafından oluşturulan verilerde daha fazla hataya sebep olan komut

satırlarının uygulamanın güvenilirlik açısından hassas komut satırları olduğu söylenebilir. SYHT bu bilgileri kullanarak, yumuşak hatalara maruz kalma olasılığı düşük ve uygulama tarafından oluşturulan verilerde daha az hataya sebep olan komut satırları için hata toleransını kaldırır. Bu yaklaşım, performansı arttırırken ve güvenilirliği yeterli seviyede tutarken, enerji tüketimlerini ve yedeklenen veri sayısını (dolayısıyla yumuşak hatalara maruz kalan bit sayısını) azaltır.Bu teknik kolaylıkla literatürde yaygın olan yazılımsal hata toleransı tekniklerine adapte edilebilir. Bu adaptasyonu yaparken de, uygulamanın çeşitli endişelerine uygun olacak şekilde bir hata toleransı kullanır. Benzeri şekilde, hibrit ve donanımsal hata toleransı yaklaşımları da bizim yaklaşımımızın sağladığı iyileştirmelerden faydalanabilirler.

*Anahtar sözcükler*: Yazılımsal Hata Toleransı, Yazılımsal Hata Enjeksiyonu, Yazılım Güvenliği için Yazılımsal Profil Çıkarma , Yazılım Güvenilirliği, Çoklu Objektifli İyileştirme: Güvenilirlik ve Erişilebilirlik.

# Acknowledgement

I am much obliged to my supervisor Assoc. Prof. Dr. Özcan Öztürk, for his understanding and guidance through this experience. His ideas were extremely helpful to me, in each step of my studies.

I am also grateful to my jury members, Assist. Prof. Dr. Bedir Tekinerdoğan and Assoc. Prof. Dr. Süleyman Tosun for their participation and their invaluable comments and suggestions.

I am grateful to Computer Engineering Department of Bilkent University for providing me tuition waiver for my MS studies.

I am thankful to Central Bank of Turkey for the allowance they provided during this study.

I would also like to thank my friends for their understanding and support throughout this experience, Osman Değer, Emir Gülümser, Gülden Törer, Seçkin Okkar, Ethem Barış Öztürk, Nesim Yiğit and Muhammed Büyüktemiz.

I would also want to thank my entire family for their support, without them and their prayers, this whole study would be for nothing.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Over the last decade, the processors have improved in many aspects through technological advancements; they have become cheaper, more powerful and less energy consuming and overall offer better and more efficient computing. In order to provide more efficient and powerful processors, hardware manufacturers keep improving their designs and fabrication technologies. There are many improvements being applied; however the following aspects are more important for this thesis: using higher density transistors and introduction of Chip Multiprocessors (CMP). The current state of the art technology is 14nm process technology which is adopted by most of the processor and System on Chip manufacturers including Intel, AMD, Nvidia and ARM. The number of transistors on integrated circuits doubles approximately every two years according to Moore's Law and to provide such advancements companies increase the transistor densities. Although, these advancements provide users cheaper, faster and more efficient processors, there are issues that need to be addressed in order to continue with such advancements. Baumann states that, "As the dimensions and operating voltages of computer electronics shrink to satisfy consumers' insatiable demand for higher density, greater functionality, and lower power consumption, sensitivity to radiation increases dramatically." [1]. According to recent studies [2] soft errors are expected to grow further as the scaling goes beyond 14nm, and with each generation 8% increase in soft-error rate is expected. This sensitivity to radiation

presents itself in the form of Single Event Upset (SEU). While these faults, in general, are considered as transient errors and do not cause any permanent damage on the hardware, a single bit flip in data may cause significant failures. In software systems, application programs, operating systems or drivers are considered as main causes of faults; however, in some cases, these transient faults may be the source of the actual failure. In 2000, Sun Microsystems acknowledged that cosmic rays interfered with cache memories and caused crashes in server systems at major customer sites, including America Online, and dozens of others [3, 4]. In a more recent event, Hewlett Packard stated that cosmic ray strikes causing transient faults was the main cause of the frequent crashes in 2048-CPU server system in Los Alamos National Laboratory [5]. For brevity and keeping focus on the concerns addressed in this thesis, the details about the formal definition of the transient faults and actual reasons behind faults will not be further discussed.

In order to prevent failures caused by transient faults, the general convention is to use bit level protection techniques like ECC (Error Correction Code) or EDAC (Error Detection and Correction Code) in the memory architecture. In order to comply with the needs of ECC, a circuitry that is capable of encoding (Hamming Code, Reed Solomon) the data (encoded data is used for error detection and recovery), additional data space to store the encoded data bits, error checking and recovery mechanisms are introduced to existing memory architectures. For error detection and recovery, the Hamming Code encoding requires 8 extra bits to be encoded and stored for each of the 64 bit cells in the memory architecture. The added circuitry will use these encoded data bits for error detection and recovery purposes.

ECC hardware implementing Hamming Code is able to detect 2 bit errors and can correct 1 bit errors in memory which is called SECDED (single error correction, double error detection). This SECDED is the convention in ECC because; in order to recover from multiple bit errors a higher number of bits needed to be stored, more hardware will be introduced to system for encoded data calculation and data recovery. These additional requirements will result in an even more expensive and slower system which is impractical in most cases. Multiple bit failures are considered to happen much less frequently compared to SEU, which makes SEU detection by far the most important concern [4].

While ECC offers great level of protection and recovery, it is sometimes omitted for being costly and increasing the access times in memory [6]. Parity RAM is another hardware solution, which requires less hardware than ECC. However, faults can only be detected, but not corrected with this protection. In general, Parity RAM is also considered to be costly and slower than RAM that is not providing any protection, therefore may also be omitted. The convention is that, memory units that are lower in the memory hierarchy, such as L1 and L2 caches, are equipped with parity protection. In case of a failure, the data is restored from its original location in RAM. Although most of the memory hierarchy is seemingly under protection, there are parts inside the CPU architecture that are not being protected (due to limitations of hardware fault tolerance techniques) and hence are open to transient faults. ECC and parity bit protection techniques cannot be applied to most parts of the CPU architecture and are often criticized as they are not scalable to address the reliability concerns of the entire computer architecture.

To give a more specific example, consider a system having ECC protection in its RAM memory and parity protection in its cache level memory. Any SEU on the data located in the RAM; will be detected and corrected before it can cause any faulty behavior. The L1 and L2 caches will detect any SEU using the parity bits and restore its data from the RAM which is known to be protected by ECC. However, when faults occur inside the ALU (Arithmetic Logic Unit) or in the instruction fetch-decode unit or the registers inside the CPU, the fault detection and recovery is not possible. ECC protection for these internal parts of the CPU is known to be costly, power consuming and slow, and thus is considered not scalable and usually not adopted by CPU manufacturers. For instance, protection of the data in a CPU register file using ECC is shown to be extremely costly in terms of both performance and power [7]. For ALU, such protection will disrupt the pipeline architecture, impair the performance of the whole processor while increasing the power consumption and the cost. Other alternatives include using the pipelined structures inside the CPU and executing the same instruction twice and delay the output until the result is verified by the second execution. Similarly, VLIW (Very Long Instruction Word) architecture is able to take advantage of ILP (Instruction Level Parallelism) and can be used in

order to execute the same instruction twice and comparing the results. VLIW is a very common architecture, especially in GPUs (Graphics Processing Unit) which implement SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data) on a manycore architecture. Since, GPU is a many-core architecture that can process multiple data in an extremely fast manner; it has become a new alternative for General Purpose Computing and referred to as GPGPU (General-Purpose GPU). In both VLIW and GPU, executing the same instruction multiple times will have an impact on the ILP and impair the system performance dramatically, while decreasing the availability.

The bit level hardware protection is not commonly adopted in the low-level hardware hierarchy due to aforementioned concerns. The manufacturers are often forced to implement high level protections, by which they are able to obtain promising results with less severe impacts on the performance and cost. These architecturally high level approaches are called "macro-reliability protection" [7]. Macro-reliability protection often uses duplication of coarse-grained structures such as CPU cores or hardware contexts inside the processing unit to provide transient fault tolerance in a more cost-effective and scalable manner [7]. While this approach overcomes the scaling problems of the prior bit-level techniques, macro-reliability schemes adopt a rather inflexible one size fits all protection over the whole CPU architecture. This strict protection scheme will not be able to adapt different levels of performance and reliability requirements and most of the time will end up overprotecting the entire system. This overprotection will be reflected to the end-user as higher power consumption levels with significant losses in performance and availability while increasing the overall cost. The end-users may want to use the same underlying hardware for different types of applications having different levels of reliability requirements, while in some applications faults are intolerable, in others imprecise results may be acceptable. Similarly, the user may want to upgrade the system configuration to increase the overall reliability or performance of the system, in order to adapt rapidly changing requirements of the market. These system upgrades will be much more costly because of the one size fits all protection approach. Moreover, in most cases the underlying hardware cannot be modified and the user may still want to improve the level of protection from the transient faults. These examples can be multiplied, but concisely, the

reliability and performance concerns need to be handled in a more adaptable way to fulfill the specific performance and reliability requirements of an application. This is where software fault tolerance (SFT) techniques become more appealing alternatives with promises of providing a more flexible way for users to adjust the reliability and performance levels according to their needs and in case of a lack of underlying hardware support.

SFT techniques rely on some form of redundancy similar to Hardware Fault Tolerance (HFT) techniques. SFT techniques use different forms and levels of redundancies. For instance, a bit-level approach in software can be used in terms of error-detection and recovery, just like hardware ECC and parity protections. This will require the application to handle encoding of Hamming Code (or Reed Solomon) extra bits, store them in memory, check in case any error occurs and recover the data. In practice, the software approach of ECC will perform worse than hardware ECC, since bit level encoding calculations can be better implemented in hardware as well as, detection and recovery calculations. Since the calculations cannot be directly injected in the main thread due to performance overheads, the software ECC will be performed in another thread in the form of periodical sweeps to data residing in memory. Rapidly changing data in memory cannot be addressed with software ECC, since the recovery data needs to be recalculated in each data change. The parts of the memory where data is more immutable is rather convenient for this protection scheme like L1 instruction cache where the running application resides. The data in memory is partially protected and the protection of the application code is done without any distinction. Additionally, any error occurring between the sweeps will still affect the execution and even spread throughout the rest of the program.

Another SFT technique is instruction duplication, where each instruction is duplicated and the results of the duplicate instructions (shadow instructions) are compared with the main instruction's results at synchronization points. The synchronization points in the execution cycle are mainly chosen as the store instructions which store the data from registers to memory. The main idea is to keep the data in the memory intact, detect errors before data is written back to memory, and prevent any corruption. The instruction duplication is expected to double the execution latency of the application, however with the use of ILP (Instruction

Level Parallelism) techniques and some additional improvements the performance overhead can be reduced. For instance, SWIFT (an SFT instruction duplication technique), is able to improve execution latency to 1.41x compared to the baseline where no SFT is applied [4]. Similar to performance overheads, other concerns emerge with instruction duplication. Such as code size and volatile memory requirements. Instruction duplication scheme can also be used in hybrid techniques, where hardware supports software implementation. Hybrid techniques were proposed to tackle the performance bottlenecks of SFT-only instruction duplication techniques, with provided additional hardware assistance.

In addition to software redundancy at the instruction level, it is also possible to implement redundancy at the thread level, where an identical copy of the main thread runs for reliability. Two different types of redundant thread mechanisms come to mind, running the redundant thread in the same CPU that supports simultaneous multithreading capabilities (SMT) and running the redundant thread in another CPU core which is possible in Chip Multiprocessor (CMP) systems. The redundant thread introduces a synchronization problem between the threads since it requires a slack between leading and trailing threads in which the trailing thread will follow the leading thread. The trailing thread will detect and recover from faults that may occur. The redundant thread mechanism in CMP is also referred to as CRT (Chip level redundantly threading). CRT scheme also introduces a communication overhead, due to data needed by multiple processors. Therefore, it requires require additional hardware queues to be implemented inside the CPU.

The details of the SFT techniques described above will be discussed in more detail in the related work section. The novel idea behind the SFT techniques is that these techniques provide a more flexible, cheaper alternative to HFT methods. However, despite its advantages, these systems overprotect the application as a whole without taking advantage of the application and hardware characteristics. In addition to performance overhead, volatile and non-volatile memory requirements increase; additional hardware requirements emerge with the use of SFT techniques, which defeats the whole purpose of using them. Moreover, the redundancies introduced to system (shadow instructions, hardware queues, etc.) increase the number of data bits that are vulnerable to SEUs, thereby leading to

a higher soft error rate [8].

Based on the drawbacks of current SFT techniques, it is necessary to take a fresh look at software fault tolerance, where reliability and performance requirements of the running applications considered. More specifically, SFT schemes should aim at a balanced protection, that provides required levels of protection while decreasing the cost and performance overheads. In order to achieve a balanced protection, SFT techniques should be applied selectively. In the rest of this thesis, this is called selective SFT (SSFT). In order to selectively apply fault tolerance, the application should be carefully analyzed using software profiling information.

There are two different analysis in using software profiling information for reliability, static and dynamic. Static analysis uses the offline information that is obtained during the compilation process of the application code. Compilers often uses passes to analyze the application. The application code is first parsed and converted in to structure called Control Flow Graph (CFG). CFG consists of basic blocks that are connected by edges. Basic blocks are straight lines of code, that does not contain any jump instructions. The jumps between basic blocks are provided by edges that connect basic blocks. CFG shows the paths that can be traversed during the execution of the program. After CFG is formed, compiler processes the application code to eliminate dead code and optimize program execution. An example for static analysis can be type inference for fault-tolerance prediction [9]. This study analyzes the instruction operand types, an information that can acquired during compilation. EPIC, another technique using static analysis, uses error propagation and CFG data to understand the impacts of a soft error [10]. Although, offline analysis provide invaluable information to understand the application characteristics in terms of reliability, an online analysis may provide a different perspective. SSFT follows an orthogonal path, using run-time information to understand the effects of soft-errors.

For dynamic analysis of an application, we use statistical data that is produced during program execution. The statistical data provides us, the paths that application traverses in CFG and the the execution counts for each basic block. The statements that are located in basic blocks with high execution counts, are

more vulnerable to transient faults. Moreover, the output produced by the application can be analyzed in order to understand the weaknesses of the application. The statements that causes heavy damage in program output, when distrupted by a SEU, should be protected by some form of fault tolerance in order to preserve the reliability. Additionally, the hardware and environmental conditions can also be an effective factor, when considering the rate of transient faults.

SSFT will use aforementioned dynamic analysis, to selectively protect the code segments that are most likely to damage software reliability. The code segments that are less likely to damage application execution, can be removed from fault tolerance without impairing reliability. During this selection, the amount of redundancies introduced to the system are reduced, while, the probability of transient fault occurrence and specific output quality requirements of the user are considered. SSFT will increase the performance, decrease hardware requirements and therefore cost, while effectively preserving the software reliability. The motivations behind SSFT, briefly discussed above, will be explained in depth with examples in the next chapter.

# Chapter 2

# Motivation

The motivation behind using SFT techniques is that they are more flexible and cheaper unlike their HFT counterparts. The HFT techniques protect the whole CPU hardware without considering the running application. Moreover, HFT techniques are expensive, and therefore not scalable (especially bit-level techniques like ECC or Parity). Furthermore, they increase the access times of volatile memory, thereby decreasing the availability and increasing the power usage. While most of these overheads are valid, it will be shown that SFT techniques do not completely overcome these problems.

To overcome the limitations of SFT, we propose Selective SFT, where we choose specific portions of the application that are most vulnerable to transient faults. Specifically, we profile the application and apply SFT to program segments that are likely to cause the most damage to program execution. By careful selection of these program portions, the overall output quality is preserved with minimal safekeeping.

Both EDDI [11] and SWIFT [4] protection schemes use instruction duplication for each and every line of the code. They use different registers for the duplicated instructions and faults are detected at synchronization points (store instructions). Similar to HFT techniques, both of these techniques overprotect the whole CPU hardware without any distinctions. Application binary and data is protected in a unified manner with no distinction.

One can observe that not all parts of the code have the same importance level. For instance, a register may contain debugging data that does not affect the outcome of the program. An error occurring in this register, will not affect the application output. Similarly, a register file that is masked by an "AND" instruction, will not cause any errors in the program output if the transient fault occurs in the masked bits. Moreover, a register file that contains obsolete information (i.e. a loop variable after the loop is execution is finished), will not affect the software execution in case of a transient fault occurrence. As a last example, data in the register file may contain highly precise information which is not required for that application (Double precision data is not vital for floating point calculations). Any error occurring, in high precision bits will not impact the output. In each of these examples, the application will resiliently recover from the soft errors occurring in these sections without any serious impact of the application outcome, therefore totally abandoning duplicated instruction approach or using a more lightweight protection scheme for these sections will not have major impact on software reliability. However, no such distinctions are made in duplicated instruction schemes, all instructions and data are duplicated and protected.

HFT methods are known to be expensive and not scalable, whereas SFT methods are expected to be cheaper and not require additional hardware. EDDI, as one of the initial single threaded SFT techniques, has a geometric mean of 1.62x, whereas SWIFT has 1.41x execution time compared to the baseline without any fault tolerance. Therefore, reliability is achieved at the expense of availibility and performance. In other words, SWIFT requires a better performing CPU (1.41x) to achieve the same performance levels. In addition to performance, memory footprint will also be affected. Specifically, application binary size will be 2.4x larger compared to the baseline SWIFT implementation [4]. Moreover, the extra shadow instructions will increase the pressure on CPU registers, cache, and RAM.

The issues above show that instruction duplication technique also has its own drawbacks similar to HFT. While these issues cannot be ignored or resolved completely, their impacts can be safely reduced, especially for applications that are more resilient to soft errors. Consider an application in which the calculations

do not require 100% precision, that is an imprecise or approximate result is acceptable. These types of computations are commonly used in soft computing applications, which are naturally more resilient to soft errors since an exact result is not always essential. For an application where 95% precision is sufficient, a protection scheme that protects some parts of the application and gives 95% precision will be sufficient. According to SEU model, a bit is defined as ACE (Architecturally Correct Execution) if a transient fault affecting that bit will cause the program to execute incorrectly [12]. When the redundancy is reduced in the system, the number of bits susceptible to soft errors will decrease, hence the number of ACE bits will decrease.

Another example that can make use of SFT is software ECC, where an ECC encoding is done over the instruction cache memory [6]. The software ECC thread is a high priority thread that detects and corrects errors in the running application code. The sweep performance affects the overall availability and performance of the system since other processes are halted during the sweeps. Profiling information about the running application and selectively choosing the instruction data to protect will result in less sweep time, therefore will put less pressure on the system performance and eventually increase the availability.

Thread-level redundancy which runs an identical copy of the main thread for error detection requires a slack between the main and trailing thread. CRT (Chip level redundantly threading) systems which may be preferable for performance (2 CPUs are running the threads instead of one) and reliability concerns, since the trailing thread will be running in a CPU that is physically far away from the CPU that the main thread is running. This way overheads due to running threads will be reduces while eliminating the possibility of a fault corrupting both threads [13]. However CRT has a major impact on inter thread communication in that the thread communication transforms into inter processor communication which requires fast communication channels. This communication overhead is hidden by enabling a longer slack between redundant threads, which effectively stalls the main thread. When SSFT idea is applied to CRT or CRTR (Chip-level Redundantly Threaded multiprocessor with Recovery), the trailing thread will not require the same amount of resources as the original thread and the resources seized by the trailing thread can be safely released when trailing thread

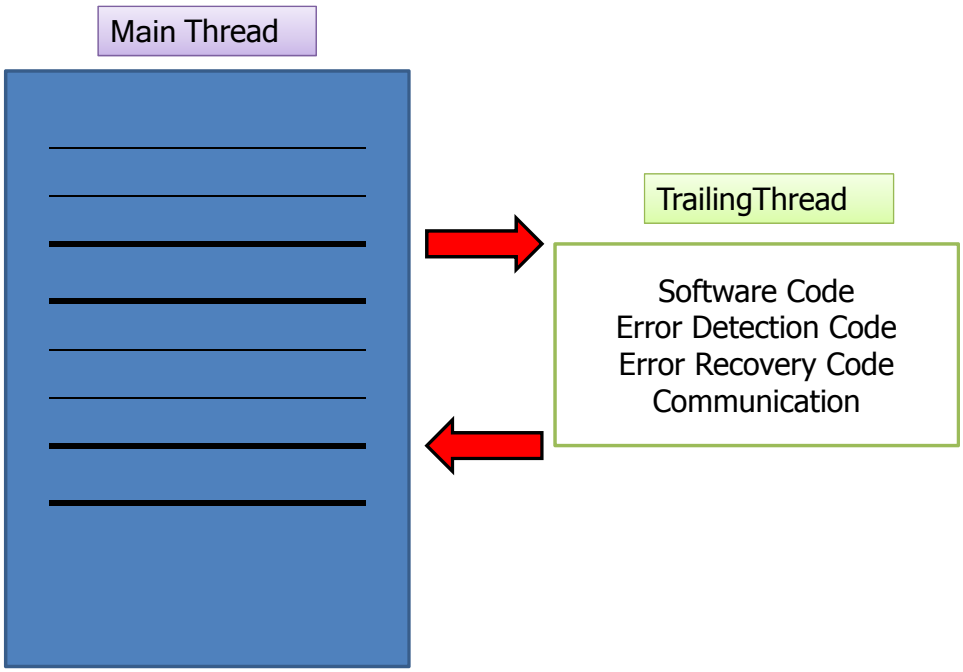


Figure 2.1: Improvements for CRT.

is not used. The slack between main and trailing thread can be reduced through SSFT which will improve the overall performance, reduce power consumption and resource requirements. Similar to the SFT techniques presented above, the improvements and the time saved can be put to better use for recovery purposes. Figure 2.1 shows how SSFT can be applied to CRT, where bold lines represent a redundant fault detection and recovery thread. As can be seen from the figure, trailing thread does not need to be an exact replica of main thread, i.e, the lines that are not bolded ar not executed by trailing thread. The spare time achieved by not executing these statements, can be used for recovery and to compensate communication delays.

Selectively applying reliability can also be implemented to accommodate hardware fault tolerance techniques. The compiler previously informed about the underlying hardware and the running application can make informed decisions about where in memory each instruction should be placed. The instructions or data that are more resilient to soft errors can be placed in locations that are unprotected or only protected with parity, whereas the instructions that are most likely to cause great damage are placed in ECC covered memory locations. In

this manner, the costs can be reduced and HFT techniques can be applied more effectively.

Beyond the benefits brought to fault tolerance techniques, SSFT mainly utilizes the key fact that not all applications require the same amount of reliability. Some applications are actually more tolerant to imprecisions and approximations which make them more resilient to transient faults. These applications (referred to as Soft Computations) may require some amount of reliability when they also have conflicting performance and availability concerns. In stream processing applications, financial calculations and even some safety critical systems, where a margin of error is acceptable. The SSFT idea is feasible to apply and potentially will bring a balance between performance and reliability.

The arguments presented above constitute the motivational base for profiling the running application and selectively applying the fault tolerance techniques. The details about fault injection, software profiling, fault detection and recovery will be discussed in the next section.

# Chapter 3

# Related Work

## 3.1 Fault Injection

The fault injection techniques follow various paths in simulating the transient faults that are caused by cosmic radiation. MEFISTO [14], VERIFY [15] and DEPEND [16] tools inject faults into a simulation model. RIFLE [17] and MESSALINE [18] tools inject faults at hardware pin-level. FIAT [19], and FERRARI [20] are tools that inject faults into physical systems using software implemented fault injection (SWIFI). The SWIFI idea brings a new perspective to fault injection in that, the faulty conditions can be simulated without hardware requirements. These tools have the general problem of being specifically designed for a certain hardware and therefore cannot be adapted to different configurations. Tools that are more adaptable emerged later on, such as NFTAPE [21], GOOFI [22], PROPANE [23] and SWIF-IT [24].

NFTAPE tool supports multiple fault models (bit-flips, communication and IO errors), multiple fault event triggers (path-based, time-based, and event-based triggers), multiple targets (distributed applications, software implemented fault tolerance (SIFT) middleware layer, black box applications, communication interface, and operating system) and supports memory dumps when required. [21]

GOOFI tool on the other hand, is an object-oriented JAVA and SQL based

tool, in which user can use existing fault injection techniques or extend the tool by defining their own and run fault injection tests. The tool targets Thor RD microprocessor which is a SAAB Ericsson Space AB processor and is created solely for highly dependable space applications [22].

Propagation Analysis Environment (PROPANE) is a software profiling and fault injection tool for applications running on desktop computers. PROPANE supports the injection of both software faults (by mutation of source code) and data errors (by manipulating variable and memory contents) [23]. PROPANE's capabilities for software profiling, is more focused on the error propagation characteristics of the running application.

SWIF-IT is more of a kernel level tool developed to run under Linux, which injects fault at memory locations and inspects the impacts of the injected fault. The fault injector being implemented within the kernel has the limitation that memory corruptions are restricted to the kernel's view of the hardware meaning that corruption of a data structure inside the process table is easier compared to a data in a specific memory location [24]. Additionally, this tool offers error recovery schemes for the faulty memory locations. The recovery schemes are based on simple redundancy techniques like Hamming Code and majority voting, in which multiple copies of the same data is stored and in case of an error it is recovered using the value that has the majority.

While these software implemented fault injection (SWIFI) tools provide some level of fault injection capabilities, we have implemented our own fault injection and testing tool which provides us the required data for our approach.

## 3.2 Fault Detection and Recovery

When considering transient fault detection and recovery, there are three main classes of techniques: hardware fault tolerance (HFT), software fault tolerance (SFT) and hybrid techniques (software implemented hardware supported).

### 3.2.1 Hardware Fault Tolerance

HFT techniques can be further categorized in to two, bit-level approaches and macro-reliability approaches. Bit-level approaches mostly rely on redundantly storing extra data bits for the data in memory in order to detect and recover from any transient faults. One of the simplest approaches for error detection is parity. Parity for a data is calculated by simply applying the XOR operation on the data bits. For instance, even and odd parity for the following 7 bit data "1101011" is calculated as follows.

$$1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1 \quad \text{Even Parity} \tag{3.1}$$

$$\sim (1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1) = 0 \quad \text{Odd Parity} \tag{3.2}$$

The idea is store the parity information in memory or in data transactions so that a single bit error in data can be detected. For instance, in an ASCII data transmission, the resulting parity bit is added as the 8th bit data and the data will be sent as "11010111" where the last bit is the even parity bit. The receiver will check the received data by simply applying the parity calculation on data and comparing the calculated parity bit with the parity bit received. In case a parity error is detected, data transmission is repeated. Similarly, this is also used in memory hierarchy. Especially, CPU caches adopt this idea in hardware so that in case of an error inside the data in the cache, data is invalidated and requested from memory. Parity bit is able to detect a single bit error inside the data; however, it does not offer any recovery options.

For recovery purposes, different error detection and recovery schemes have

| Check Number | Check Positions | Positions Checked |
|---|---|---|
| 1 | 1 | 1,3,5,7,8,11,13,15,17... |
| 2 | 2 | 2,3,6,7,10,11,14,15,18... |
| 3 | 4 | 4,5,6,7,12,13,14,15,20... |
| 4 | 8 | 8,9,10,11,12,13,14,15,24... |
| . | . | . |
| . | . | . |
| . | . | . |

Figure 3.1: Check positions for Hamming Code.

been proposed and used in the literature. Hamming Code, Reed-Solomon Code and other cyclic code schemes are commonly used in the industry for this purpose. For brevity, only Hamming code (the recovery method adopted in ECC memory) will be explained here. Hamming code can be used to detect and correct single bit errors, and with an additional parity bit added, it can also detect double errors. The idea is to put parity protections on the positions that are powers of two starting from the first bit position. For a 64 bit data, the parity will be placed on 1, 2, 4, 8, 16, 32, and 64 check positions which will make a sum of 7 parity bits. The check positions and the positions checked are shown in Figure 3.1.

For instance, a check position 4 (100 in binary) has the check number 3 meaning that, all the data bit positions having 1 in their 3$^{rd}$ bit needs to be added in parity calculation, which are **1**00 (4), **1**01 (5), **1**10 (6), **1**11 (7), 1**1**00 (12) and

so on. As an example for detection and recovery, consider a 7 bit data, 1101011, and calculate the parity for check positions 1, 2 and 4, powers of two up to the total number of data bits. The bits are numbered from right to left starting from one and the data in position x is shown as $d_x$.

$$P_1 = d_1 \oplus d_3 \oplus d_5 \oplus d_7 = 1 \oplus 0 \oplus 0 \oplus 1 = 0 \tag{3.3}$$

$$P_2 = d_2 \oplus d_3 \oplus d_6 \oplus d_7 = 1 \oplus 0 \oplus 1 \oplus 1 = 1 \tag{3.4}$$

$$P_3 = d_4 \oplus d_5 \oplus d_6 \oplus d_7 = 1 \oplus 0 \oplus 1 \oplus 1 = 1 \tag{3.5}$$

Consider the case when a single bit flip occurs in bit location 6, then the parities will look like as the following.

$$P_1 = d_1 \oplus d_3 \oplus d_5 \oplus d_7 = 1 \oplus 0 \oplus 0 \oplus 1 = 0 \tag{3.6}$$

$$P_2 = d_2 \oplus d_3 \oplus d_6 \oplus d_7 = 1 \oplus 0 \oplus 0 \oplus 1 = 0 \tag{3.7}$$

$$P_3 = d_4 \oplus d_5 \oplus d_6 \oplus d_7 = 1 \oplus 0 \oplus 0 \oplus 1 = 0 \tag{3.8}$$

The location of the error is determined by comparing the parity values. $P_1$ is correct, therefore we write a 0; $P_2$ is incorrect therefore we write a 1 to the left, obtaining 10; $P_3$ is incorrect as well, therefore putting another one to the left we end up with 110 which indicates the data in the $6^{th}$ bit position is flipped. This way, the Hamming Code is able to perform 1 bit error detection and 1 bit data recovery. In order to have single error correction, double error detection (SECDED), an even data parity is added to 7 bit data (XOR of all data bits).

Although bit level approaches offer detection and recovery options, they are often found not to be scalable, and therefore have not been used widely in CPU architectures. This is due to their impact on performance and power consumption. The alternative is to follow a higher level approach called macro-reliability. Some of the older systems like HPs NonStop Cyclone System [25] or IBMs S/390 G5 processor [26] or Triple redundant 777 primary flight computer [27], all rely on redundant CPU cores (Boeing 777 even has multiple ARINC data buses), through

which faults are detected, corrected. Even in case of a total hardware failure of a single hardware, these systems promise to operate uninterrupted. There are also simpler approaches in duplicating hardware for redundancy such as extra pipelines in processors, queues for memory loads and stores, extra branch predictors [13, 28, 29, 30], additional data bits for detection of possibly incorrect data [31].

Recently, with the emergnce of Chip Multiprocessors (CMP), hardware duplication and usage has become much more attractive. However, the he additional hardware is required to overcome the difficulties in communication, race conditions and other problems that arise with reliability concerns.

Intel Itanium proessor is a good example of hardware reliability scheme implementation. Specifically, Itanium processor offer parity protection in low-level caches, ECC protection in high level caches. Errors in pipelines are detected using residues, which are calculated during mathematical operations, or using parity bits [32]. The instruction level faults are detected and corrected inside the pipelining architecture. Whenever a soft error occurs, the instruction is simply re-read correctly from the instruction buffer and restarted through the pipeline as if the error had never occurred (referred to as replay). An error occurring in the instruction buffer is handled in the following manner; all instructions in the instruction execution pipeline, the instruction buffer, and the instruction fetch pipeline are removed, then the erratic instruction and the instructions after it are re-read from cache (referred to as refetch). Soft errors in translation lookaside buffers (TLBs) and in general purpose and floating point register files are handled by firmware which eventually restarts the OS or application to resume running as if no errors occurred (referred to as resteer). The technique described above is handled by hardware without any software interventions.

### 3.2.2 Software Fault Tolerance

Software fault tolerance techniques mostly rely on redundant data and computations and comparison of the original and redundant data (or calculation). The recovery of the faulty data is generally addressed as a separate problem. For

software fault detection, there are single threaded techniques that duplicate the instructions like EDDI [11] or SWIFT [4]. The instruction duplication was presented first as an ALU instruction duplication scheme. The idea was to increase reliability, by keeping the redundancy as low as possible, simply by using the same registers and abuse the VLIW architecture's ability to execute the same instruction on multiple data and compare the outcomes that are executed by the same CPU instruction [33].

In order to generalize this approach and protect all instructions inside the system with better reliability, error detection by duplicated instructions (EDDI) was proposed. EDDI simply duplicates all the instructions and reorders the duplicated (or shadow) instructions that occur before the store instructions, so that the instruction level parallelism (ILP) capabilities of CPU are utilized. The results of the main instruction (MI) and shadow instruction (SI) are compared before the store instructions so that the data in memory is always kept intact. All store and load instructions (memory operations) are duplicated and the only control over control flow errors relies on the count of instructions that are run by shadow and main instructions. According to the experimental results, EDDI has an execution time of 1.61x compared to the baseline execution [4, 11]. Normal duplication of the instructions or execution of the same instructions twice is expected to have a 2x execution latency, however exploiting ILP, the performance is improved by 20% percent. However, this improvement all depends on the ILP capabilities of the underlying CPU. More specifcially, instead of using a 4-way issue CPU, a 2-way issue CPU has an execution latency of 1.82x. Experiments show that EDDI is able to provide a 98% error coverage capability [11]. Figure 3.2 shows how EDDI adds the shadow instructions and schedules the instructions. As can be seen from this figure, the SI (marked with an apostrophe) are interspersed with main instructions for better ILP and the comparison is done just before the data is written back to the memory.

SWIFT tries to tackle the problems that are not addressed or partially addressed by EDDI and tries to improve the overall performance by rather simple optimizations. One simple optimization is that the I5' instruction in Figure 3 can simply be omitted since it is not critical and the stored value will require


Figure 3.2: EDDI instruction duplication and scheduling example.

additional memory space and an additional rather slow memory instruction to be executed (referred to as EDDI + ECC). Another improvement in SWIFT is in terms of the control flow checking mechanisms. EDDI does not have a direct control flow control but rather implicitly checks the number of MI and SI. This check may cause invalid branches to be taken, invalid store and load instructions to be executed that may disrupt the MI execution or feed these instructions with invalid data [4]. SWIFT overcomes this problem by adding the branch instructions as synchronization points and designating block signatures to the executing block, so that the control flow is validated by comparison of the expected block signature and the executing block signature. The block signatures are stored and updated for each block in GSR and RTS registers. This improvement is referred to as EDDI + ECC + CF (Control Flow Checking). One observation for EDDI + ECC + CF is that, the control flow checking mechanism is required only when the output is stored in memory since the main purpose is to keep the data in memory intact. Therefore, the CF mechanism, signature comparison etc can be safely omitted in blocks that do not have store instructions (referred to

as "SCFOpti"). Another simple observation is that, the comparisons for branch instructions is actually covered by the idea of comparison of block signatures which ensures correct branching, henceforth the branch synchronization can be safely omitted (referred to as "BROpti"). SWIFT does not offer an actual performance improvement over EDDI, however it improves EDDI with control flow checking mechanism and by removing redundancies. Based on the conducted experiments, it is shown that SWIFT has an execution latency of 1.41x compared to the baseline implementation.

Another software only fault tolerance technique is software EDAC (or ECC) technique which is a software implementation of ECC (Error Correcting Code). The implementation aims only to protect the instructions that are placed in the memory hierarchy before execution. Due to the dynamic nature of the data it is not preferred and is argued to be not practical to use software ECC [6]. In this approach, the protection of the software code in memory is done in terms of sweeps in which the ECC walks through each memory block that contains software instructions and checks for errors. When a sweep takes place, it takes the highest execution priority; therefore any other software should be halted during the sweep interval which eventually is a bottleneck in availability and performance. The sweeps cannot be cached since; in each sweep the memory will be read once and checked for errors. Another issue for software ECC is that, the ECC software also runs in memory and itself is also susceptible to transient errors in memory. This requires to execute multiple copies and a cross check between these copies. Although software ECC provides some protection when hardware ECC does not exist and offers recovery options compared to other SFT approaches, it does not offer any protection over data and the memory is left unprotected between the sweeps.

There are other software techniques in the literature having different levels of duplications or controls, such as control flow only detections (using signature comparisons for blocks, execution parity calculations) [34],high level code duplication and result comparison [35], analysis for different levels of duplication and tailoring between instruction and procedure call duplication for energy consumption reductions [36], and process level duplication [37].

Software fault tolerance techniques that depend on simultaneous multithreading (SMT), mostly require a hardware support for queuing the load values (LVQ), register values (RVQ), and branch outcomes [13, 28, 29]. Therefore these schemes should be considered under the umbrella of hybrid techniques as they require additional hardware.

In terms of recovery, SWIF-IT offers hamming code and majority voting based recovery schemes in which multiple copies of the same data is stored and in case of an error, it is recovered using the value that has the majority. SWIFT-R [38] technique is an addition to SWIFT that has recovery capabilities. The following techniques are suggested in SWIFT-R. Triple execution of the same instruction and decide the result by majority voting in case an error is detected (referred to as SWIFT-R). Another extension is using AN-CODE to back up a multiple of the original data (possibly 3) and in case of an error, restoring the data by simply using the AN-CODE coefficient. For instance, a data x is multiplied by 3 and stored as y. In case of an error, if y is divisible by 3 then y is correct and x should be recovered as $x = \frac{y}{3}$, otherwise y is incorrect and should be restored to $y = 3x$. SWIFT-R suggests that, the data containing single bit information inside a 64 bit register should not be affected with the bit flips of the rest of the 63 bits that are irrelevant. Therefore, these data bits can be ignored and masked (data&0x1) in order to decrease the vulnerability of the register data (instead of having 64 bit vulnerability, only a single bit is susceptible to transient faults). However, the applicability of this scheme is limited due to performance overheads. For the masking idea, it can be argued that the register value having a single bit value is hard to be ensured and therefore not easy to apply in most cases.

### 3.2.3 Hybrid Techniques

The hybrid techniques are generally offered as extensions of software only fault tolerance techniques, to overcome the single points of failures or performance bottlenecks that cannot be dealt with software. One of the extensions suggested to improve software techniques is CRAFT, which is an extension over SWIFT technique.

In SWIFT, two problems for memory operations were not addressed and considered to be the limitations of the approach. One problem in SWIFT is that, the store instructions are single points of failure [8], since the fault checking is done in software before the store instruction is issued. Any errors occurring before commit and after the error check cannot be detected and will cause the corrupted data to be stored in memory, which may later feed and corrupt other instructions. In order to resolve this store instruction issue, the comparison is removed and replaced with another store instruction. The second store instruction is fed with a signature that is not an actual store instruction; rather, it is issued to validate the original store instruction. In order to provide validation, the stores do not directly store the data to memory, instead they are queued in a protected hardware queue and the commit for stores is delayed until the second (or shadow) store instruction is executed and the data to be written is confirmed to be valid [8].

Similarly, for load memory operation, in order to duplicate the data loaded from memory and have an exact replica of the loaded data, SWIFT does not replicate the load instruction (for memory mapped IO, two load instructions may return different values); instead, it inserts a move instruction to copy the loaded value. However, there are two intervals where transient failures can be corruptive. The first interval is, after the load instruction and before the move instruction, and the second interval for errors is after the memory address verification and before the actual load instruction. In the first interval, the original and the copy will contain a corrupted value which will not be detected. In the second interval, the data loaded will be issued to a faulty address and will contain invalid data. In both cases, the faulty data may eventually feed a store instruction at some point. In order to prevent these faults, the memory load values are queued in a hardware protected buffer called Load Value Queue (LVQ), where only the main instruction loads the data from memory and the shadow instruction loads the value queued inside LVQ, which prevents from feeding the shadow instructions with corrupted data. The data in LVQ can be safely discarded after the shadow instruction is fed with the buffered data. These additional buffers improve the performance while dealing with the vulnerabilities that cannot be addressed by SWIFT. The performance results compared to the baseline are given as 1.334,

1.376, and 1.314 which are improvements with CSB, LVQ, and CSB + LVQ, respectively [8]. Additionally, the vulnerability factor for silent data corruption (SDC), which are the undetected errors, is decreased from 90.5% to 89.0% when both techniques, CSB and LVQ, are applied.

Redundant multithreading techniques can also be considered as hybrid techniques, due to their hardware requirements. With the idea of multithreading as a way of maximizing on-chip parallelism [39, 40], the idea of using multithreading as a way to improve reliability also emerged in various approaches [41, 30, 29]. Chip Multiprocessors (CMP) and CPU level multithreading enabled researchers to investigate different alternatives [29]. Redundant multithreading can be achieved with two alternative ways, running the leading and trailing threads in the same single CPU core that supports multithreading capabilities (SMT) or running the threads in different CPU cores (CRT) preferrably in adjacent cores in order to reduce the physical distance and communication delays. AR-SMT [30], SRT [41, 28, 29] and SRTR [28] were suggested as alternatives for SMT techniques, in which two threads (main and trailing thread) run the same instructions in parallel threads and the results are compared for detection of transient faults. In all SMT techniques (and even CRTs), queues or similar queueing mechanisms are built to enable communication between main and trailing thread. Specifically, LVQ is used to buffer the load values of the main thread, register value queue (RVQ) is used to buffer the register values of main thread for comparison, branch outcome queue (BOQ) is used to store the branch outcomes for the main thread, and store buffer (StB) is used to verify the values issued by store instructions from the main thread. The queue values are created by main thread and consumed by trailing thread in order to check for faults.

SRT chooses stores as synchronization points, instead of register updates. Secondly, a slack fetch organization handles and tries to keep the slack ( trailing thread follows main thread behind certain number of instructions ) between threads to a pre-defined number and organizes fetches accordingly.

SRTR scheme is actually the extended version of the SRT detection scheme with recovery abilities. In order to provide the recovery, some SRT modules were

reconsidered. In SRT, leading instructions may commit to memory before checking for faults. However, this is not feasible for SRTR since recovery mechanism depends on trailing thread values, and therefore can not recover when the values are committed to memory. Secondly, SRTR compares the leading and trailing instruction values as soon as trailing instruction is complete without being have to wait for leading instruction commit. To make this possible and reduce communication requirements, register values are placed in RVQ from which the trailing thread can make comparisons and detect faults. In order to reduce the bandwidth requirements and pressure on RVQ, a dependency check mechanism called dependence-based checking elision (DBCE) is applied to check the true dependency chains of registers so that only the last instruction in a chain is used and placed in RVQ.

Although AR-SMT uses the multithreading infrastructure, it neither aims to exploit the advantages of a CMP structure nor attacks the problems of running the redundant threads in seperate CPU cores. Slipstream [42] processor idea was the first to take the performance advantages of a CMP and use CMPs for reliability purposes. Later on CRT [29] and CRTR [13] were suggested as alternatives for fault detection and recovery in CMP systems. In Slipstream, the register commits are asynchronous in order to overcome the interprocessor communication delays. The checks are only done at memory commits, and the memory commits are done only after the checks are completed. The memory instructions are not copied as in Slipstream (only a single copy of memory is used) only the register files are duplicated. Also it extends the DBCE idea in SRTR with death and dependence based checking elision (DDBCE), in which the formerly ignored masking instructions are also added to dependency chains. The DDBCE chains a masking instruction only if the source operand of the instruction dies after the instruction. Register deaths ensure that masked faults do not corrupt later computation [13].

## 3.3 Software Profiling For Fault Tolerance

Software Profiling for fault tolerance is the analysis of the application in order to understand the characteristics of the application in terms of fault tolerance. The characteristics of the application or the profiling information can be put to use for better protection of the running application. The profiling info can provide the programmer, which parts of the program are more susceptible to transient faults, what is the approximate possibility of occurrence of some fault, what is the impact of an error to the program output, in what path does the errors propagate. These details combined with the underlying hardware, such as operating system or other system specifics will provide invaluable information in terms of providing the best fault tolerance fit to the reliability requirements of the system with minimal cost. This thesis mainly relies on this fact.

One of the studies in this area uses instruction operand types as predictors for transient fault impacts [9]. The instruction operands are separated into four base groups called F, O, C and A. "F" is an abbreviation used for floating point operand, "O" is for memory offset, "C" is for comparison operand and "A" stands for ALU instruction operand. By this grouping, each register in the instructions are grouped according to their usage. For instance, a register operand is of type O, C, A when the register operand is used in memory offset, comparisons and ALU instructions. Using this grouping, software characteristics is tried to be predicted. A case study, Brake by wire [9] is used to evaluate the power of operand type analysis. The analysis shows that, the brake by software is more likely to be vulnerable to F and O type operands, meaning any errors in these operands cause major defects and corruptive faults. Although the type inference idea provides some insight of software vulnerabilities, it does not give any specific detail about how to improve the software fault tolerance techniques. Moreover, although grouping the instructions may seem like a good predictor, all memory instructions cannot be considered important when some of the memory instructions cause critical errors. In other words, not all data written to memory or read from memory carries critical or invaluable information. Or some instruction operands may contain critical data despite the fact that, most of that type

of operands contains insignificant data. As a hypothetic example, consider the brake by wire case in which "F" and "O" type operands seems to carry more vital information. While this gives a better idea about operands, it may not always be true. For instance, an "A" instruction operand may also carry vital information. Naturally, this particular operand should also be considered for protection when reliability is a concern. However with type inference technique it may conversely be grouped into "A" type operands which are considered to be safe to ignore in terms of reliability.

Another study has a similar approach to ours; EPIC suggests that error propagation and the effect of errors is a requirement that should be taken into account when developing dependable software. They track error propagation paths as well as the impact of the errors. For each input-output couple they suggest that there is a permeability that is defined as the conditional probability of an error occurring on the output given that the input is faulty [10]. They extend the idea for multiple input-output systems punishing the modules with large I/O count. When considering a single input and its impact on single output, the probability of an error showing in this output (or the weight of an input) is calculated as the multiplication of the error permeability of the I/O systems in the backtrack tree. Figure 3.3, is an example that shows I/O paths and weights for outputs in EPIC [10].

For instance, the calculation of W2 is follows:

$$W_2 = P^B_{2.1} P^B_{1.3} P^D_{1.1} P^E_{2.1} \tag{3.9}$$

The error permeability is only one side of the story and it should be combined with the effect of the errors that propagated the criticality of the error. The criticality of the errors is a coefficient for the weights calculated which is decided by the system administrator and is a value between 0 and 1, where 1 denotes the highest possible criticality value.

In EPIC, inputs of the system are assumed to be independent of each other and the dependencies are tried to be resolved with the coefficients assigned. An input may have different effects on the output according to the variation or values of

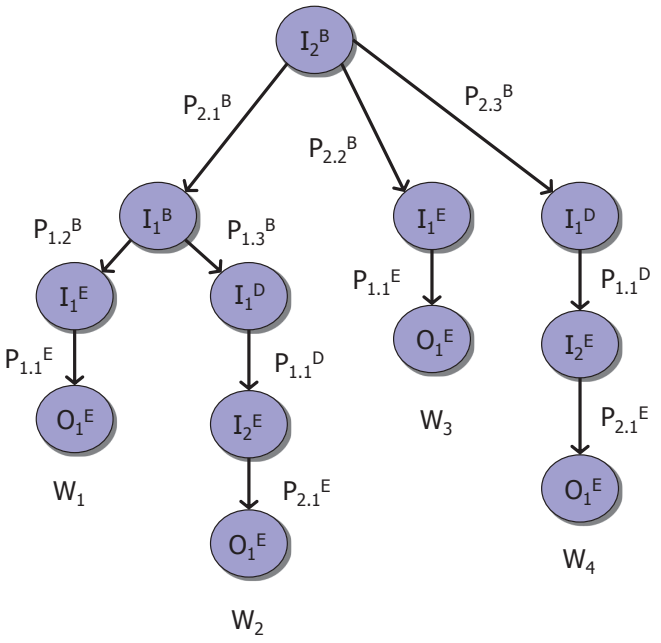


Figure 3.3: Weight calculation for error paths.

other inputs. In addition, error permeability of a single input can be misleading, an input may have critical effect on the output (high permeability), although the probability that it is used in calculation of the output may be low (a branch that is taken only 0.1% of the time). The impact of the risk and the probability of the risk occurrence should be evaluated independently and the system administrator should be able to assign coefficients for these factors. In addition, the errors that seem to disappear in error permeability paths may go unnoticed and eventually have dramatic impacts in the program output. For instance, these errors may cause invalid branches to be taken, input data corruption, and invalid memory accesses.

The testing environment chosen for EPIC is in embedded (or circuitry) systems domain in which, input-output signals and their relations have more clear cut definitions. However, in practice, a single output may not be present for an

input (an input may only be used for data offset, branch decision, etc.) or the output may feed another output as an input.

EPIC uses the gathered profiling information for simple executable assertions (EA), which checks for an output signal to be in a pre-defined range and injects these checks in software where an input has high error permeability on the output. The input-outputs are considered in signal level in EPIC, therefore the data in between input and output is simply ignored.

Another approach similar to ours, although has a very different context and a very different fault model, supports the ideas presented in this thesis. The context of the work is financial calculations in stream processing environments where large amount of data is constantly processed and a continuous output is produced. Partial fault tolerance (PFT) is tested for stream processing applications in order to understand if PFT is viable. Specifically, PFT is partial selection of the data for protection in order to achieve decent reliability with minimal cost and impact on performance. Output quality for the financial calculations is used as a key metric for deciding which parts of the data to protect. Although PFT idea is presented for data protection in stream processing applications, we can borrow and apply the same idea for general reliability concerns. Applying PFT is not reasonable without a clear understanding of the impact of faults on the quality of the application output [43]. Similar to PFT, our SSFT idea also requires an output quality assessment when selectively applying the reliability measures in the running application. Our expectation, similar to PFT, is that SSFT will provide better resource utilization, less power consumption and better performance.

Our design is different from EPIC in following aspects; we use a probabilistic approach for error occurrence calculation, and estimate an expected probability for an error to occur. Additionally, we extend the error impact idea, using again the golden runs idea, and use different measures for error impact calculation. We then combine the probability and error impact values, and with user defined coefficients calculate an expected error rate for different variables inside the software. We do not reduce the inputs and outputs to signals; therefore our approach is more generic and can be applied regardless of the running application or hardware. We then use optimization techniques for best fit protection that has the

minimum protection cost (performance, hardware requirements), while providing a decent amount of protection. The next section will explain the details of the approach taken in this work.

# Chapter 4

# Our Approach

## 4.1 Preliminaries

In this section, we will briefly introduce the definitions required to understand this thesis. Basic block is a fundamental term, which other notions are build on. A basic block is a piece of code that does not contain any branches or jumps. Branches or jumps provide transitions between blocks. Basic blocks have single entry and exit points. The transitions between an entry point of one basic block, to exit point of another basic block, which are called edges. In order to have a deeper understanding of a software execution cycle, the compilers form a data structure to present the pieces of software which is called Control Flow Graph (CFG). To obtain this graph, the software is decomposed into basic blocks that are connected by edges.

GCC, the compiler infrastructure used in this thesis, is a collection of compiler front ends for languages C, C++, Objective-C, Fortran, Java, Ada, and Go etc. GCC converts these high level code into GIMPLE statements as a middle-layer form, which is a three-address representation which is formed in tuples of no more than 3 operands (with some exceptions like function calls) [44]. In order to imitate SEU, we inject errors on the parameters of the GIMPLE statements, which we will refer to as statement parameter throughout the rest of this thesis.

The term program execution cycle referred in this thesis, is a process that begins with execution of application binary and ends with generation of the output.

## 4.2 Overview

Selective Software Fault Tolerance, relies on the fact that the characteristics of the running application and the underlying hardware have major impact on the reliability of the whole system. Profiling information can be put to use for fine tuning reliability measures and effectively protecting the entire program. Our main idea of software profiling for reliability, is rather simple and intuitive. The statements that are executed the most in percentage, and the parameters that are used in these statements are most likely to be vulnerable to transient faults. Additionally, the statements that cause the most dramatic effect in terms of program execution cycle and program output quality are likely to require the most protection when reliability is considered. These ideas when combined constitute an effective approach for analyzing the program vulnerabilities.

The statements that are executed the most in percentage can be analyzed by statistical analysis of executing program. The statistical information about the number of executions of a single line of code can be obtained through compiler directives. When we know the execution statistics for each satement, we can simply acquire the percentage of execution (PE) of a statement in the program execution cycle.

$$\mathbf{PE_i} = \frac{\text{Execution count for statement i}}{\sum_{i=1}^{n} (\text{Execution count for each statement})} \tag{4.1}$$

PE for a statement (or an instruction for assembly language) actually gives the likelihood of a statement being executed in a standard program execution cycle. The more a statement is executed, the more it is likely to be hit by transient faults. With this information, one can extract the statements that are most vulnerable and these statements can be considered as main points of failure when SEUs are of concern. It should be noted that program execution depends on the input data. Therefore, the execution statistics and PE for each statement is likely to change for varying inputs. For simplicity, each test input is given the same probability and the final execution counts are taken as the average of the

execution counts for different inputs. However, this estimation can be tailored using the likelihood of the input data; therefore a finer approximation for PE can be achieved.

The statements that affect the program execution and output quality the most can be analyzed through experimentation methods. For experimentation, the SEU should be imitated for a single statement (even better to analyze a single parameter for a line of code). After a bit flip is injected inside the program, the program should be executed and the behavior of the application should be observed. This observation will involve the quality of the program output, or abnormal execution termination after the injection. The output quality will be measured by comparing the results before and after error injection (which is also called Golden Run Comparison). The quality measure is different for each and every different program, therefore different measures need to be applied. Typical measures we use for our approach and details will be discussed in section 4.4. From these measures, one can infer the quality effect (QE) of a SEU for a given parameter inside a statement. The QE will be measured as a percentage and for simplicity the program hang-ups, memory segmentation errors etc. will be considered as a 100%QE. For applications that do not need to generate output, for each input data, this QE value can be modified accordingly. For instance, for an electronic wheel steering application, an output is generated in every few microseconds. Although, the application is safety critical, it will work uninterrupted, in case of a single failure in output.

When considering QE for a transient fault, there are different observations and aspects that need to be considered. A critical observation here is that, some statements may not require reliability precautionary measures since they may contain irrelevant information, dead code or code only needed for debugging purposes. These statements do not have a QE, therefore may be ignored in terms of reliability. Similarly, some parts of the program may only be for testing purposes or contain test data. Those statements, should also be ignored when QE experimentation takes place. Our technique allows user to selectively inject errors, so that irrelevant functions are ignored.

The output quality is subjective, since different applications produce different kinds of outputs which may or may not tolerate soft errors. This subjectivity requires a human observation for output quality, since quantitative analysis may not be sufficient. For instance, consider a lossless compression algorithm. Any error in the output is intolerable, since the output is useless when any byte errors occur. Conversely, a lossy compression algorithm used in JPEG images or media compressions will be able to tolerate some margin of byte errors, therefore, is more flexible in terms of soft errors. Similarly, applications that contain calculations where imprecise or partially correct results are acceptable are called soft computing applications. Especially, these types of applications, where soft computing is used as a computation convention, are by nature, more resilient to soft errors. Therefore, a margin of error is presumed to be acceptable which makes these applications perfect for Selective SFT optimizations.

In addition to QE and PE parameters, other parameters should also be considered when estimating the impact of transient faults in the program execution. For example, at the hardware level, process technology will have an important effect. According to recent research [2], dimension scaling will cause 8% increase in soft-error rate with each generation. The process technology, therefore, should also be considered as an input when vulnerabilities to transient faults are being measured. While shrinking process technology, has its own advantages in power consumption, heat dissipation, it also makes the entire system more vulnerable to transient faults.

By using the QE and PE parameters, one can simply evaluate the expected rate of error for each statement parameter inside the program. Expected rate of error (ER) can be calculated by the following equation.

$$\mathbf{ER} = F\,(\mathrm{PE}) \times F\,(\mathrm{QE}) \tag{4.2}$$

Expected rate of error is, the error expectation percentage in the program output in case of a transient fault occurrence inside a statement argument. Statement parameter or statement argument term shall be used throughout the rest of the thesis referring to the variables inside a statement to which error injections can be made (i.e. transient faults can alter the value of the parameter causing errors).

The $F(PE)$ and $F(QE)$ are functions that can be used to adjust the error rate factors' impacts. For simpler cases, where the parameters are to be used as is, F function can be taken as a unit function. For instance, in some cases, the percentage of execution should be adjusted in order to increase the significance of this parameter meaning that the error occurrence probability is an important factor for calculating the expected error rate. Consider that the QE effect is required to be empowered for very low error rates and weakened for values larger than 50%, then a model function F(QE) can be constructed as $F(QE) = \left(\frac{QE+25}{QE+100}\right) \times 100$. For a different scenario where QE is required to be empowered for larger QE values, than $F(QE)$ may be constructed as $F(QE) = \left(\frac{QE \times 30}{110-QE}\right)$. When a simpler logic, like diminishing the overall error values is required following expression can be used $F(QE) = m \times \log_n(QE)$. Figure 4.1 shows the effects of applying different functions on the QE values.

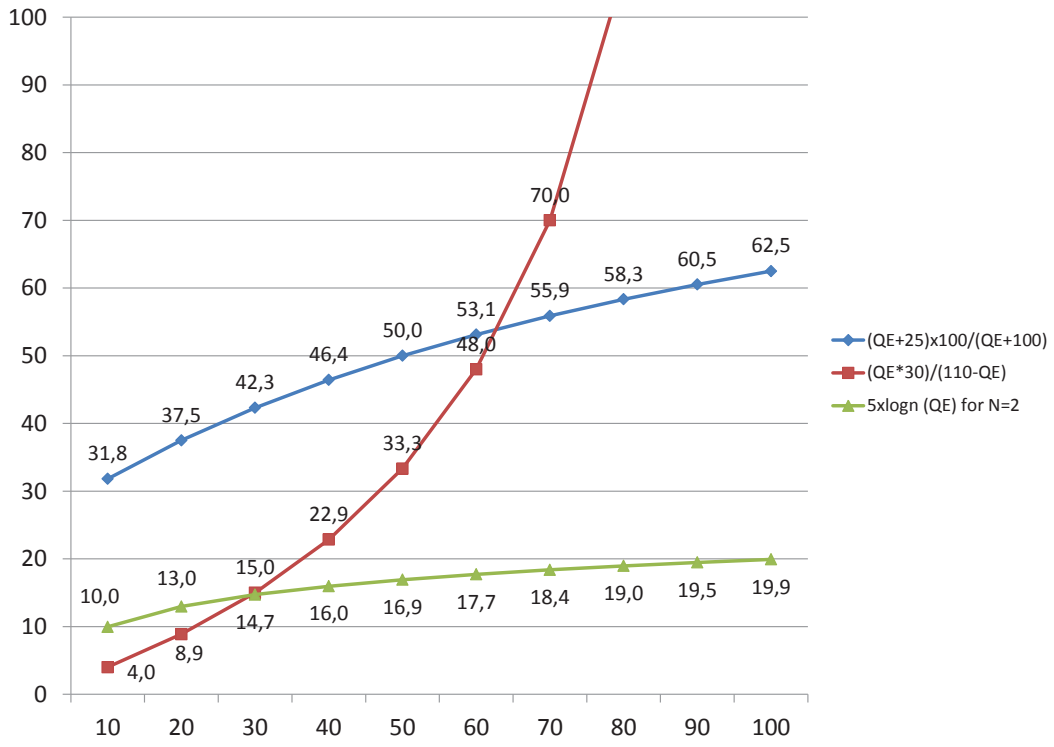


Figure 4.1: Example functions to adjust parameters.

In order to obtain the expected rate of error for the entire program, the ER values for each individual statement argument should be accumulated. Formula 4.3 shows how PER (Program Expected Error Rate) value is calculated.

$$\mathbf{PER} = \text{AER} \times \sum (\text{ER}) \tag{4.3}$$

In formula 4.3, AER is the Architectural Error Rate which is a physical measure of the rate of failures (or the percentage of transient fault event occurrence). The general convention in literature is to use MTBF (Mean time between failures), instead of failure rates since the MTBF value is a positive value and a large value compared to the failure rates. For instance, consider a machine that is expected to fail in every 5000 hours, the MTBF is considered as 5000 hours, which is easier to use than an error rate of 0.0002 failures/hour. Failure rates are generally considered with a factor of time, rather than number of instructions executed. We can calculate AER from a given failure rate as follows. For instance, if a CPU has a rate of 1 failures/second and CPU can execute $10^9$ instructions per second, we achieve an AER value of $10^{-9}$.

This AER parameter can be affected by the density of the transistors used in manufacturing the CPU unit, CPU clock frequency, and environmental radiation. For instance, a CPU that is manufactured using denser transistor technology is more susceptible to transient faults then a CPU that is manufactured using sparser transistor technology, i.e, a 22 nm CPU is more likely to be affected by radiation than a 90 nm manufactured CPU. In addition AER factor can also be affected by environmental radiation. Normand states that, in avionics the SEU rates are correlated to neutron flux, which is also correlated to the altitude [45]. For instance, an application running in a plane flying at 10000 ft is less likely to be hit by neutron radiation than an application running in a plane flying at 60000 ft.

Since, PER is a measure of the rate of error that a program is expected to produce against transient faults; we use PER as a reliability measure for the program and try to optimize our reliability measures while keeping PER parameter in an acceptable range. The acceptable PER value defined for the application, will be referred to as $\lambda_{PER}$ throughout the rest of this thesis. $\lambda_{PER}$ value can be

adjusted for the reliability requirements of different applications. For the applications that do not tolerate any errors, the $\lambda_{PER}$ value is zero meaning that the program is fully protected and is not produce any errors in case of a transient fault.

For our optimizations, our method is to leave out as much statement parameters unprotected as possible, while preserving $\lambda_{PER}$ value of the application. The statement parameters that are protected requires, some sort of reliability measures to be taken like instruction duplication, parameter duplication and similar which result in performance overhead, increased hardware requirements, and more power consumption. When we reduce the number of statement parameters that are protected, we end up with a system that has better resource utilization, less power consumption and better performance. In order to choose the statement parameters that have less critical impacts on PER, we can sort the statement parameters in terms of their ER values. For instance, a statement parameter with an ER value of 0 can safely be omitted without affecting the PER value of the running application, meaning a transient fault that changes the value of this statement parameter does not have any impact in the produced software output.

Our approach works as follows; we first accept each and every statement parameter to be protected as default by some form of reliability measure (redundancy, memory protection, etc.), therefore the PER value is zero at the beginning. The statement parameters are sorted in ascending order according to ER values. Then, we remove protection from each statement parameter one by one. PER value is re-calculated according to the ER values of the new set of parameters. When we reach $\lambda_{PER}$ value, we can no longer optimize since the PER value will not be acceptable according to reliability requirements of the application. When $\lambda_{PER}$ value is zero, we can only take out the statement parameters that have zero ER values. These parameters do not require any protection and can be omitted when reliability measures are taken.

We call this optimization technique selective SFT (SSFT), meaning selectively and carefully choosing the parameters that require Software Fault Tolerance and

leaving the rest of the parameters unprotected. By the use of aforementioned optimization, the underlying hardware and the specific output quality requirements of each application is taken into account. This way, the amount of redundancy is minimized even though some statement parameters inside the program are left unprotected. This optimization provides a reliability level tailored for each application, which in turn, increases the performance, decreases the hardware requirements (therefore reduces cost), decreases power consumption, and reduces the overall redundancies introduced to the system (therefore reduces the vulnerabilities).

As discussed in chapter 2, our technique can be used in conjunction with any redundancy technique. It can be used to reduce the number of duplications in, those techniques that use instruction duplication such as EDDI or SWIFT. Similarly, it can be used for software and even hardware ECC techniques, to selectively choose the statement parameters to protect. For multithreading techniques such as CRTR or SMT, our technique can be applied for safely releasing the resources seized by the trailing thread, reducing the slack between main and trailing threads, therefore improving the overall performance while reducing power consumption and resource requirements.

As expected, our technique will provide better optimization when $\lambda_{PER}$ value is above zero, meaning that a margin of error is acceptable for the application. The next section will discuss the details of our system architecture.

## 4.3 Selective SFT

Selective SFT (SSFT) technique uses static and dynamic analysis for profiling the application. As explained in section 4.1, compilers form Control Flow Graph (CFG) to understand application characteristics better. CFG is a static form of analysis, which contains all possible paths that an application can traverse during execution. In addition to static information captured in the CFG, statistical data recorded during program execution will also give a better insight about the application. Statistical data is recorded during program execution and is a dynamic analysis instrument. As such, SSFT records the damage in output (which is also another dynamic analysis instrument) for a statement parameter in case of a transient fault occurrence. Figure 4.2 shows a partial CFG for an

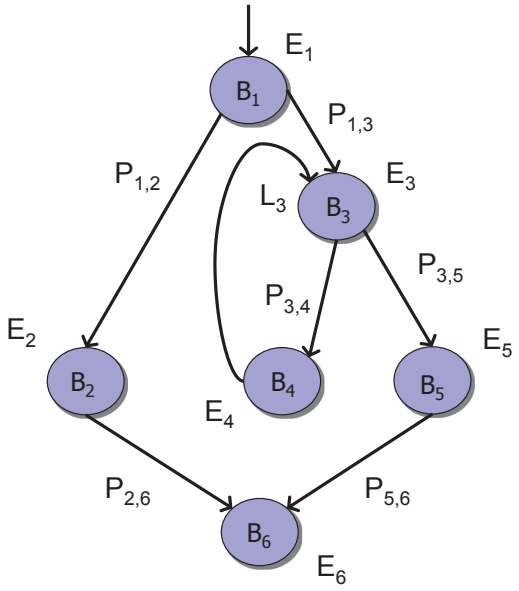


Figure 4.2: An example CFG graph.

application. Let the probability of branch (shown as directed edges) that connects

basic block $B_i$ to $B_j$ be $P_{i,j}$. For instance, the probability of the branch that connects $B_1$ to $B_2$ is shown as $P_{1,2}$ in Figure 4.2. Also, let the set of predecessors for a basic block be $PRE_i$. For instance, the set predecessors for basic block $B_6$ (or $PRE_6$) contains two basic blocks, namely $B_2$ and $B_5$. The cyclic edges in the CFG shown in Figure 4.2, shows the loops in the application. Let $L_i$ be the loop count for basic block $B_i$, i.e, for example $B_3$, will have a loop count equal to $L_3$. $B_3$ will be executed $L_3 + 1$ times, since loop control block is executed once more before the loop exits. Finally, let each basic block $B_i$ be executed $E_i$ times. The entry block $B_i$ in Figure 4.2, will be executed once. Equation 4.4 shows how execution time of basic block $B_j$ is calculated.

$$E_j = \sum_{B_i \in PRE_j} (E_i \times P_{i,j}) \times (L_j + 1) \tag{4.4}$$

For example, for basic block $B_3$, $B_1$ is the only predecessor in $PRE_3$, therefore $E_3 = E_1 \times P_{1.3} \times (L_3 + 1)$. The value $L_3$ is different than 0 since, $B_3$ has an incoming cyclic edge meaning that $B_3$ is loop control block. For basic block $B_6$, $PRE_6$ contains the basic blocks $B_2$ and $B_5$, and $L_6$ is zero since the basic block does not have an incoming cyclic edge. The execution count of $B_6$ is, $E_6 = (E_2 \times P_{2.6}) + (E_5 \times P_{5.6})$. The branch probabilities $P_{2.6}$ and $P_{5.6}$ is 1 since these branches are unconditional, hence $E_6 = E_2 + E_5$. We use these $E_i$ values for calculating the vulnerability of the statements inside the basic blocks. The statements that have high percentage of execution in the program execution cycle, are likely to be affected by transient faults. Equation 4.5 shows how percentage of execution for a statement in a basic block can be calculated.

$$PE_i = \frac{E_i}{\sum_{i=1}^{n} E_i} \tag{4.5}$$

Let parameter k of statement j in $B_i$ is exposed to SEU be expressed as $P_{i,j,k}$. Let the output produced by the application, when parameter k is affected by a transient fault be $O_{i,j,k}$. Let the output produced by the application without any errors be $O_{golden}$. We use a "COMPARE" function to evaluate the quality of output $O_{i,j,k}$, according to the requirements of the application. The result of the evaluation is stored in parameter $QE_{i,j,k}$. Equation 4.6 shows how the damage caused by SEU in $P_{i,j,k}$ is estimated.

$$QE_{i,j,k} = \text{COMPARE}(O_{golden}, O_{i,j,k}) \tag{4.6}$$

Let the probability of SEU for an application be $P^{SEU}$ and probability of statement i to have SEU be $P_i^{SEU}$. Equation 4.7 shows the probability calculation for $P_i^{SEU}$.

$$P_i^{SEU} = PE_i \times P^{SEU} \tag{4.7}$$

The error expectation for parameter k of statement j in basic block $B_i$, depends on the probability of SEU occurrence for statement j and the damage caused by statement parameter k when SEU occurs. Note that, error expectancy also contains the probability of SEU for the application. The estimation of error expectation will then be:

$$ER_{i,j,k} = P_i^{SEU} \times QE_{i,j,k}. \tag{4.8}$$

The error expectancy of an application (or PER value in our previous expression) is an accumulation of the error expectancies of the statement parameters. SSFT orders the parameters according to $ER_{i,j,k}$ values and removes as much parameters as possible, without violating the error margin of the application. The reliability of the application is not impaired while improving the performance, reducing the hardware requirements and power consumption.

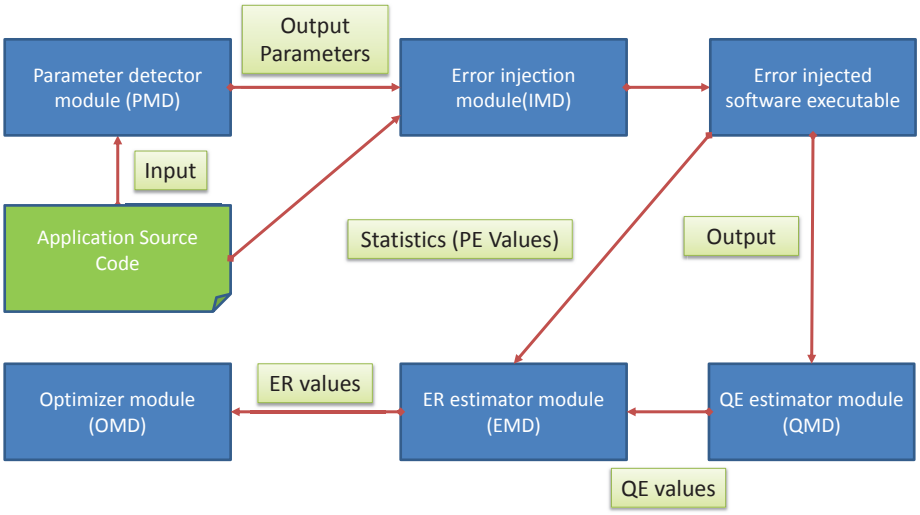


Figure 4.3: SSFT system architecture.

## 4.4 Implementation Details

Our overall system architecture is shown in Figure 4.3. As can be seen from Figure 4.3, "Parameter Detection Module" (PMD) takes "Application Source Code" as input, and detects all the integer and floating point parameters inside the source code to which error injections can be made. "Error Injection Module" (EMD) takes the statement parameters and the "Application Source Code" as input. EMD then, injects an error to a certain parameter in a statement and compiles the application code to produce "Error Injected Application Binary". The "Error Injected Application Binary" then, is executed to produce output and statistics data. "QE Estimator Module" (QMD) uses the output produced by "Error Injected Application Binary" to calculate the QE value for the error injected statement parameter. EMD takes QE and PE values as input and estimates the ER values. When all the statement parameters detected by our PMD are injected with errors and ER values are calculated, "Optimizer Module" (OMD) can sort the statement parameters according to their ER values, and

optimizes until the user defined PER value is reached.

In order to estimate ER (expected rate of error) values for statement parameters of the running application, we have to perform error injections in statement parameters. Existing error injection tools like NFTAPE, GOOFI, PROPANE and SWIF-IT was not useful for our purpose, since they are either architecture specific or has limited error injection capabilities. Additionally, some of the studies similar to ours use "GNU Debugger" tool, which has error injection capabilities on the running application. The problem with this tool is that it does not offer any automatic capabilities whereas our approach require thousands of different errors to be injected automatically, in addition to compiling the error injected code and recording the output. Moreover, our experiments require a tool that has high level language error injection capabilities, can discriminate and record different types of error injections and is flexible. For error injection, we preferred a high level approach since it will result in fewer test runs and will be sufficient for our purposes. A detailed low-level error injection tool would have injected errors in different register values or memory parameters which eventually will impact the value of a single parameter inside a high level statement.

In order to optimize the statement parameters, we applied two algorithms. Algorithm 1 shows the steps of ER estimation for each statement parameter. Algorithm 2 uses the data obtained from Algorithm 1, to selectively pick the statement parameters that can be optimized (i.e. can be omitted with a fault tolerance approach).

**Algorithm 1** Error Expectancy Estimator(ER-set).

**Input:** P-set, set of statement parameters
**Output:** ER-set, set of ER values of statement parameters

**procedure** PARAMETER-SCANNER(P-set, I-set, F-set)
  I-set$\leftarrow \emptyset$
  F-set$\leftarrow \emptyset$
  **for all** $p \in$ P-set **do**
    **if** $p.type = integer$ **then**
      I-set $\leftarrow$ I-set $+ p$
    **else if** $p.type = float$ **then**
      F-set $\leftarrow$ F-set $+ p$
    **end if**
  **end for**
**end procedure**

**procedure** ESTIMATE-PE(execution_count)
  return $\frac{\text{execution\_count}_p}{\sum_{i=1}^{n} \text{execution\_count}_i}$
**end procedure**

**procedure** ESTIMATE-ER(statistics$_p$)
  return $ER_p \leftarrow$ F(QE$_p$) $\times$ F(PE$_p$)
**end procedure**

**procedure** ER-ESTIMATOR(I-set, F-set)
  **for all** $p \in$ I-set $\cup$ F-set **do**
    p.value $\leftarrow$ p.value XOR $(1 \ll \text{Rand}(32))$
    executable$_p$ $\leftarrow$Compile(P-set)
    output$_p$ $\leftarrow$Execute(executable$_p$)
    execution_count$\leftarrow$Statistics(executable$_p$)
    QE$_p$ $\leftarrow$Error-Compare(output$_p$, output$_{golden}$)
    PE$_p$ $\leftarrow$Estimate-PE(execution_count)
    ER$_p$ $\leftarrow$Estimate-ER(QE$_p$, PE$_p$)
    ER-set$\leftarrow$ ER-set+ER$_p$
  **end for**
**end procedure**

**Algorithm 2** Optimizer.

**Input:** ER-set, set of ER values for statement parameters
**Output:** R-set set of parameters that can be removed, PER value
**Require:** Initial PER value is 0.

```
procedure OPTIMIZER(ER-set, acceptable_PER)
    SortAscending(ER-Set)
    R-set← ∅
    for all p ∈ ER-set do
        calculated_PER ← Calculate_PER(R-set + p)
        if calculated_PER ≤ acceptable_PER then
            R-set ← R-set + p
        else
            PER ← Calculate_PER(R-set)
        end if
    end for
end procedure
```

For our error injection purposes, we use the GCC compiler infrastructure. GCC uses different passes in the compilation process which include parsing the code, converting high level code to GIMPLE statements, compiler-level optimizations, code elimination and many other passes. GIMPLE is a three-address representation which is formed in tuples of no more than 3 operands (with some exceptions like function calls) [44]. Our error injection is implemented as a GCC compilation pass, and is capable of detecting the parameters inside GIMPLE statements, and injecting errors on these parameters in the form of SEU (Bit flips on data). A single high level language statement may correspond to multiple statements in GIMPLE since auxiliary variables are needed when breaking the code into 3 operand form. Figure 4.4 shows the conversion of a statement into GIMPLE.

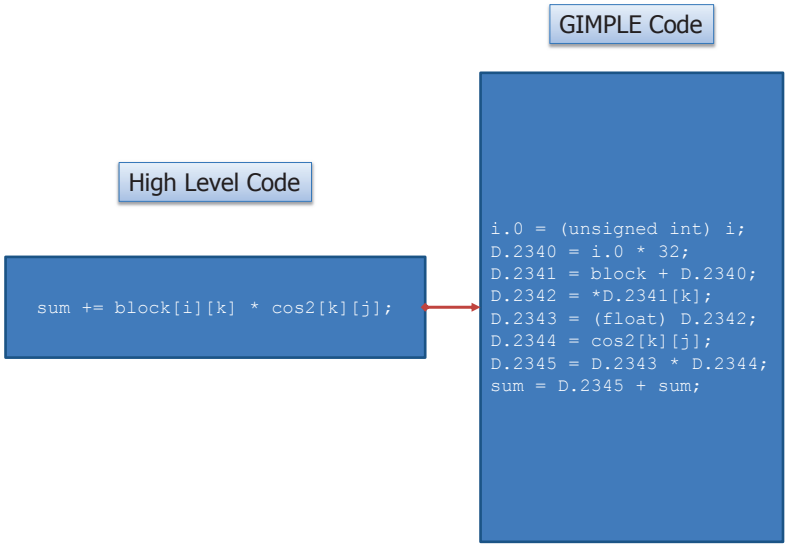


Figure 4.4: GIMPLE code example.

Our error injection tool inside GCC has three major capabilities. First, we are able to record the integer and floating point number variable parameters for each function implemented in the running application. This will provide the base data for our error injection scripts, which will simply inject the error to a specific parameter inside a specific function, compile and run the software. The detection and recording of the statement parameters is presented as "Parameter Detector Module" in Figure 4.3.

In order to use these error injection parameters and compile and run the error injected software, we use scripts and modify the GIMPLE statements. Specifically, we notify the compiler to inject a single bit flip error into a specific operand of a GIMPLE statement during compilation. After the compilation is done, the error injected application executable is built and the results of error injection is recorded for analysis. Figure 4.5 shows the compilation process for our implementation.

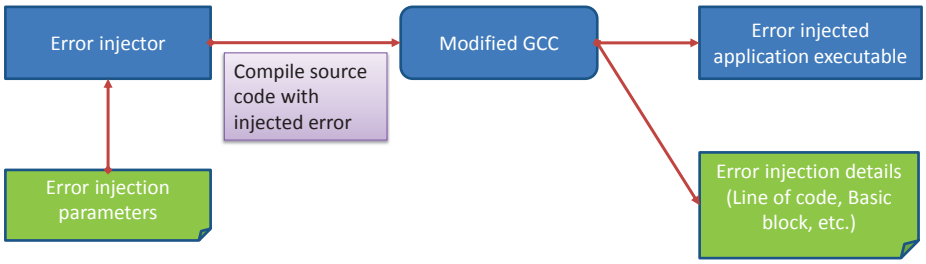


Figure 4.5: Modified GCC compiles the application source code to produce error injected application executable.

After the compilation of modified executable, we run the executable and record the output with a specific error signature (the name of the function and the index of the integer or float parameter can be used for such purpose). The output recording requires some sort of book keeping mechanism so that the outputs can be easily traced back to error injections. For instance, for a program that generates a file as an output, the output name may comply the error injection details so that the output can be associated to the error injection. Similarly, for a program that generates a result as an output, the program output should be written to a text file including the error injection details. This recording, in some cases, requires modification to the application source code which is one of the obstacles in our implementation. Figure 4.6 shows how we implement the overall test procedure.

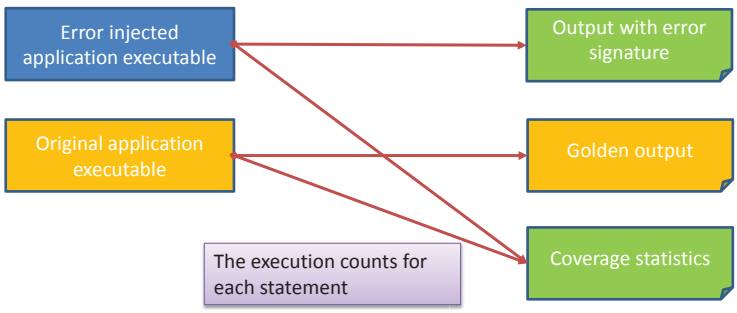


Figure 4.6: Test runs produce output and coverage statistics.

For estimating the error rate, golden run comparisons are performed. Application is first executed without any error injections which is called the "Golden Run". Then, the error injected copies of the same application is executed. The output generated by the "Golden Run" is used as a yardstick to measure how the error injected software output strays from the "Golden" output value. In our implementation, we compare the results of the golden output and the error injected outputs, and record the results in a file. For error percentage calculations, different criteria is used for each program. These criteria may actually be subjective and require user provided coefficients in order for the error percentages to be more meaningful. For instance, a JPEG conversion or filtering application may be more tolerable to errors in the output; however a lossless compression application, like ZIP compression, can be intolerable to any sorts of error. Similarly, a stray in the PI calculation application may have a more dramatic effect in the output, while a statistical analysis tool can be more tolerable to errors in the output.

For simplicity, we have chosen criteria for error percentage calculations as

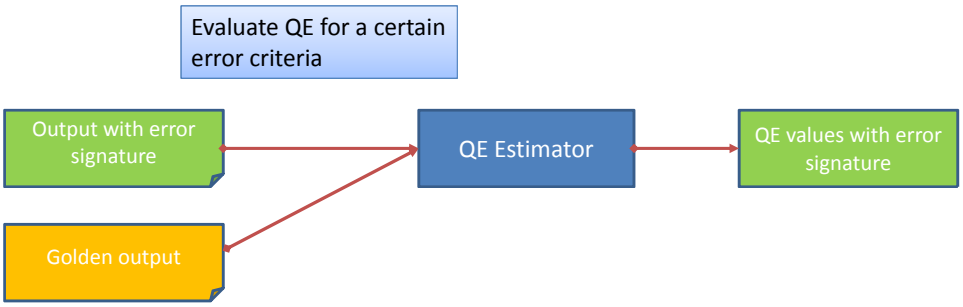


Figure 4.7: QE is estimated by golden run comparisons.

simple as possible. For applications that generate files as outputs, we have used the error criteria as the amount of data that does not match the Golden output. For applications which generate ranking information as output, like in the case of a sorting output, we use the ratio of the number of out of order elements to the number of all elements in the list. In case of single result output applications, like mean value, standard deviation, or square root calculation, we measured the error rates by simply measuring the percentage of stray in final results. The error rates are captured and stored in the form of percentages and this information is used for optimizations and expected error rate calculations along with other statistical information. Figure 4.7 shows how error rate calculations are handled.

As explained before, our technique adds a statistics parameter during compilation in order to record the execution statistics data. The execution statistic we collect is called Coverage Statistics, which is generated by "GCOV" tool provided by GCC infrastructure. When the compiled software is executed, GCOV interferes and records program execution statistics like the percentage of branch decisions, the number of executions for each line of code, etc. The recorded data can then be visualized when necessary. Since our primary interest is in the vulnerabilities against transient errors for each statement, we use GCOV to generate the source code containing the number of executions of statement. These execution counts provide valuable information when identifying the sensitivity of each statement. The reliability measures for these code pieces should be stricter for keeping reliability at acceptable levels. Figure 4.8 shows how this data is generated. These results can be processed and used for our software profiling and optimization purposes for better tailored software reliability.

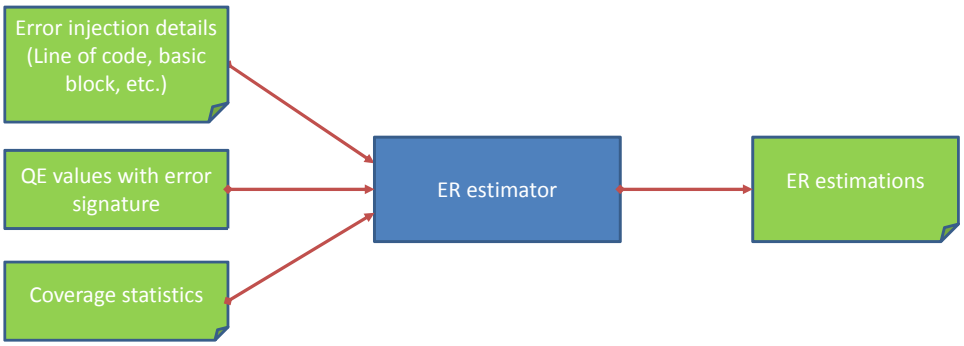


Figure 4.8: ER estimations are produced using QE values, coverage statistics (PE values) and error injection details.

The collected results (ER estimations) can be used for different optimizations, i.e. same data can be processed differently according to different PER values. As discussed in chapter 4.2, $\lambda_{PER}$ value is the acceptable PER value for an application. When considering an application that cannot tolerate any errors, that is $\lambda_{PER}$ value is 0, the optimizations can only take place on the statement parameters that do not cause any forms of errors in the program output in order to yield a 0 PER value. This allows us to provide different levels of optimizations for different reliability requirements.

We modified the GCC compiler and added our error injection pass into the GCC compiler framework in order to implement error injection in the system. This pass has been implemented using C programming language, where we instruct the GCC compiler to inject the errors. We have chosen benchmarks that are coded in C programming language with minimum number of external libraries. We combine the data generated from different sources (our own, GCC, GCOV, etc.) using JAVA programming language.

Due to limited capabilities in GCC, error injections for pointer or array data are missing from our implementation. Section 4.5 will discuss over an example how we used the control flow graph and statistical information in order to obtain the PE parameter.

## 4.5 CFG Based Vulnerability Estimation Example

As explained in section 4.1, CFG represents the execution properties of the application in basic blocks and edges. For example, Figure 4.9 shows the CFG of the DCT function from the Compress benchmark.

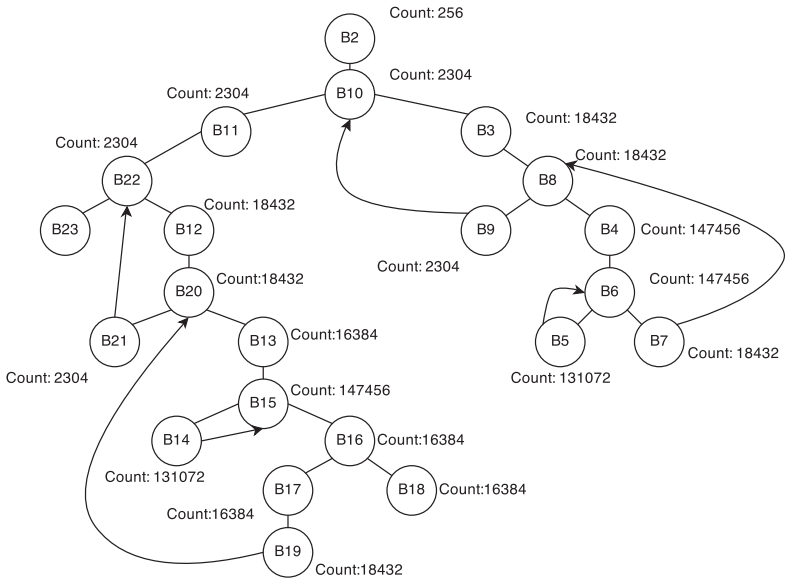


Figure 4.9: CFG for DCT function of Compress benchmark.

The execution statistics in Figure 4.9 were obtained using GCOV, whereas CFG itself is obtained using GCC compiler. During the execution of the benchmark, DCT function is called 256 times, which is the reason for the entry basic block, B2, to be executed 256 times. Therefore, each instruction in B2 is also executed 256 times.

Figure 4.10 shows the code fragment corresponding to the basic blocks B2 through B10 of Figure 4.9. Note that, these basic blocks are on the right branch of the given CFG.

The execution statistics in the CFG are in accordance with branch probabilities. For instance, basic block B10, which contains the necessary GIMPLE code

```
1   int i;
2   int j;
3   int k;
4   float sum;
5
6   for (i = 0; i < B; i++) {
7     for (j = 0; j < B; j++) {
8       sum = 0.0;
9       for (k = 0; k < B; k++) {
10        sum += block[i][k] * cos2[k][j];
11      }
12      temp2d[i][j] = sum;
13    }
14  }....
```

Figure 4.10: Code fragment for DCT.

for the first "FOR" loop (for loop with index i), is executed 2304 times. The GIMPLE code for this loop is shown in Figure 4.11.

```
<bb 10>:
  if (i <= 7)
    goto <bb 3>;
  else
    goto <bb 11>;
```

Figure 4.11: GIMPLE Code for basic block 10.

In this example, "B" is a symbolic name used for constant value 8. Basic block B10 is executed 9 times making the total execution count $256 \times 9 = 2304$, which is compatible with the expected statistics. Similarly basic blocks B3 and B8 are executed $2304 \times \frac{8}{9} \times 9 = 18432$ times which is the execution count recorded by GCOV. Additionally, in GCOV, the branch probabilities are recorded as %89 for B3 and %11 for B11, respectively.

```
<bb 2>:
  i = 0;
  goto <bb 10>;
```

Figure 4.12: GIMPLE Code for basic block 2.

Our approach first compiles the "Compress" benchmark and injects the errors for each statement parameter of each function in "Compress". GCC generated

GIMPLE code for basic block B2 is shown in Figure 4.12. In order to inject an error to basic block B2, we instruct compiler with the necessary parameters. We explicitly indicate whether floating point parameters will also have error injected to them. After the error injection, the basic block takes the form shown in Figure 4.13

```
<bb 2>:
  i = 0;
  i = i | 0x00001000;
  goto <bb 10>;
```

Figure 4.13: GIMPLE Code for Basic Block 2 after error injection.

The injection in Figure 4.13 causes the loop variable "i" to have value 8 instead of 0. When the loop variable "i" takes value 8 instead of 0, the loop is not executed therefore no calculations are made and the sum variable now contains an undefined value. According to our measurements, this error injection causes 100% error in the output for the "Compress" benchmark (i.e., this parameter has a QE value of 100%). This indicates that, the value of this variable is extremely important and the software does not tolerate any errors in this statement parameter hence this parameter should be protected using some form of fault tolerance. However, PE (the percentage of execution) is also an important factor and should also be considered when ER value for this statement parameter is calculated. According to our measurements, the total number of statements executed in the entire "Compress" benchmark is "727360". Therefore, $PE_{B10} = \frac{2304}{727360} = 0.3\%$ meaning that, this particular parameter takes 0.316% percent of the program execution cycle.

Similarly, $ER = 100 \times 0.00316 = 0,316$, assuming that adjusting functions for PE and QE are both unit functions. The ER parameter is used in calculating PER (Program Expected Error Rate) and eventually is used in our optimizations. For optimizations, the parameters that have lower ER values can easily be ignored when fault tolerance techniques are applied.

As another example, the basic block B6 has an execution count of 147456. When we instruct the compiler to inject an error in this basic block, basic block

takes the form in Figure 4.15 instead of the one in Figure 4.14.

```
<bb 6>:
  if (k <= 7)
    goto <bb 5>;
  else
    goto <bb 7>;
```

Figure 4.14: GIMPLE Code for basic block 6.

```
<bb 6>:
  k = k | 0x00001000;
  if (k <= 7)
    goto <bb 5>;
  else
    goto <bb 7>;
```

Figure 4.15: GIMPLE Code for Basic Block 6 after error injection.

According to our measurements, this injected error, causes an error rate of 78.68% in the output. Based on this error rate, we can assume that an injection to this statement parameter causes heavy damage in the output, hence should be handled with care when using a fault tolerance technique to increase reliability. The corresponding PE will be equal to $\frac{147456}{727360} = 20\%$, meaning that execution of this inner loop block, corresponds to 20% percent of the execution cycle. When calculated without any adjustment ER will have $78.68 \times 0.2027 = 15.95$ error expectancy. This indicates that this statement parameter is far more important than other statement parameters when reliability is considered, since it constitutes a larger portion of the software execution cycle. However, 100% percent error or a crash in program is undesirable; therefore the error rate needs to be improved by adjusting the functions accordingly.

PER value is calculated as the sum of all error expectancies from all statement parameters. Table 4.16 shows a portion of the error rates, execution counts, and the basic block that statement parameter belongs to, sorted according to the ER values.

| | Injected Statement | Block | Error Rate | Execution Count | Total Execution | ER Value |
|---|---|---|---|---|---|---|
| **1** | 14,-7,dct | 15 | 0 | 147456 | 727360 | 0 |
| **2** | 16,-5,dct | 16 | 0 | 16384 | 727360 | 0 |
| **3** | 23,2,dct | 16 | 0 | 16384 | 727360 | 0 |
| **4** | 6,-15,dct | 7 | 0 | 18432 | 727360 | 0 |
| **5** | 7,-14,dct | 8 | 0 | 18432 | 727360 | 0 |
| **6** | 9,-12,dct | 10 | 0 | 2304 | 727360 | 0 |
| **7** | 20,-1,dct | 22 | 81,00775 | 2304 | 727360 | 0,25660176 |
| **8** | 19,-2,dct | 21 | 83,33333 | 2304 | 727360 | 0,263968313 |
| **9** | 10,-11,dct | 11 | 85,27132 | 2304 | 727360 | 0,270107129 |
| **10** | 8,-13,dct | 9 | 86,43411 | 2304 | 727360 | 0,273790406 |
| **11** | 0,-21,dct | 2 | 100 | 2304 | 727360 | 0,316761989 |
| **12** | 24,3,dct | 17 | 83,72093 | 16384 | 727360 | 1,88583881 |
| **13** | 15,-6,dct | 16 | 85,27132 | 16384 | 727360 | 1,920761806 |
| **14** | 21,0,dct | 4 | 88,37209 | 16384 | 727360 | 1,990607571 |

Figure 4.16: Table for Error Rates and ER values for statement parameters.

From the table 4.16, we selectively apply SFT and obtain a new PER value. As a starting point, we assume that all of the statement parameters are protected through some SFT and the PER value is initially equal to 0 for the application. Then from the set of parameters (sorted similar to table 4.16), we choose the parameters that are to be left out and not protected by the SFT. For instance, if we do not protect parameters with an ER value of 0 (lines 2 to 7), the PER value for the application does not change since these parameters are proven to be not causing any errors in the application in case of a SEU event. The optimization of the rest of the parameters is infeasible if the user set the $\lambda_{PER}$ value as 0.

However, for soft computing applications and applications that are more tolerable to errors, the rest of the parameters can be added to the optimization list until the specified PER value is reached. For example, the parameter in basic block 22 (line 7) has an ER equal to $81.0075 \times \frac{2304}{727360} = 0.256$. When this parameter is considered alone, overall application PER will be equal to $ER \times AER = 0.256 \times 10^{-7} = 2.56 \times 10^{-9}$. If the acceptable value for PER is too low (for instance, $10^{-9}$ could be picked as an acceptable error expectancy

for the program) then this parameter cannot be removed from the list of parameters that will exercise SFT techniques since, the PER value exceeds the acceptable PER value when this parameter is left unprotected. However, if the application is more tolerant to faults, a larger PER value can be used (for instance,$10^{-8}$ for the previous example). This way, other statement parameters can be added to the list of unprotected parameters. As can be seen in table 4.16, parameter in line 8 has an ER value equal to $83.33 \times \frac{2304}{727360} \times 10^{-7} = 2.64 \times 10^{-9}$, whereas parameter in line 9 has an ER value of $85.27 \times \frac{2304}{727360} \times 10^{-7} = 2.7 \times 10^{-9}$. When we do not protect these two parameters, the PER value will be equal to $5.2 \times 10^{-9}$ and $7.9 \times 10^{-9}$, respectively. Therefore, both parameters can be included in the unprotected parameters.

However, when parameter in line 10 is considered, corresponding ER and PER values are equal to $86.43 \times \frac{2304}{727360} \times 10^{-7} = 3.16 \times 10^{-9}$ and $11.06 \times 10^{-9}$, respectively. This PER does not allow leaving this parameter unprotected. Similar to this example, we selectively add as many parameters as possible to the list of unprotected parameters according to $\lambda_{PER}$ value.

The above values are for illustration purposes and are not used for our final results, since it is only applied to a single function in the "compress" benchmark to better show our approach. Chapter 5 will continue with experimental evaluation.

# Chapter 5

# EXPERIMENTAL EVALUATION

## 5.1 Benchmarks and Setup

We implemented our selective software fault tolerance technique in a modified version of GCC, where we introduce error injection capabilities. We used ten benchmarks to test the effectiveness of our scheme. Table gives these benchmarks and their salient features. Note that, these are all implemented in C since we are operating on a modified GCC environment.

| Benchmark | Source | Number of Basic Blocks | Code Size (Bytes) | Number of Execution Cycles |
|---|---|---|---|---|
| bs | Generic | 23 | 4580 | 32 |
| compress | UTDSP | 70 | 7355 | 727360 |
| edge_detect | UTDSP | 49 | 8365 | 1466059 |
| fft_1024 | UTDSP | 24 | 5151 | 85042 |
| jpeg | UTDSP | 940 | 93460 | 15629 |
| lpc | UTDSP | 116 | 13342 | 19155 |
| mpeg2enc | MediaBench | 2646 | 220086 | 28149452 |
| qsort | Generic | 51 | 4926 | 471 |
| sqrt | Generic | 28 | 4149 | 63 |
| st | Generic | 54 | 4284 | 30051 |

Table 5.1: The characteristics of the benchmark codes used in this study.

The third column gives the number of basic blocks in the CFGs of each benchmark. The fourth column gives the maximum memory space occupied by the basic blocks of the applications (at runtime). Note that, if no memory space optimization is performed, this is the "memory occupancy" (or "memory consumption") of the application. Our objective is to reduce the memory occupancy over the course of execution by exploiting the lifetimes of basic blocks. The last column of Table 5.1 gives the number of execution cycles for each benchmark, when no reliability optimization is used.

We have mainly used UTDSP [46] as our benchmark suite since it fits well with our approach. Scientific benchmarks we used from UTDSP are compress, edge detect, FFT, LPC and JPEG. Compress benchmark uses discrete cosine transform to compress a 128 x 128 pixel image by a factor of 4:1 while preserving its information content. Edge detect benchmark, detects the edges in a 256 gray-level 128 x 128 pixel image relying on a 2D-convolution routine to convolve the image with kernels (sobel operators) that expose horizontal and vertical edge information. FFT is 1024-point complex fast fourier transformation benchmark (radix-2, inplace, decimation-in-time). LPC benchmark is a linear predictive coding (LPC) encoder benchmark. JPEG is a JPEG filtering program, which filters images. In addition to these UTDSP benchmarks, we have also added some commonly used applications like binary search, quick sort, square root calculation, and statistics. Statistics application includes sum, mean, variance, and standard deviation calculations. Additionally, we used MPEG-2 encoding benchmark (from MediaBench [47]), which does conversion of uncompressed video frames into MPEG-1 and MPEG-2 video coded bit stream sequences.

## 5.2 Results

Our experiments and optimizations are affected by ER values for individual statement parameters. If a parameter in a statement, does not cause any errors in the output (ER value is 0), then the instruction can be eliminated since it does not require any safety precautions against transient errors.

Aside from the ER values, for individual statement parameters, our experiments and improvements are affected by architectural factors. In order to represent architectural factors, we defined the architectural error rate parameter (AER). AER parameter is directly related to how the CPU is manufactured (higher density transistors will end up with higher transient faults) and the environmental factors (the outside radiation levels, clock rates, how close the circuits are designed to one another). The AER factor is expected to grow exponentially as the nanometric scaling continues [1]. For our base implementation, we take AER factor value as $10^{-7}$. AER factor does not affect our implementation results, since only PER value is affected by this parameter. The acceptable PER value can be adjusted accordingly for different AER values, hence we see similar results with various AER values.

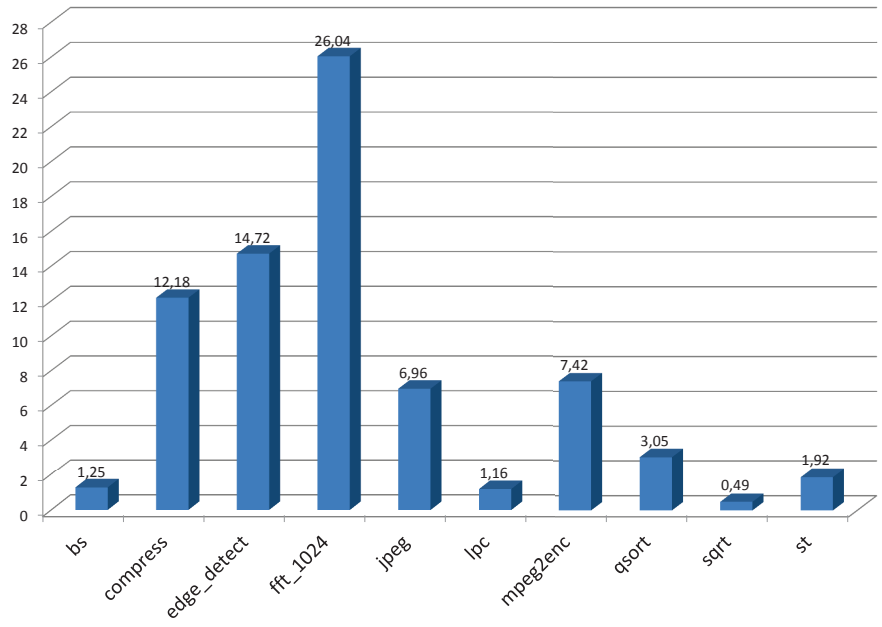


Figure 5.1: PER values for our benchmarks ($10^{-6}$).

As explained before, program expected error rate (PER) depends on ER values and AER parameter. More specifically, PER value indicates the error expectation in the output of an application without applying any protection. For example, fft_1024 benchmark has an error expectation of $2.6 \times 10^{-5}$, which is obtained by using the transient fault and the quality impact of the error when occurred. If there are no errors, that is the application is protected, PER value will be equal to 0. Figure 5.1 shows the PER values for our benchmarks.

As can be seen from this figure, applications have a wide spectrum of PER values ranging from $4.9 \times 10^{-7}$ (sqrt) to $2.6 \times 10^{-5}$ (fft_1024). This wide range of values are mainly due to the fact that PE values for each statement parameter have a different value since they are calculated using execution counts. Specific execution counts for our benchmarks are shown in Figure 5.2.

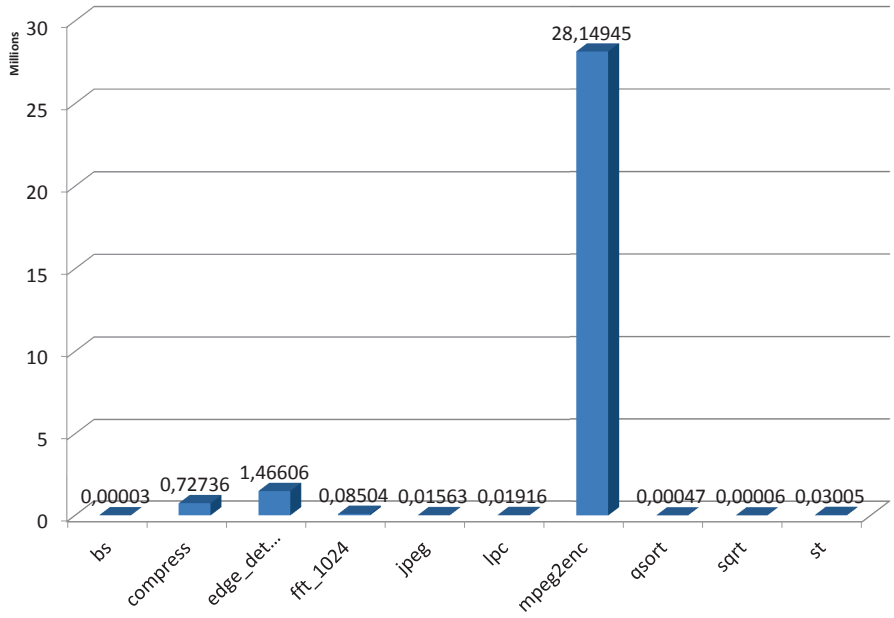


Figure 5.2: Execution counts for our benchmarks ($10^6$).

As indicated, PER values are greatly affected by the execution count. However, execution counts for an application are not directly correlated with error expectations. For example, when a program has many loops or lots of lines of code that take much longer to execute, it does not mean that, that program is more prone to transient faults. Instead, if frequently executed code blocks have major impact in the output, the application would have greater error expectancy.

As explained in Chapter 4, our goal is to reduce the number of statement parameters that require reliability precautions. For this purpose we set a $\lambda_{PER}$ value for the application (discussed in section 4.2) in order to pick the statement parameters that can be eliminated. For instance, a 0 value for $\lambda_{PER}$ means that, the software can not tolerate any errors and only the statement parameters that has 0 ER value can be safely ignored when reliability precautions are applied. Figure 5.3 shows the amount of improvements with our approach when the application is executed without any errors, that is with a $\lambda_{PER}$ value of 0.

According to these results, improvements brought by our approach is 40.85%

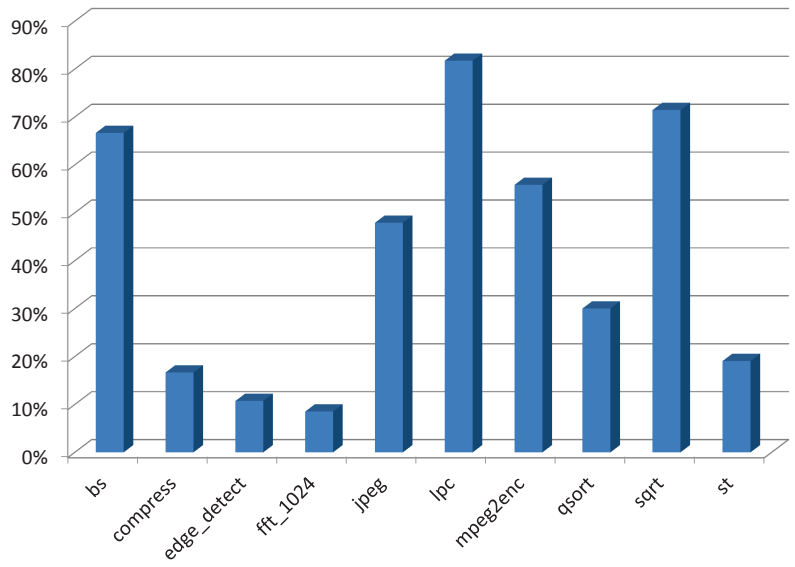


Figure 5.3: Improvements with our approach when the application is executed without any errors, that is $\lambda_{PER} = 0$.

on the average. This indicates that 40.85% of these application code segments can be optimized without any compromises in terms of reliability. The improvements range from 8.5% to 81.75% which shows that our optimizations mostly rely on the characteristics of the application. In the next set of experiments, we use our approach with two different software fault tolerance techniques, EDDI [11] and SWIFT [4]. We compare the execution time, program size, and instruction count improvements, when EDDI and SWIFT are seperately optimized using our framework. Note that, the fault tolerance technique used is orthogonal to our approach and can be chosen independently. In Figure 5.4, benchmark execution times are compared to the baseline SWIFT and EDDI execution times. Our implementation provides improvements between 2.5% and 23.8%, and has a mean value of 12%, when applied over the SWIFT technique. The reduction in the execution times are not as impressive as the previous results. First of all, SWIFT is already optimized, and it can benefit from the advantages of Instruction Level Parallelism (ILP).

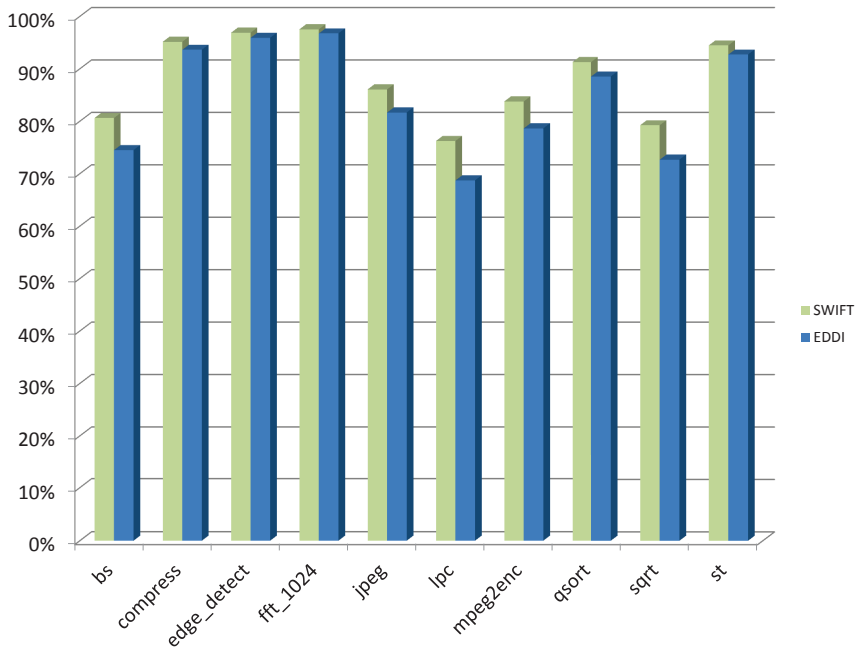


Figure 5.4: Normalized execution times compared to SWIFT and EDDI approaches without applying our technique.

On the other hand, our normalized execution times with respect to base EDDI implementation are between 3.3% and 31.3%, and has a mean value of 15.7%. Next, we give the improvements brought by our approach in the program size. The code size reductions we achieve are shown in Figure 5.5. As can be seen from Figure 5.5, our approach reduces the code size in SWIFT between 5% and 48%, has a mean value of 23.9%. Similarly, for EDDI, our reductions in code size vary 5.6% to 53% has a mean value of 26.5%.

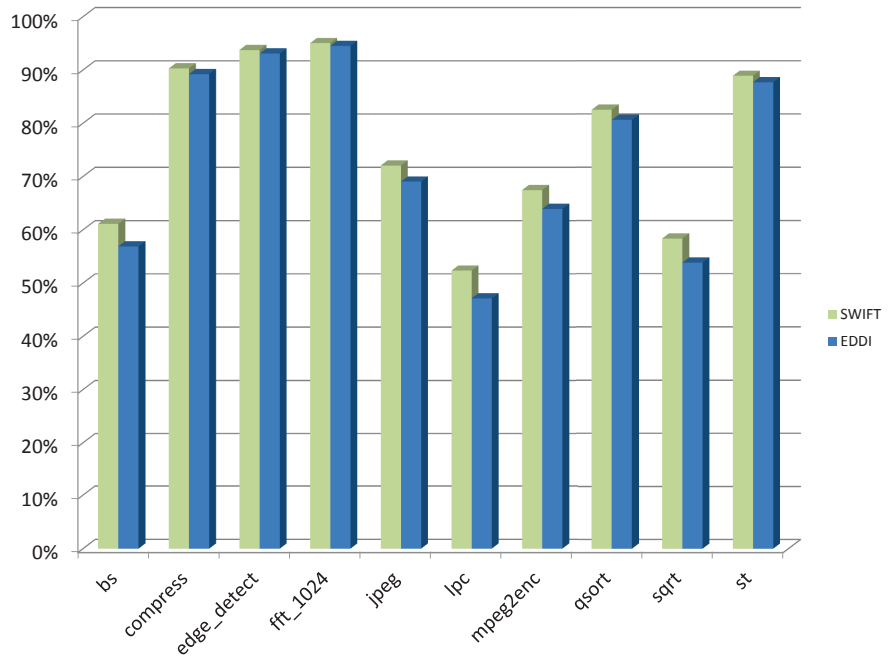


Figure 5.5: Program binary size reductions compared to SWIFT and EDDI approaches without applying SSFT.

Similar to code size, we see considerable reductions in the instruction counts. Specifically, the average instruction count reductions are 22.5% and 25.8% for SWIFT and EDDI, respectively. As can be seen from Figure 5.6, reductions range from 4.7% and 45%, and 5.4% and 52% for SWIFT and EDDI, respectively.

While not presented here, we expect to see a much better improvement in power consumption, since, although ILP techniques improve execution times for SWIFT and EDDI, the power consumptions will not be improved. When all other parameters are the same, power consumption will be similar to the number of executed instructions.

In our experiments discussed so far, we assumed that applications are not tolerable to errors. That is, these applications have $\lambda_{PER} = 0$. However, there are many applications in the soft computing domain which potentially accept results with an error margin. For such applications, our approach is capable of providing much better results since we are able to identify higher number

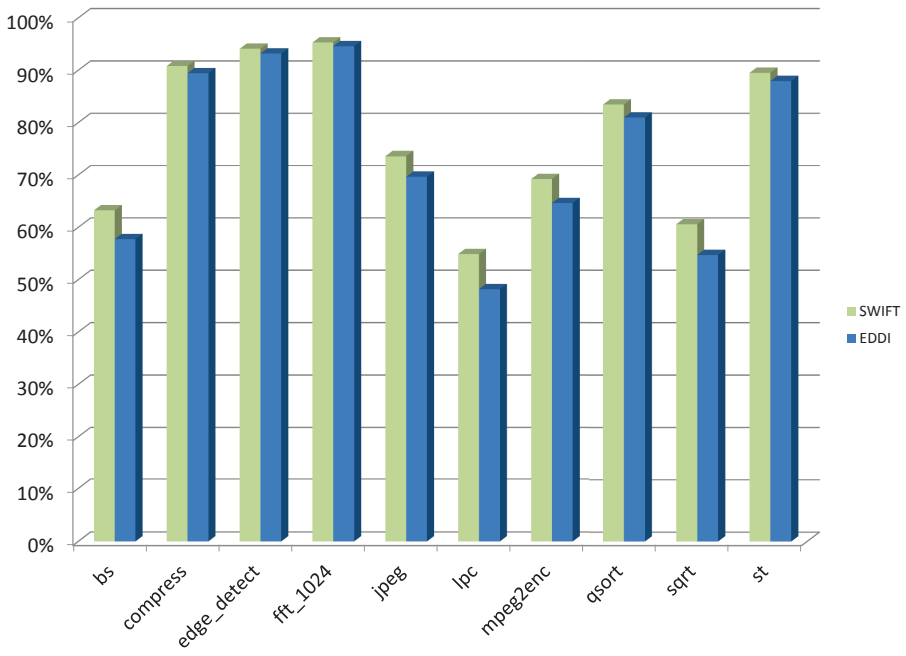


Figure 5.6: Instruction count reductions compared to SWIFT and EDDI approaches without applying our technique.

of statement parameters without violating the error margin indicated by $\lambda_{PER}$ parameter. In the next section, we present a sensitivity analysis on $\lambda_{PER}$ values.

## 5.3 Sensitivity Analysis

For soft applications where a margin of error is acceptable, $\lambda_{PER}$ parameter can be adjusted to increase the amount of parameters that can be removed from the list of protected parameters. Figure 5.7 shows the percentage of parameters that can be excluded from protection without violating the $\lambda_{PER}$ value.

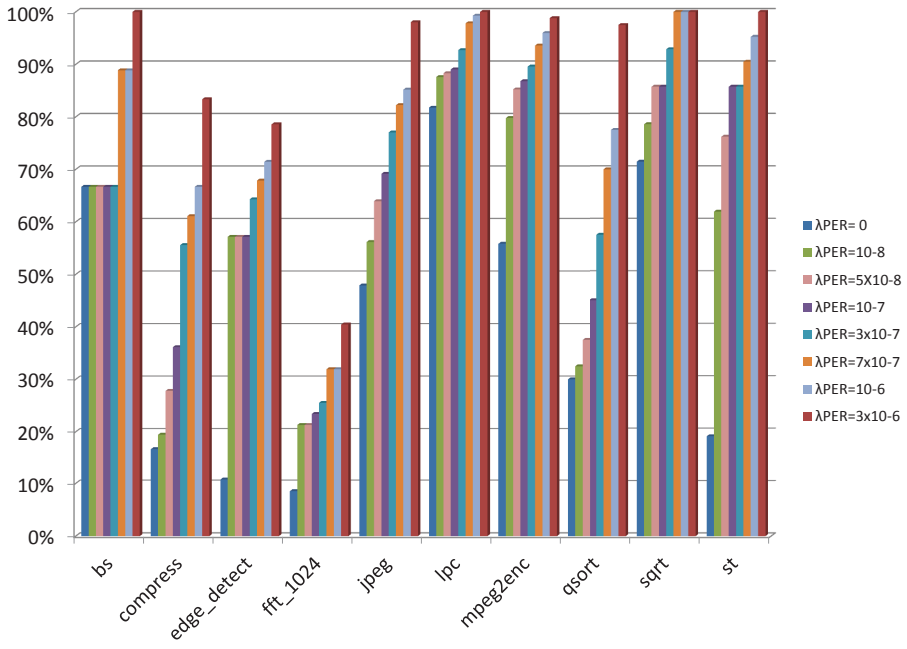


Figure 5.7: The percentage of parameters that can be excluded from software tolerance for different $\lambda_{PER}$ values.

Compared with $\lambda_{PER} = 0$, the amount of parameters excluded from SFT will increase from from 40.85% to 64.5% when $\lambda_{PER} = 10^{-7}$. This improvement highly depends on software characteristics. "st" benchmark improved by 66.7%, "edge_detect" benchmark improved by 46.4%, "mpeg2enc" benchmark improved by 31%, "jpeg" and "compress" benchmarks improved around 20%, "qsort" benchmark and "sqrt" benchmarks improved around 15%, while "bs" benchmark did not improve any further. As seen in Figure 5.7, when $\lambda_{PER}$ value is set to a value that is too high, most of the statement parameters are excluded from fault tolerance, which may not be desirable. The $\lambda_{PER}$ value should be

adjusted according to the application requirements. For the following figures, we used SWIFT approach (without applying SSFT) as baseline and show our improvements over this technique for different $\lambda_{PER}$ values.

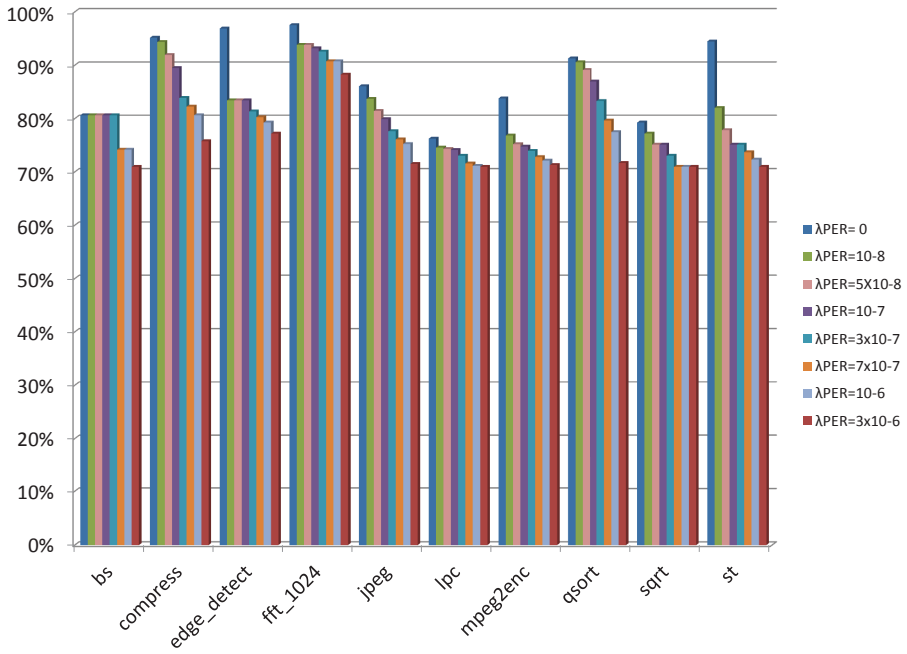


Figure 5.8: Normalized execution times compared to SWIFT approach without applying our technique for different $\lambda_{PER}$ values.

When $\lambda_{PER} = 10^{-7}$, our technique reduces execution time between 6.8% and 25.9% with a mean value of 18.8% for SWIFT technique. The execution times improve further, when $\lambda_{PER}$ is set to higher values.

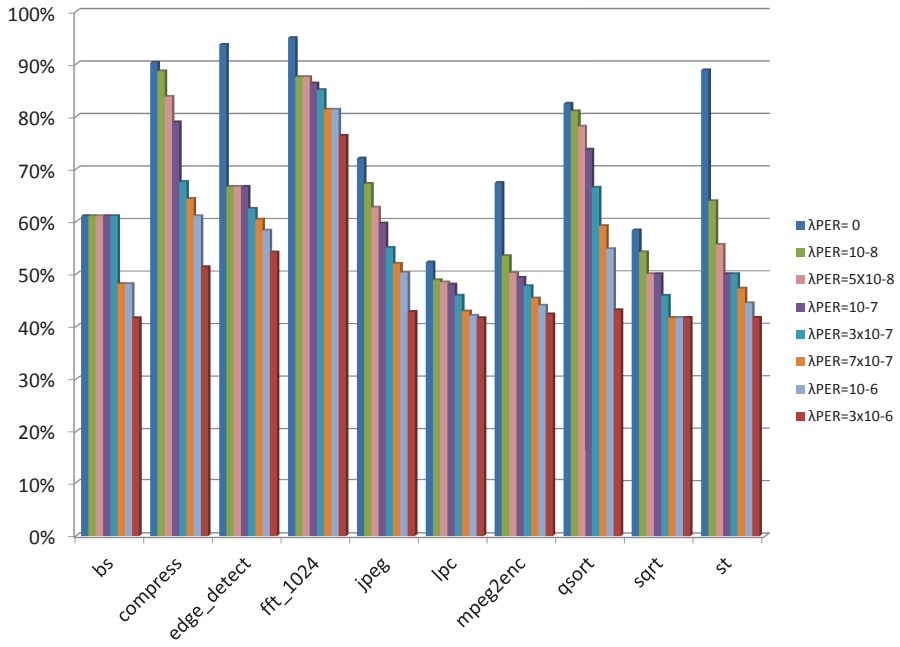


Figure 5.9: Program binary size reductions compared to SWIFT approach without applying our technique for different $\lambda_{PER}$ values.

When $\lambda_{PER} = 10^{-7}$, our technique reduces program size between 13.7% and 52%, with a mean value of 37.6% for SWIFT.

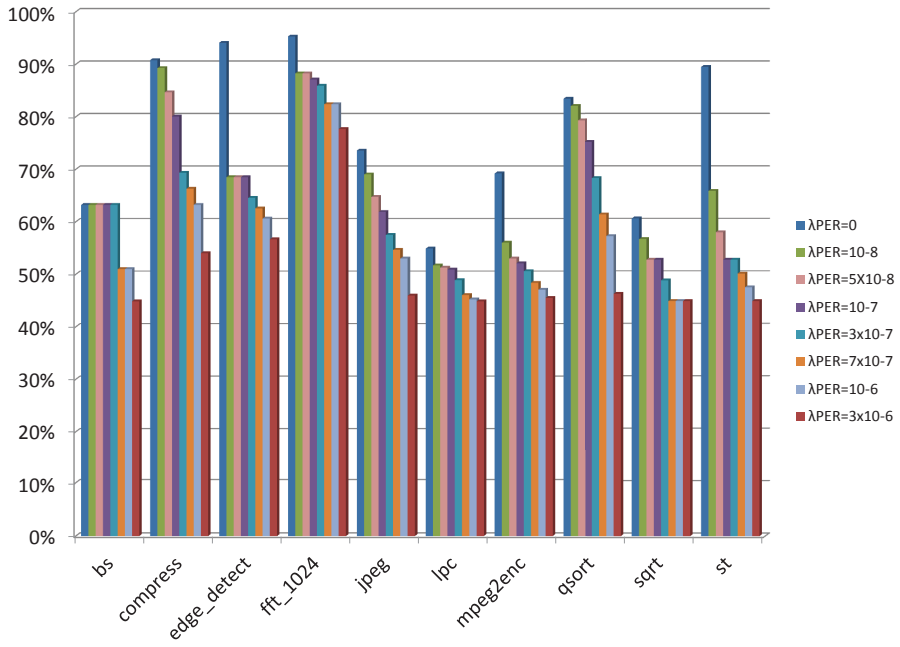


Figure 5.10: Instruction count reductions compared to SWIFT approach without applying our technique for different $\lambda_{PER}$ values.

Our technique reduces instruction count between 12.9% and 49.11%, with a mean value of 35.56% when it is applied on SWIFT and the $\lambda_{PER} = 10^{-7}$.

In the next figure, Figure 5.11, we give the average percentage of parameters that can be excluded from fault tolerance (over all benchmarks) over a range of $\lambda_{PER}$ values. Specifically, when $\lambda_{PER} = 10^{-7}$, on the average 23.5% reduction is possible. Along with other figures, this indicates that, for soft computing applications, SSFT will further reduce the redundancies; hence will provide better performance, higher availability and cheaper overall cost (hardware requirements and power consumptions).

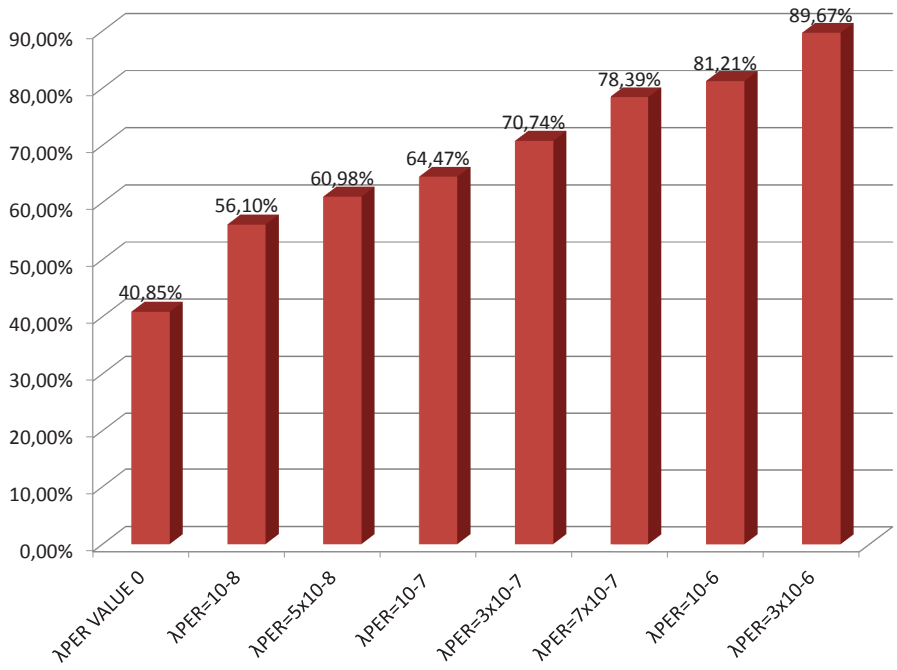


Figure 5.11: The average rate of parameters that can be removed from software tolerance for different $\lambda_{PER}$ values.

# Chapter 6

# Conclusion

In order to improve software reliability and provide protection against transient faults that does not cause permanent damage in hardware, reliability techniques have been proposed. Software reliability can be achieved through hardware, software, or hybrid techniques, which all rely on some form of redundancy. These redundancies cause performance overhead, higher power consumption and increase the cost of hardware components required by the system.

It is necessary to keep the redundancies minimal in order to reduce their effect in the system, while maintaining necessary protection. This requires a better understanding in terms of reliability and performance requirements of the running application and the environmental factors that affect of transient fault occurrence (such as hardware specifications and environmental radiation levels).

We propose SSFT (Selective software fault tolerance) that uses software profiling information to understand the vulnerabilities of the running application in terms of reliability and tries to reduce the redundancies while providing the required levels of protection. Reduction in redundancies will decrease the overall system cost and performance overhead.

Our experiments shows that, even for applications that cannot tolerate any errors and require 0 error expectancy, some optimizations can still be made on the

statement parameters. For instance, for benchmarks that are more amenable to such optimizations (as in LPC benchmark), 81.75% of the statement parameters can be safely omitted without impairing the reliability of the software. A fault tolerance technique that does not have any optimizations would have protected 81.75% of the statement parameters that do not even require protection. The impact of this overprotection can easily be seen from our results. For the same LPC benchmark when SWIFT fault tolerance technique is applied, our approach improves the execution time of the software by 23.8%, the program binary size by 47.7%, and the instruction count by 45.1%. Similarly, when it is applied with EDDI fault tolerance technique, our approach improves the execution time of the software by 31.3%, the program binary size by 52.9%, and the instruction count by 51.8% . The reduced redundancy will provide less power consumption, better performance; therefore improve the availability and will decrease the number of data bits susceptible to soft errors. Additionally, the reduced execution time will allow a cheaper and slower CPU, a smaller non-volatile storage, and a smaller volatile memory. These improvements will reduce the cost of the hardware requirements due to applying the software fault tolerance. On the average, our technique provides 11.9% execution time, 23.9% binary size, and 22.6% instruction count reduction for SWIFT, 15.7% execution time, 26.5% binary size, and 25.9% instruction count reduction for EDDI, respectively.

For soft computing applications that can tolerate some margin of error, i.e, has less strict reliability concerns our technique provides even better reductions. Instead of enforcing zero error, if we allow an error rate (PER) of $10^{-6}$, our benchmarks, on the average, improve from 40.85% to 81.2%. This further improves SWIFT and EDDI execution times by 13% and 18.3% respectively. Similarly, program size is reduced by 30.9% to 35.4%, and instruction count is reduced by 28.73% to 34.5% for SWIFT and EDDI, respectively. These results show that there is more room for improvements for soft computing applications.

One key advantage of our technique is that, since the redundancies have severe effects in terms of performance, power consumption and hardware requirements, a small amount of reduction in redundancies results with a higher rate of improvement. The results above indicate that SSFT provides decent improvements

in terms of performance and great improvements for program binary size and program instruction count while preserving the reliability of the software.

In addition, our technique can be applied in combination with any fault tolerance technique. Hardware, software and hybrid fault tolerance techniques can all benefit from reduced redundancies provided by our approach. Similarly, hardware fault tolerance techniques can be informed by the compiler about the vulnerabilities of the software and can make informed decisions about where in memory each instruction should be placed. For instance, a more critical statement argument or instruction will be placed in an ECC protected memory, whereas the rest of the data can be placed in a parity protected memory, which will reduce the cost of the hardware, reduce the power consumption and improve the response time. Moreover, software techniques can be improved in terms of performance overhead, power consumption, and memory requirements.

Our approach is flexible in the sense that it can be applied to applications that do not tolerate any errors without any compromises. The amount of optimization, however, is affected by the acceptable rate of error, i.e, software that can tolerate some error rate (soft computing applications), can be improved further, and more redundancy can be eliminated when reliability measures are taken. Overall, the application is provided with the most suitable fault tolerance, the overall costs are reduced, and the protection level provided by the fault tolerance is kept in an acceptable level that suits the reliability requirements of the application.

One limitation in our technique is that, it relies on profiling information. Since the software behavior may dramatically change for different data sets, the program should be tested with a variety of data sets, in order to provide a more representative profiling information.

Another limitation is that, the impact of injected error also depends on the parameter and resulting data. In order to better simulate error injections, bit flips should be randomized and the impact of different forms of bit flips should be normalized. To obtain accurate results it is necessary to perform multiple tests.

As future work, we are planning to extend our approach such that it will

enable hybrid techniques like CRTR and SRTR to utilize, and hardware fault tolerance such as ECC protected memory to use it.

We also would like to implement fault injections and redundancy reductions on array and pointer parameters. This was not possible due to limitations in GCC framework. We expect to see further improvements with such additions.

# Bibliography


[1] R. Baumann, "Soft errors in advanced computer systems," *IEEE Des. Test*, vol. 22, pp. 258–266, May 2005.

[2] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. August, "Design and evaluation of hybrid fault-detection systems," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pp. 148–159, IEEE, 2005.

[3] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, 2005.

[4] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, pp. 243–254, IEEE Computer Society, 2005.

[5] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 329–335, 2005.

[6] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented edac protection against seus," *Reliability, IEEE Transactions on*, vol. 49, no. 3, pp. 273–284, 2000.

[7] G. A. Reis, *Software modulated fault tolerance*. Princeton University, 2008.

[8] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled fault tolerance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, pp. 366–396, 2005.

[9] G. Munkby and S. Schupp, "Type inference for soft-error fault-tolerance prediction," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pp. 65–75, IEEE, 2009.

[10] M. Hiller, A. Jhumka, and N. Suri, "Epic: Profiling the propagation and effect of data errors in software," *Computers, IEEE Transactions on*, vol. 53, no. 5, pp. 512–530, 2004.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63–75, 2002.

[12] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 29–40, IEEE, 2003.

[13] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 98–109, IEEE, 2003.

[14] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into vhdl models: the mefisto tool," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pp. 66–75, IEEE, 1994.

[15] V. Sieh, O. Tschache, and F. Balbach, "Verify: evaluation of reliability using vhdl-models with embedded fault descriptions," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pp. 32–36, IEEE, 1997.

[16] K. K. Goswami and R. K. Iyer, "A simulation-based study of a triple modular redundant system using depend," in *Fault-Tolerant Computing Systems*, pp. 300–311, Springer, 1991.

[17] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, "Rifle: A general purpose pin-level fault injector," in *Dependable ComputingEDCC-1*, pp. 197–216, Springer, 1994.

[18] J. Arlat, 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," *Software Engineering, IEEE Transactions on*, vol. 16, no. 2, pp. 166–182, 1990.

[19] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "Fiat-fault injection based automated testing environment," in *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pp. 102–107, IEEE, 1988.

[20] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *Computers, IEEE Transactions on*, vol. 44, no. 2, pp. 248–260, 1995.

[21] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, pp. 91–100, IEEE, 2000.

[22] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Goofi: Generic object-oriented fault injection tool," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pp. 83–88, IEEE, 2001.

[23] M. Hiller, A. Jhumka, and N. Suri, "Propane: an environment for examining the propagation of errors in software," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 81–85, 2002.

[24] J. Freschl and D. Xue, "Swif-it: A tool for memory fault injection and protection," 2005.

[25] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple instruction issue in the nonstop cyclone system," 1990.

[26] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, *et al.*, "Ibm's s/390 g5 microprocessor design," *Micro, IEEE*, vol. 19, no. 2, pp. 12–23, 1999.

[27] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1, pp. 293–307, IEEE, 1996.

[28] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 87–98, 2002.

[29] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pp. 99–110, IEEE, 2002.

[30] E. Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pp. 84–91, IEEE, 1999.

[31] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *ACM SIGARCH Computer Architecture News*, vol. 32, p. 264, IEEE Computer Society, 2004.

[32] Intel Corporation, "Rachet up reliability for mission-critical applications," 2012.

[33] J. G. Holm and P. Banerjee, "Low cost concurrent error detection in a vliw architecture using replicated instructions.," in *ICPP (1)*, pp. 192–195, 1992.

[34] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pp. 137–143, IEEE, 2003.

[35] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pp. 33–42, IEEE, 2001.

[36] N. Oh and E. J. McCluskey, "Low energy error detection technique using procedure call duplication," in *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*, 2001.

[37] A. Shye, V. Janapa, M. Daniel, and A. Connors, "Transient fault tolerance via dynamic process-level redundancy," 2006.

[38] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pp. 83–92, IEEE, 2006.

[39] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 392–403, ACM, 1995.

[40] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *ACM SIGARCH Computer Architecture News*, vol. 24, pp. 191–202, ACM, 1996.

[41] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 25–36, ACM, 2000.

[42] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: improving both performance and fault tolerance," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 257–268, 2000.

[43] G. Jacques-Silva, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer, "Fault injection-based assessment of partial fault tolerance in stream processing applications," in *Proceedings of the 5th ACM international conference on Distributed event-based system*, pp. 231–242, ACM, 2011.

[44] Richard M. Stallman and the GCC Developer Community, "Gnu compiler collection internals," 2010.

[45] E. Normand, "Single-event effects in avionics," *Nuclear Science, IEEE Transactions on*, vol. 43, no. 2, pp. 461–474, 1996.

[46] C. Lee and M. Stoodley, "Utdsp benchmark suite," 1998.

[47] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 330–335, IEEE Computer Society, 1997.