Improving redundant multithreading performance for soft-error detection in HPC applications

Costa Rica Institute of Technology, Master's Degree Thesis

By

Diego Simon Perez Arroyo

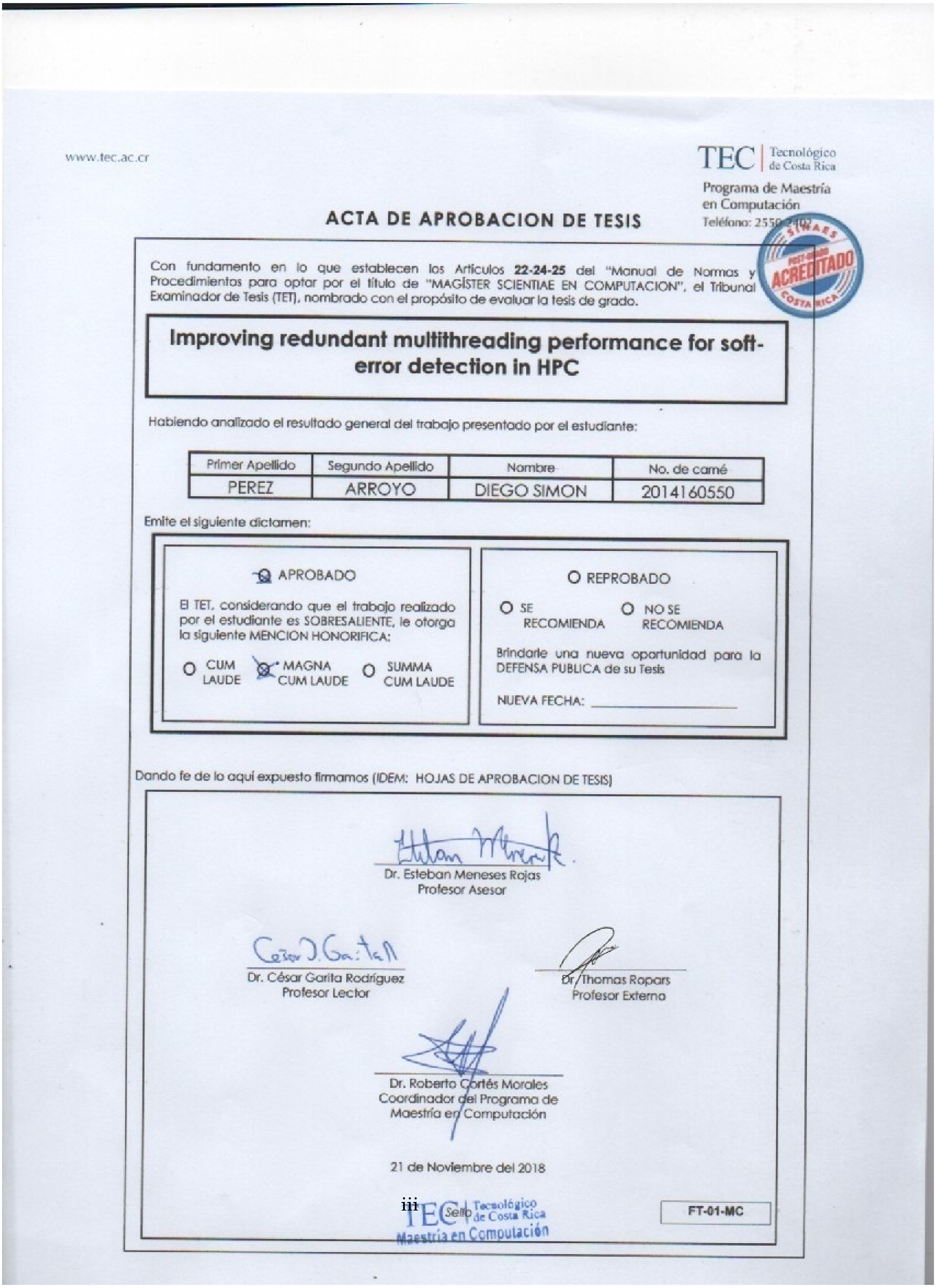
Advisors:
Esteban Meneses, PhD &
Thomas Ropars, PhD

TEC Tecnológico
de Costa Rica

School of Computing
Master of Computer Science Program

Thesis submitted to the consideration of the Computation Department, to opt for the degree of Magister Scientiae in Computing, with emphasis in Computer Science.

December 4, 2018



www.tec.ac.cr

TEC | Tecnológico de Costa Rica

Programa de Maestría en Computación
Teléfono: 2550-2402

SINAES POST-GRADO ACREDITADO COSTA RICA

ACTA DE APROBACION DE TESIS

Con fundamento en lo que establecen los Artículos **22-24-25** del "Manual de Normas y Procedimientos para optar por el título de "MAGÍSTER SCIENTIAE EN COMPUTACION", el Tribunal Examinador de Tesis (TET), nombrado con el propósito de evaluar la tesis de grado.

Improving redundant multithreading performance for soft-error detection in HPC

Habiendo analizado el resultado general del trabajo presentado por el estudiante:

Primer Apellido	Segundo Apellido	Nombre	No. de carné
PEREZ	ARROYO	DIEGO SIMON	2014160550

Emite el siguiente dictamen:

☒ APROBADO

El TET, considerando que el trabajo realizado por el estudiante es SOBRESALIENTE, le otorga la siguiente MENCION HONORIFICA:

○ CUM LAUDE ☒ MAGNA CUM LAUDE ○ SUMMA CUM LAUDE

○ REPROBADO

○ SE RECOMIENDA ○ NO SE RECOMIENDA

Brindarle una nueva oportunidad para la DEFENSA PUBLICA de su Tesis

NUEVA FECHA: ____________________

Dando fe de lo aquí expuesto firmamos (*IDEM*: HOJAS DE APROBACION DE TESIS)

Dr. Esteban Meneses Rojas
Profesor Asesor

César J. Garita

Dr. César Garita Rodríguez
Profesor Lector

Dr. Thomas Ropars
Profesor Externo

Dr. Roberto Cortés Morales
Coordinador del Programa de Maestría en Computación

21 de Noviembre del 2018

Sello TEC Tecnológico de Costa Rica
Maestría en Computación

FT-01-MC

Abstract

As HPC systems move towards extreme scale, soft errors leading to silent data corruptions become a major concern. In this thesis, we propose a set of three optimizations to the classical Redundant Multithreading (RMT) approach to allow faster soft error detection. First, we leverage the use of Simultaneous Multithreading (SMT) to collocate sibling replicated threads on the same physical core to efficiently exchange data to expose errors. Some HPC applications cannot fully exploit SMT for performance improvement and instead, we propose to use these additional resources for fault tolerance. Second, we present variable aggregation to group several values together and use this merged value to speed up detection of soft errors. Third, we introduce selective checking to decrease the number of checked values to a minimum. The last two techniques reduce the overall performance overhead by relaxing the soft error detection scope. Our experimental evaluation, executed on recent multicore processors with representative HPC benchmarks, proves that the use of SMT for fault tolerance can enhance RMT performance. It also shows that, at constant computing power budget, with optimizations applied, the overhead of the technique can be significantly lower than the classical RMT replicated execution. Furthermore, these results show that RMT can be a viable solution for soft-error detection at extreme scale.

Conforme los sistemas de HPC se mueven hacia una escala extrema, los *soft errors* que producen corrupciones de datos silenciosas se convierten en una gran preocupación. En esta tesis, proponemos un conjunto de tres optimizaciones al clásico enfoque de Redundant Multitreading (RMT) para permitir más rápida detección de *soft errors*. En primer lugar, aprovechamos el uso de *Simultaneous Multithreading* (SMT) para colocar los hilos hermanos de replicación en el mismo core físico, para eficientemente intercambiar datos y descubrir errores. Algunas aplicaciones de HPC no pueden aprovechar totalmente SMT para mejorar su rendimiento y en cambio, proponemos utilizar estos recursos adicionales para tolerancia a fallas. En segundo lugar, presentamos *variable aggregation* para agrupar múltiples valores en uno sólo y utilizar este último para acelerar la detección de *soft errors*. Terceramente, introducimos *selective checking* para disminuir el número de valores chequeados a un mínimo. Las últimas dos técnicas reducen la sobrecarga de rendimiento en general, al relajar el alcance de la detección de errores. Nuestras evaluaciones experimentales, ejecutadas en procesadores multi-núcleo modernos con pruebas representativas de HPC, demuestran que el uso de SMT para tolerancia a fallas puede mejorar el rendimiento de RMT. También muestran que a un presupuesto de poder de computacional constante, con las optimizaciones aplicadas, la sobrecarga de rendimiento de la técnica puede ser significativamente menor que la ejecución replicada utilizando el clásico enfoque de RMT. Más aún, estos resultados muestrans que RMT puede ser una solución viable para detección de *soft errors* en una escala extrema.

Index terms— Soft Errors, Bit Flips, Soft Error Detection, Redundant MultiThreading, Hyper-Threading, Hyper-Threads, HPC, Performance Improvement, Algorithms, Detección de Soft Errors, Computación de Alto Rendimiento, Mejoras de Rendimiento, Algoritmos

Dedication and acknowledgements

I dedicate this thesis to my parents Sonia and Danilo. They have taught me so many important lessons in my life, so many that I could never finish thanking them.

This thesis would never have been possible without the help and guidance of both my tutors, Thomas and Esteban. I have nothing more to say to them but to express my sincerest gratitude for everything they have done throughout the journey of this thesis.

My girlfriend Sofia has also been an important part of my life during this part of my life, I thank her with all my heart for all the efforts that she has done supporting me on this adventure. Special acknowledgements goes to Olger Calderon, the best master's colleague I could ever ask for.

We also thank the School of Computing at the Costa Rica Institute of Technology for a travel grant.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Processor's manufacturers have been able to keep making faster and less energy consuming chips for a long time. This has been possible because of several innovative improvements in several fields, like architecture design, better fabrication materials, among others; one example of this are the smaller and faster transistors with tighter noise margins and low threshold voltages currently present in processors. Another example are multicore chips, a current trend in technology, in which several computing units are placed in the same chip sharing different kinds of resources. This allows more than one instruction to be processed concurrently and therefore, if the problem allows it, reduces significantly the overall execution time of a calculation. Even though the combination of such breakthroughs in processor manufacturing have created really fast and efficient chips, it has also made them very complex systems which are more susceptible to *transient errors* than previous generations [28].

Transient errors (also known as soft errors) are different from design or manufacturing errors in one aspect in particular: they are caused by external events and hence they are pretty much unpredictable. If, for example, a poor design choice causes overheating on a specific core after an amount of time, it can be objectively quantified and determined. Soft errors are caused by foreign factors such as high-energy particles striking the chip, which are more difficult to measure since they cannot always be anticipated. These events do not provoke permanent physical damage on the processor, but they can cause a *bit flip* (a change of state in a transistor) that can alter data silently (without the hardware noticing) and potentially corrupt the program state.

Main memory already has protection mechanisms like error correction codes (ECC) against bit flips, ensuring that every read value is the same as the one that was written; but sadly processor's cache and registers are still vulnerable. Supercomputers nowadays are very expensive systems used to calculate complex data. Therefore, the time they spend to compute a result is commonly long. For example running a complex weather simulation in order to produce an accurate forecast; or the right type of chemotherapy is better to use on a specific patient. If a silent data corruption affects the outcome in such scenarios, it can cause one out of two situations. Either the result is so compromised that is easy to verify that is wrong or it could be mistakenly accepted. But, either way the time and physical resources the supercomputer spent processing would have been wasted. Plus, in the worst case scenario where the soft error goes undetected and the result is interpreted as correct, a wrong valuable conclusion can be drawn form it.

To protect against these kind of errors, replication is typically used. The result of a calculation is performed not once, but multiple times and the results are compared. Replication comes in two flavors: hardware or software. There are many types of redundant hardware specialized to be reliable against this sort of problem, but are more costly than regular hardware. Software replication on the other hand, allows common commercial processors to implement this mechanism as well; making it the common solution for supercomputers and for this thesis as well [37].

There are two main challenges when dealing with soft errors. The first one is being able to detect that an error happened. The second is correcting the error and ensuring that the final result of the calculation is correct. In common solutions, redundancy is used at some level. Hardware replication, though efficient, requires expensive specialized micro chips, however software replication allows common hardware to implement this feature. Usually, instructions are replicated and frequent checks are placed to detect errors. If the check fails then the application is usually restored to a previous checkpoint. But, as expected from replicating the application, this technique incurs a lot of performance overhead.

Software replication can also be further classified into two categories, Instruction Level Redundancy (ILR) or thread-local replication and Redundant Multithreading (RMT). Basically the difference between them is whether the replication happens in the same thread or is distributed into two threads, respectively. Redundant Multithreading approaches [23] [34] [37] try to minimize the overhead that ILR produces, by separating the work into two threads: one that only calculates values from the original code and another one that also calculates values and checks if both results are the same.

In Redundant Multithreading schemes the inter-thread communication is the performance bottleneck. We propose a set of three optimizations to the classical RMT approach to allow faster soft error detection. The first one is to leverage Simultaneous Multithreading (SMT) to allow faster data transfer between sibling replicated threads by placing them on the same physical core. SMT threads share the cache L1 of a physical processor, among other execution resources [20]. The fact that this level of cache is shared among them, makes the data exchange significantly faster than if threads were pinned on different cores. More importantly, some HPC applications cannot fully exploit SMT for performance improvement and instead, we propose to better utilize these additional resources for fault tolerance.

The second optimization we propose is called *variable aggregation*. It reduces the communication traffic between threads by grouping multiple values together into a single merged value, while still delivering soft error detection. Finally, we introduce *selective checking* to reduce to a minimum the amount of checked values, by identifying the points of the application where checks are strictly necessary. The last two improvements relax the soft error detection scope but reduce significantly the overall performance overhead of RMT.

Our experiments, executed with HPC representative benchmarks on recent multicore processors prove that using SMT for fault tolerance, enhances RMT performance by just pinning threads to the correct cores. It also shows that with optimizations applied, the general overhead of the technique can be significantly better than the classical RMT replicated execution. Furthermore, these results also show that RMT can be a viable solution for soft-error detection at extreme scale.

Chapter 2

Background

The objective on this chapter is to provide an overview of concepts, techniques and other information necessary to properly understand the current thesis. For that, in the first section, concepts of faults, errors and failures on a computing systems are defined. Later on, some causes and consequences of soft errors in particular are also mentioned. The chapter then continues with 3 more sections. Detecting soft errors, where the most common used strategies are presented and the ones of more interested to the thesis are further explained; like the case with Redundant Multi-Threading error detection technique and its different versions, based on the communication pattern between threads. Correcting soft errors section, which tries to refer current schemes of recovery once an error has been identified. Finally, the simultaneous multi-threading section defines such technique, presents the Intel version (Hyper-Treading) and explains how this feature can be used to accomplish efficient soft-error detection.

2.1 Faults, Errors and Failures

The current thesis proposal relies on the following concepts defined in [2]:

- The *function* of a system is what the system is intended for, and is described by the specification in terms in functionality and performance.
- *Correct service* is delivered when the service implements the system function.
- A *system failure* is an event that occurs when the deliver service deviates from the correct service.
- An *error* is the part of the system state that may cause a subsequent system failure.

- A *fault* is the adjudged or hypothesized cause of an error.

Having stated such concepts, it is easier to analyze soft errors. One natural question about them is: how often do they happen? Should a regular laptop customer be aware of such errors? In [7] a large study of 193 data centers over 16 months was conducted. The figure 2.1 shows the amount of single-bit memory errors reported on the systems. Although most of the centers experienced very few faults, the image does serve the purpose of demonstrating that soft errors really do happen in real life computers. One point to mention is that the data centers on such analysis were not protected with ECC or any other of memory error code correction mechanism. Most of today's supercomputers have some kind of protection on main memory, but processors registers and caches are still vulnerable to these kinds of faults.

In general the probability of a soft error is low, but the error rate in future HPC super computers is expected to increase [6]. Even now, with current low errors rates, many silent data corruptions have provoked important loses. Sun Microsystems, for example, has acknowledged that important clients such as eBay, American Online and Los Alamos National Laboratory have experienced system failures caused by transient faults [22].

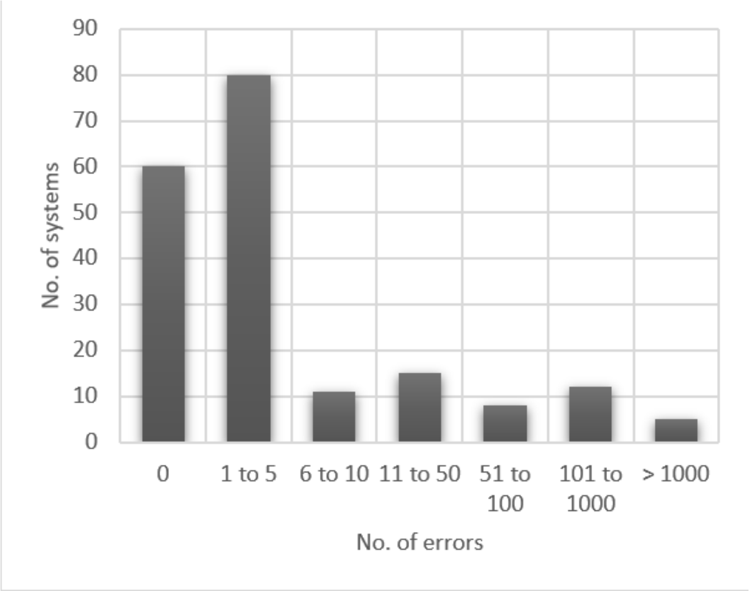


FIGURE 2.1. Histogram of the number of memory single-bit errors reported by 193 systems over 16 months, data from [7].

2.2 Causes and Consequences of Soft Errors

Among the most commonly mentioned causes for soft errors are energetic-particle strikes and fluctuating power supply [15] [28] [37]. Smaller transistor's sizes and lower power voltages allow faster and less energy consuming chips, but it also makes them more susceptible to neutron and alpha particles [7]. There have been other studies such as the one in [3], that show evidence that other factors such as temperature and the position of the sun in the sky (actually it would be more correct to say the position of the earth related to the sun) are also related to the presence of soft errors.

Consequences of soft errors are very different. A single bit flip can be as dangerous to cause a whole system failure or as innocent to provoke absolutely nothing. It is possible to classify the consequences in two types: visible and silent. In first one, there are events such as segmentation faults, which can happen if a bit flips changes the address of a read/store to non accessible memory. Another example of this category is an infinite loop, a bit flip can modify a loop variable resulting in an endless scenario. This type of consequences result in a system detectable event, usually the operating system will notice it and do something about it, like killing the process. Although it is quite serious that the application is stopped and has to be restarted, at least the user knows that something went wrong.

The other type of consequences are the ones that can be more dangerous, the silent ones. A soft error can go undetected for several reasons. It may be because it happened on an address that was not being used, or that was not going to be used anymore; in either case it does not cause a failure and the system can finish correctly. There are also cases in which depending on the algorithm properties the bit flip can be masked. An iterative process that converges to a result may take a couple of extra iterations because a soft error modified the value being calculated, but can still complete normally; of course, that depends on how the soft error corrupts the value.

But sometimes, in the most unfortunate scenario, the error can alter important data without being detected. The algorithm continues as if nothing happened and the final result can be significantly compromised. In the case of supercomputers that are used for expensive calculations, all the time the system took to finish might have been totally in vain because of a soft error. On certain occasions, the output can be determined to be correct if it falls in a certain interval, depending on the algorithm properties. However, there might be a very unlucky scenario, in which the final value is assumed to be correct and important decisions are made with it. But, whether or not it can be established that the final value is error-free, all the time spent producing it was wasted; which at the end results in money looses.

In order to deal with soft errors, two main phases have to be performed: detection and correction. Section 2.3 describes the first one, while the latter one is presented in Section 2.4.

2.3 Detecting Soft Errors

For HPC applications, different approaches have been proposed to detect via software the soft errors that are difficult to detect at the hardware level. Some approaches rely on full replication [9, 25]: the flow of the application is replicated and the outputs of sibling instances are compared to detect differences that would indicate that an SDC has occurred. Such a technique can detect SDCs with high precision, but it is a costly solution as it doubles the amount of resources required to run the application and it may slow down the execution time due to the required synchronization between the replicas. The core idea of replication lies in the low probability of soft errors. Since it is quite unlikely that a bit flip happens even once, by replicating the instructions and comparing their results, the probability of a soft error corrupting data inadvertently can be further reduced. It would be very unlikely that two bit flips happen one after the other one at very specific time periods, at the two precise memory addresses where the values to be compare are and at the same precise bit in both values.

The other main approach to detect soft errors relies on data analysis to detect unexpected variations in the values of some program variables, that could reveal the occurrence of SDCs [4, 31]. Such solutions induce much less overhead than replication. However, these techniques are only applicable if the data of the application changes in a predictable way, which is not always the case [5].

As mentioned in Chapter 1, replication can be classified into hardware and software approaches. In the next two subsections each one of them is explained.

2.3.1 Hardware Replication

Hardware redundant approaches are transparent to the programmers. Specialized hardware is in charge of replicating and comparing the instructions in order to be reliable against soft errors. Many approaches are mentioned in [28] like a *watchdog* processor to compare the values against the main running processor. There are real system like the ones in the Boeing 777 airplanes [35] which replicates the processor and use checkers to validate the redundant computations.

There are other, somewhat unusual, ways to perform hardware detection for soft errors. Upasani *et al* in [32] present a physical way to detect particle strikes. The basic idea is to add to the processor *acoustic wave detectors* in order to be able to literally hear a particle strike. A more detailed explanation presented in the same paper:

"*Alpha and neutron particles can cause soft errors in semiconductor devices. Upon a collision of a particle with a silicon nucleus, the ionization process creates a large number of electron-hole pairs, which subsequently produce phonons and photons. Generation of phonons and photons indicate that a particle strike results into a shockwave of sound, a flash of light or a small amount of heat for a very small period of time. Therefore, we may try to detect particle strikes by detecting the sound, light or heat.*"

The authors in [32] choose an acoustic wave detector to identify particles strikes through the sound they generate. They claim that the type of device they propose leads to few false positives and that it is not too costly to be integrated into a common processor. While it is very interesting to know there are other ways to prevent soft errors, such approaches are out of the scope of the current work.

These specialized hardware are more costly than the commercial ones. Which is to be expected since there are more physical parts and more logic is necessary in the chip. Also, this kind of hardware might not be all the time necessary and if it cannot be turned off, it means such expensive resources may be wasted sometimes. For that, physical replication is not the common choice and although there are many design proposals regarding how is best to replicate in hardware, the current focus of the thesis is software replication used to detect soft errors.

2.3.2 Software Replication

Software replication approaches are more attractive because, contrary to their counterpart, are much cheaper since they do not require expensive specialized hardware. Of course there are some disadvantages of software-only schemes. They usually incur non negligible resource overhead (time, cores, memory) and they are also not transparent for the programmer. Software redundancy can be classified in two levels, process or thread level. Process replication creates a full process clone of the application (which duplicates the memory footprint but maintains and protects each process data in a separate block). Values are shared between processes to compare results and detect any mismatch that could indicate an SDC. At thread level, data is not fully replicated as it can be shared between sibling threads, also synchronization and communication between threads is easier than with processes. In this work we focus our attention to thread-level replication mechanisms, they can be further classified into Instruction Level Redundancy (ILR) and Redundant Multi-Threading (RMT).

2.3.2.1 Instruction Level Redundancy

In this category, also called thread-local error detection, all is accomplished in the main thread. Instructions are replicated, creating a separate (shadow) data flow along the original one, and integrity checks are added in order to detect errors. The second data flow works using different registers, therefore allowing safe value comparisons by the checks. Also, since there is no dependency between the master and shadow instructions they can potentially be executed in parallel leveraging from instruction-level-parallelism present in modern processors [15]. The Figure 2.2 shows a common ILR code transformation. A simple sum in performed twice in the main thread and extra operations are added to perform the integrity checks.

2.3.2.2 Redundant Multi-Threading

The second group of software replication at the thread level is redundant multi-threading error detection. In this case, the work is distributed in two threads, they can be called *producer* or *leading* thread and *consumer* or *trailing* thread. In ILR error detection, the code size of the application increases significantly and all the checks are added on the critical path of the program, resulting in a lot of performance overhead. Redundant multi-threading tries to solve this issue by

```
a = 5
b = 8
c = a + b
print(c)
```

(a) Original code

```
a1 = 5
b1 = 8
a2 = 5
b2 = 8
c1 = a1 + b1
c2 = a2 + b2
check c1 , c2
print(c)
```

(b) Replicated code with ILR

Figure 2.2: Code transformation with ILR

distributing the work in two threads; this makes sense since multi-core chips are so popular in the market [23] [34] [37].

The original application code is replicated in both threads. Every time a value needs to be checked for soft errors, the leading thread produces the value, the trailing thread consumes it and checks it against its own calculated value. RMT removes the checks from the critical path of the application by having the second thread be the one that compares the results. Figure 2.3 shows how the original code from Figure 2.2 can be replicated into siblings RMT threads; blue color in this case is used to denote the new instructions added for error detection.

```
a = 5
b = 8
c = a + b
produce(c)
print(c)
```

(a) Producer thread code

```
a = 5
b = 8
c = a + b
c2 = consume()
check(c,c2)
```

(b) Consumer thread code

Figure 2.3: Code transformation with RMT

In ILR as well as in RMT not all original instructions are duplicated; for example in both Figures 2.2 and 2.3 the *print* instruction is only executed once. It is common practice [15] [23] [34] [37] that library function calls as well as store/loads to/from memory are excluded from replication. In the case of library functions, since the code is not available to instrument and the result from two calls of the same method might be different (as with the case of “*rand()*” that generates random numbers), the returned value of the procedure is shared from the leading to the trailing thread.

Stores and loads from memory are also not replicated but, the addresses and values are checked in the trailing thread in order to make sure those operations run free of error. The reason is because soft error detection techniques focus on faults happening in the processor and not in the memory, since ECC and other protection mechanisms exist for RAM, but the processor’s registers are still vulnerable [11]. That is why, once a value has been loaded from memory, it is then shared from the leading to the trailing thread.

One downside of redundant multi-threading is that the threads need to be in constant communication, in order to be able to compare the results each of them obtained. Therefore, the inter-thread data sharing mechanism is the performance bottleneck of such solutions. This problem does not happen in the thread-local variant since everything happens in the same thread. Still, there are are some options such as [23] [34] [37] that are able to obtain acceptable performance overheads. In the current thesis we focus on redundant multi-threading approaches.

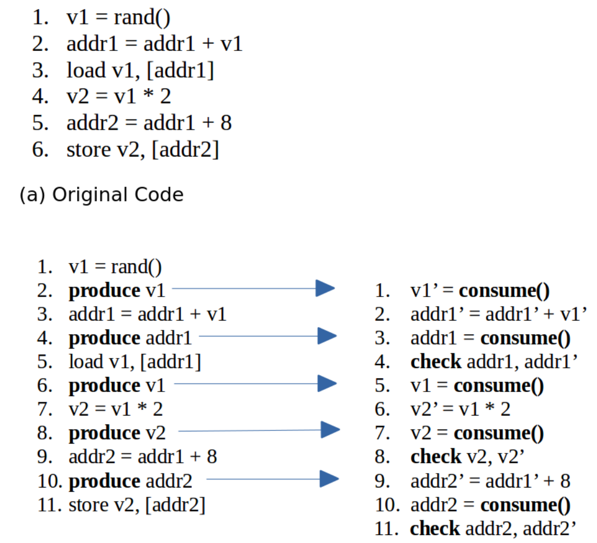
Asynchronous Communication Pattern

Inter-thread communication is commonly done via a Singe Producer/Single Consumer (SPSC) queue. DAFT [37] categorizes some communications patterns among Redundant Multi-Threading approaches. In the example of Figure 2.4 there is an asynchronous (also called unidirectional) communication pattern, where the producer does not wait for the check of the consumer, it simply pushes a value into the queue and keeps on running. In this case, if an error occurs in the leading thread that causes data corruption, the trailing thread will detect it later. But since the producer may have already made other operations with an incorrect value, the whole process must be stopped or fixed. As long as the instructions the leading thread performs do not escape its scope (local thread memory), this mechanism works fine.

However, the use of *volatile variable accesses* makes the problem more complicated. A volatile variable is one that may be modified in ways unknown to the implementation or have other unknown side effects. Memory-mapped IO accesses are an example of volatile variable accesses [37], like printing to the monitor, writing to a file, communicating through the network. Therefore, if the leading thread executes one of these operations with a soft error, the consequences can be catastrophic and irreversible; corrupting a file or sending incorrect data to another node are some examples. These operations are the ones that need to be fully validated before executing them. Therefore, it would not make sense that the trailing thread reports that an error has occurred, after it has already provoked a terrible irremediable effect. However, there might be cases where there are no volatile stores and this communication pattern is sufficient.

Synchronous Communication Pattern

On the other hand, when there are volatile stores the above mentioned scheme is not safe enough. Because of such problem, the safest option is that every time the leading thread performs any memory operation, instead of continuing, it waits for the trailing check to confirm that such operation is error-free and then makes forward progress. So, the communication pattern would be synchronous (or bi-directional). An example of this approach is shown is Figure 2.5; the right-to-left arrows denote the times the producer has to wait for the consumer. This approach implies that threads are synchronized every time a value needs to be checked. The way this is



```
1. v1 = rand()
2. addr1 = addr1 + v1
3. load v1, [addr1]
4. v2 = v1 * 2
5. addr2 = addr1 + 8
6. store v2, [addr2]
```

(a) Original Code

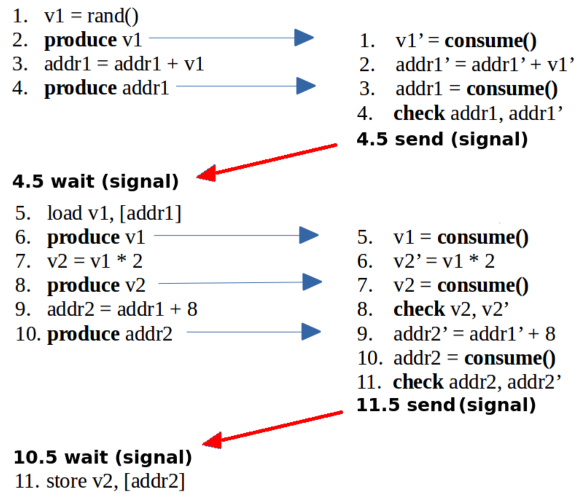
```
1.  v1 = rand()
2.  produce v1
3.  addr1 = addr1 + v1
4.  produce addr1
5.  load v1, [addr1]
6.  produce v1
7.  v2 = v1 * 2
8.  produce v2
9.  addr2 = addr1 + 8
10. produce addr2
11. store v2, [addr2]
```

```
1.  v1' = consume()
2.  addr1' = addr1' + v1'
3.  addr1 = consume()
4.  check addr1, addr1'
5.  v1 = consume()
6.  v2' = v1 * 2
7.  v2 = consume()
8.  check v2, v2'
9.  addr2' = addr1' + 8
10. addr2 = consume()
11. check addr2, addr2'
```

(b) Leading thread code (c) Trailing thread code

FIGURE 2.4. Asynchronous Code Transformation using RMT

done in the literature is via non-locking mechanisms because involving the Operative System (semaphores or mutexes) would be too expensive, so usually spin-wait loops are used. Sadly, even so, the downside of such approach, is that it increases the already non negligible overhead of asynchronous inter-thread communication, as demonstrated in DAFT [37]. The producer now spends a lot of time waiting for the consumer, rather than actually doing something useful.

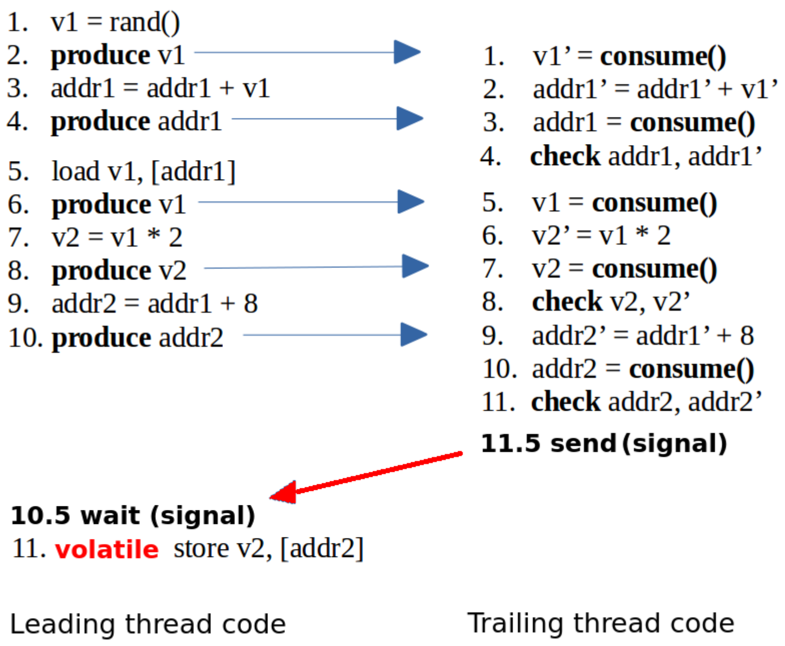


```
1.  v1 = rand()
2.  produce v1
3.  addr1 = addr1 + v1
4.  produce addr1

4.5 wait (signal)
5.  load v1, [addr1]
6.  produce v1
7.  v2 = v1 * 2
8.  produce v2
9.  addr2 = addr1 + 8
10. produce addr2

10.5 wait (signal)
11. store v2, [addr2]
```

```
1.  v1' = consume()
2.  addr1' = addr1' + v1'
3.  addr1 = consume()
4.  check addr1, addr1'
4.5 send (signal)

5.  v1 = consume()
6.  v2' = v1 * 2
7.  v2 = consume()
8.  check v2, v2'
9.  addr2' = addr1' + 8
10. addr2 = consume()
11. check addr2, addr2'
11.5 send (signal)
```

(b) Leading thread code (c) Trailing thread code

FIGURE 2.5. Synchronous Communication Pattern in RMT



```
1.  v1 = rand()
2.  produce v1
3.  addr1 = addr1 + v1
4.  produce addr1
5.  load v1, [addr1]
6.  produce v1
7.  v2 = v1 * 2
8.  produce v2
9.  addr2 = addr1 + 8
10. produce addr2
10.5 wait (signal)
11. volatile store v2, [addr2]
```

Leading thread code

```
1.  v1' = consume()
2.  addr1' = addr1' + v1'
3.  addr1 = consume()
4.  check addr1, addr1'
5.  v1 = consume()
6.  v2' = v1 * 2
7.  v2 = consume()
8.  check v2, v2'
9.  addr2' = addr1' + 8
10. addr2 = consume()
11. check addr2, addr2'
11.5 send (signal)
```

Trailing thread code

FIGURE 2.6. Semi-Synchronous Communication Pattern in RMT

Semi-Synchronous Communication Pattern

There is a third option in between waiting for the result of the trailing thread on every single memory access or just let the producer make forward process without any synchronization whatsoever, we called it *semi-synchronous communication pattern*. The basic idea is to make sure that only volatile variable accesses are correct before performing them. This is done in the literature by either synchronizing with the consumer only for those cases, or adding local comparisons in the leading thread (an ILR approach) [34] [37]. Figure 2.6 shows an example of this scheme the way [34] does it.

Such scheme reduces the inter-thread communication cost but, poses other troubles. One problem is to be able to determine volatile variable accesses and their dependencies. Such information might be available at compile time or it might be more difficult to obtain, like functions accessed by pointers. If ILR is chosen to protect volatile accesses, then the dependencies of the stores must be also replicated in the leading thread. In the case where a simple variable, already corrupted due to a soft error, is used to obtain the memory address of the volatile store, it does not help to duplicate the calculation of the memory address. Every data dependency has to be verified before performing the store. So, if the application has multiple volatile memory access points, the leading thread actually ends up replicating everything because of data dependencies, which will defeat the purpose of redundant multi-threading.

Another problem with such scheme is that since the producer thread is able to run in some cases without the check of the consumer, it might trigger exceptions. A division by zero or segmentation fault can occur because of a change in a value or an address. Such problems should not happen in a soft-error-free scenario so there might not be appropriate handlers for them in the original code; which will usually cause the program to be killed by the OS. One option, if one needs to provide recovery from this scene, is to add artificial handlers. Furthermore, they should be able to determine if the exception was due to natural causes, in which case the error should be passed to the application's original handlers (if any); or if it was due a soft error, where it should be managed differently. Another strategy could be to have the application be executed by an external program, one that can monitor and make decisions in cases where the original executable is unable to recuperate itself [34] [37]. Figure 2.7 shows an overview of the different communication patterns in Redundant Multi-Threading approaches.

The work in this thesis tries to speed up a semi-synchronous RMT scheme with several improvements and techniques. We use the scheme presented in [34] as our starting point because is the one that delivers a safe execution with acceptable performance degradation.

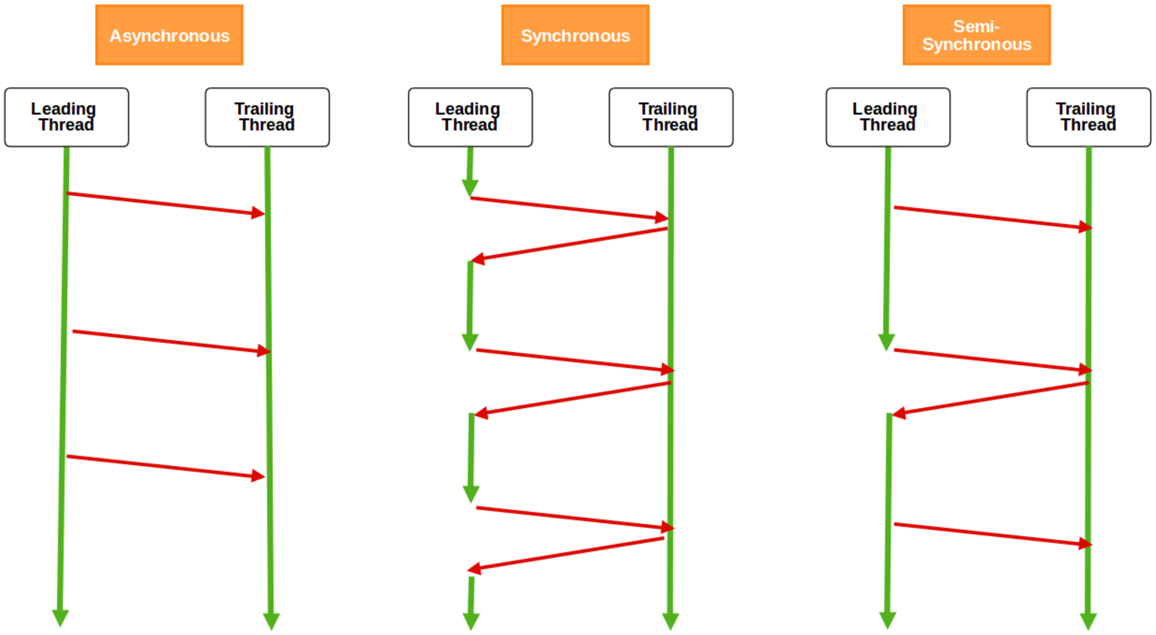


FIGURE 2.7. Redundant Multi-Threading Communication Patterns

2.4 Correcting Soft Errors

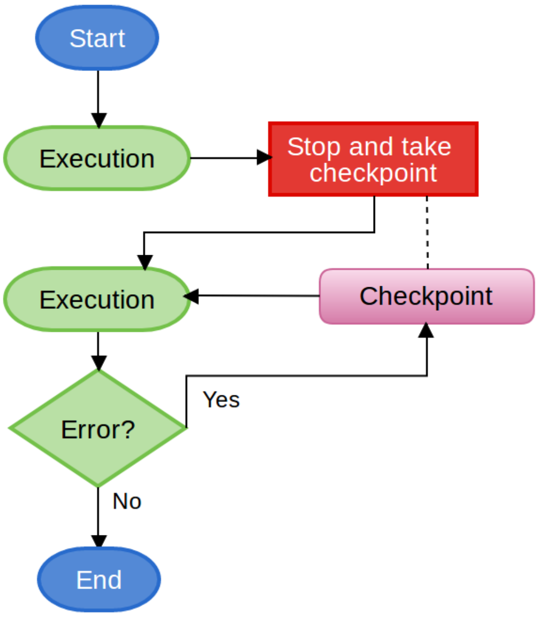


FIGURE 2.8. Checkpointing Diagram

The correction phase is the one that executes only when a soft error has been identified. Checkpointing approaches can be used in the correction phase, like mentioned in [6] [15] [23]. Figure 2.8 shows the diagram of this technique. Every certain amount of time, or instructions, the application's state (a checkpoint) is saved somewhere. The information saved should be enough so the program can restart the execution based on it. The state usually means the values of the variables at certain point in time and it is very important to make sure that every saved data is error-free. Checkpointing comes with a performance overhead, there is a penalty every time the application must be stopped in order to collect its state; depending on how much is saved each time and how many times a checkpoint is taken, the overhead varies. The benefit of checkpointing is that if an error is detected, the program is restored to the latest safe point and (hopefully) not the beginning of the execution. Since a transient fault is something very unlikely, the cost of recuperating the system in case of error is not typically a problem, but the cost of maintaining the checkpoints is what negatively impacts performance.

Another option to recover from soft errors is having Triple Modular Redundancy (TMR). Figure 2.9 shows the diagram of such technique. Two extra copies of the calculation are performed and a majority vote is used each time to decide the correct result. There is no explicit correction phase in this approach because, a soft error would be detected by having one of the three copies being different than the rest (assuming of course a Single Event Upset, SEU). And the correction phase, would be having the copy that diverges from the other two discarded and letting the program continue from there on with the other two; or the failed copy can be restored to the state of the other two. Note that this scheme has both identification and correction of soft errors, however TMR in general is a very expensive approach since it maintains two extra copies of the application [6].

These techniques have been already studied for a while and are a popular choice to correct soft errors [15] [16]. It is also not uncommon to provide only soft error detection solutions, since error correction methods can be later included [23] [34] [37]. No matter how the execution should be restored, if needed, the replication phase that allows error detection should be done efficiently.

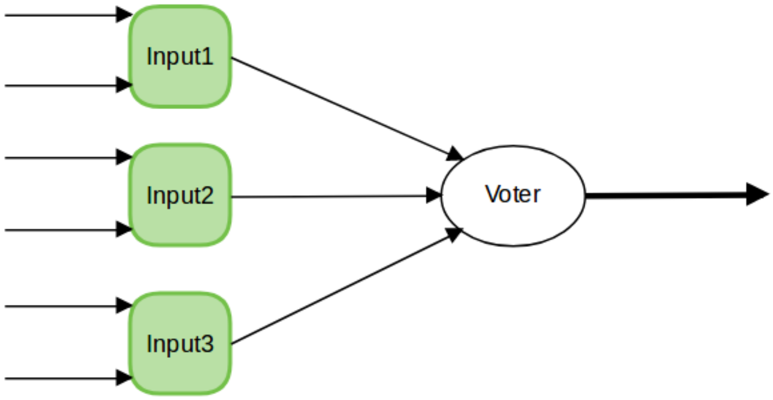


FIGURE 2.9. Triple Modular Redundancy Diagram

Because of these reasons, the thesis focus on soft error detection, specifically a redundant multi-threading approach.

2.5 Simultaneous Multi-Threading

Superscalar microprocessors implement a form of parallelism called *instruction-level-parallelism*, which allows them to execute more than one instruction during a clock cycle. It is done by concurrently dispatching several instructions to different executions units of the processor. *Simultaneous Multi-Threading* (SMT) is a technique to increase the performance of a superscalar microprocessor. It allows independent threads to better utilize the physical resources of a processor. A machine with SMT capabilities tries to allow multiple threads to execute simultaneously in the same cycle, on different functional units [27].

2.5.1 Intel Hyper-Threading

Intel's *Hyper-Threading* Technology brings the concept of simultaneous multi-threading to the Intel Architecture. Hyper-Threading makes a single physical processor appear as (at least) two logical processors. The physical execution resources are shared and the *architecture state* is duplicated for the logical processors. Each logical processor has its complete architecture state which consists of registers. Among them are the general purpose registers, the advanced programmable interrupt controller (APIC) registers, the control registers and some machine state registers; Figure 2.10 shows how a core with Hyper-Threading technology looks like. From a software perspective, since there are multiple architecture states, the one physical processor appears to be as multiple processors. This means operating systems and user programs can schedule processes or threads to logical processors as they would on multiple physical processors.

From a micro-architecture perspective, this means that instructions from both logical processors will persist and execute simultaneously on shared execution resources [20].

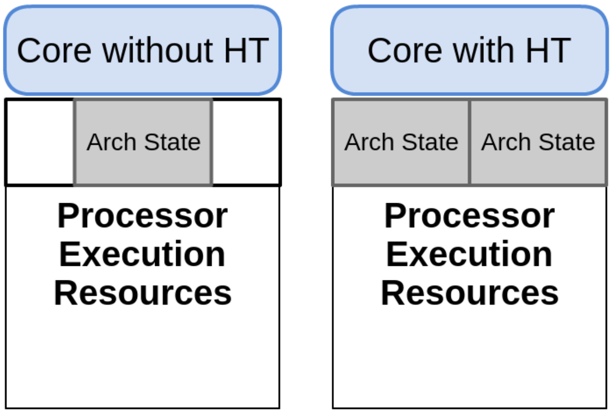


FIGURE 2.10. Core without and with Hyper-Threading

Intel in [20] claims that the number of transistors necessary to store another architecture state is an extremely small fraction of the total. But doing so favors a much more efficient resource usage, which translates to greater performance at very low cost. The logical processors nearly share all other physical resources, such as caches, execution units, control logic, branch predictors and buses.

Hyper-Threading does not do much for single thread workloads, but when multiple threads can run in parallel there may be a significant performance improvement, because it ensures that when one logical processor is stalled, the other logical processing unit (on the same core) can continue to make forward progress (without context switching). A logical processor may be temporarily stalled for a variety of reasons, including servicing cache misses, handling branch mispredictions, or waiting for the results of a previous instruction [20].

Although Intel claims that common server applications can benefit from hyper-threading, obtaining around 30% of performance improvement [20], there are cases when Hyper-Threading does not yield an improvement or even worse represents an overhead, as analyzed in [30]. Memory bound applications, in which the bottleneck is the memory latency, can benefit from Hyper-Threading. While one hyper-thread is waiting for a memory access, the other one can utilize the execution units, leading to a better resource usage. On the other hand, for CPU-intensive programs that tend to keep the execution units busy, having to share such resources with another thread will not represent an enhancement.

The use of hyper-threading in the detection phase of soft errors can be of a lot of help. Since in redundant multi-threading approaches the bottleneck is inter-thread communication, the fact that hyper-threads share the L1 level cache can be exploited. Instead of having to send data from one core to another, usually through the last level of cache (or even worse, through main memory), exchanging values using L1 cache can benefit performance significantly. Figure 2.11 exemplifies this situation.

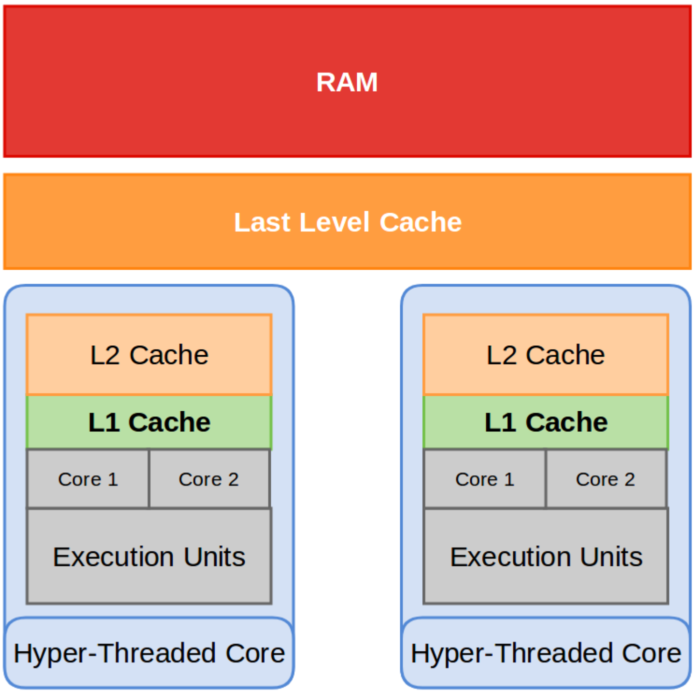


FIGURE 2.11. Diagram of Hyper-Threaded Processor

Chapter 3

Related Work

In this chapter the related work is presented. Selective Replication, a branch in soft error detection is first briefly introduced. Current solutions for soft error management are discussed, their strengths and their weaknesses are analyzed. Multiple schemes of soft error detection via software are discussed, in order to establish the current thesis approach. Finally the chapter ends discussing the proposed ways to improve Redundant Multi-Threading approach.

As it was mentioned before, there are many ways to accomplish soft error detection via hardware, but they all require specialized hardware, which is out of the scope of the current thesis. As an alternative, different software-only fault-tolerance mechanisms exist. Since common hardware is used, they need to modify the program, either after compilation by binary instrumentation or during compilation. These solutions, are categorized into Instruction Level Redundancy (thread-local) and Redundant Multi-Threading error detection. In the first category, redundancy is added in the same thread and in the latter replication is distributed in two threads.

3.1 Selective Replication

One important aspect about bit flips it that they do not necessarily lead to a system failure. For that, many authors have explored the possibility of protecting just the parts of the application that when hit by a particle strike, may result in data corruption. Protecting only areas of the program susceptible to soft errors is usually referred to Selective Replication. Calhoun *et al* in [6] try to determine the way a soft error propagates thought different kinds of HPC applications. They say that in order to understand what parts of the applications needs protection, it is first important to know how a soft error spreads to contaminate more data or instructions. The authors perform their tests with MPI applications and analyze fault propagation at two levels. The first one is rank local, how the soft error modifies local values in the same process where it happened; the second one is among ranks, the way other process are contaminated when inter-process communication occurs.

Shoestring [8] accomplishes fault tolerance based on two areas of prior research: symptom-based fault detection and redundancy through ILR. The former one relies on the fact that applications in presence of transient faults often exhibit abnormal behavior. Examples of these symptoms are branch mispredictions or memory access exceptions. They say that symptom-based detection is inexpensive but the amount of coverage is typically limited. For that they take advantage of ILR. Which can get to nearly 100% of fault coverage, however replicating the application incurs a lot of performance overhead. The authors perform compiler analysis to identify the vulnerable instructions that symptom-based fault detection fails to detect and protect them with redundancy. One downside of the solution is that the algorithm to identify such instructions requires hardware specific knowledge, for example knowledge about exception-throwing instructions from the instruction set architecture [17].

IPAS [17] also attempts to protect only the code that actually needs coverage. The authors use a machine learning process in order to identify the instructions that require duplication. They claim that protecting the whole application wastes a lot of resources due to the low probability of a soft error actually modifying important data. And even in such cases, it may not be catastrophic because in most HPC applications a soft error can be masked due to the algorithm properties (meaning that the result is acceptable even in the presence of an error). With a well-trained model, they manage to get low slowdown achieving high coverage percentage against data corruption. One downside on the solution is the time it takes the machine learning process to produce accurate results, which is considerable slower than a compiler based approach like Shoestring [8].

Regardless of what needs to be protected in the application, the replication phase must be efficient. For that, the next two sections explain further replications mechanisms.

3.2 Instruction Level Redundancy

A well known thread-local replication compiler-based approach is SWIFT [13], where instructions are duplicated within the same thread and periodic checks of both results are added, ensuring the correct execution of single-threaded applications. The resulting code needs two times as many registers as the original code, which can potentially cause register spills (using RAM when there is no more space available in the processor). For that, in architectures such as Itanium with many registers, the overhead of SWIFT is low [37]. This scheme provides only fault detection, though a common checkpointing technique can be integrated with SWIFT for full soft-error protection.

HAFT (Hardware Assisted Fault Tolerance) provides full soft error management. It uses

ILR for fault detection and Intel's version of *Hardware Transactional Memory*, Transactional Synchronization Extensions (TSX), for fault correction [15]. Hardware Transactional Memory provides a mutual exclusion mechanism, in which there is a way to create an atomic block of instructions, a transaction. When executing the transaction, modified data is kept temporarily in the core's cache. If no memory conflict (read/write, write/write) is detected with another concurrent transaction, every instruction in the block is atomically committed to RAM; so every other core can view the results. On the other hand, when a memory collision is identified, all modified data stored in the cache is reverted and no single operation of the transaction is committed to RAM [12].

Intel's TSX provides a very convenient way to explicitly force a transaction abort and therefore restore the state of the core to the beginning of the transaction. Such mechanism is exploited in HAFT for soft error recovery purposes [15]. After the code has been duplicated with ILR, the application is wrapped in HTM-based transactions in order provide recovery. When an error is detected by the ILR checks, the transaction is explicitly aborted, the state of the application is restored before the transaction began and the execution is retried.

Sadly, since Intel TSX was not thought to be a soft error recovery mechanism, a transaction may fail due to several (sometimes unexpected) reasons. Hence HAFT presents a best effort approach, where a transaction is retried a fixed number of times. If all such attempts fail, that part of the application runs again without HTM. That means, if an error occurs in this unprotected moment of execution, ILR has no choice but to permanently abort the program. Mostly this design choice in HAFT is driven by the restrictions that Intel TSX currently exhibits [15].

Another reference where HTM is used as a recovery mechanism is Fault Tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support [11]. It is an approach which leverages Intel TSX to support implicit checkpoint creation and fast rollback. The authors combine a software-based redundant execution for detecting faults with hardware transactional memory to restore the state of the application if necessary. The main idea of the paper is to redundantly execute each user process and to instrument signature-based comparison on function level. They prefer processes instead of redundant threads because the virtual memory management of the operating system guarantees physical memory isolation and therefore if one error occurs in one process it is less likely that will propagate to its duplicate. This is true for local memory to the process, but if the error happens with a non-volatile memory access (which can execute some I/O operation), the fact that they use processes instead of threads does not help at all. This proposal resembles a lot redundant multi-threading approaches, but using processes. Generally speaking, on each function boundary, a signature is created (using values of variables) which identifies the block of code. It is then shared to the duplicated process so it can be compared. Both signatures should match and if they don't, the recovery mechanism is initiated [11].

They rely on Intel's TSX for recovery purposes and so the duplicated process (process 2) has hardware memory transactions. The flow of the technique goes like this: the main process executes the first function of the program, it creates the signature, shares it with its duplicate and keeps running. When the process two receives the signature, it starts a transaction, executes the function and at the end compares the two signatures, if it detects a mismatch it commences the recovery. This asynchronous scheme is due that signature exchange within a transaction always results in an abort, due to conflicting memory accesses. Since only the duplicated process has TSX, the main one continues executing the application and consequently is a couple of functions ahead of the other one. If an error is encountered, the duplicated process aborts its transaction, but the main one is already some steps ahead and cannot be rolled back to the desired point; for that, the authors decide to kill the main process and fork a new one. It is unclear how they authors deal with volatile memory accesses, because given the scheme the leading process is allowed to execute unsafe operations without a check of correctness. While it is true that the trailing thread will later detect if something has gone wrong, the first process might have already execute an irreversible situation [11].

3.3 Redundant Multi-Treading

An example of redundant multi-threading with asynchronous communication pattern is COMET (Communication-Optimised Multi-threaded Error-detection Technique) [23]. Mitropoulou *et al* first identify that the performance overhead of most redundant multi-threading techniques lies in poorly executed inter-thread communication. Since the two threads need to exchange data frequently, if this is naively implemented it can incur significant performance overhead. They propose several code optimizations on the generated code in order to alleviate the problem, but mostly they rely on the Lynx queue [24]. The Lynx queue is a *Multi Section* Single Producer/Single Consumer (SPSC) with fast enqueue/dequeue operations. Such queues are divided into sections and only one thread is allowed to access a section at a time. The sections have a state indicating who is using it: the producer by writing values or the consumer reading already produced fields. The synchronization of threads happens only at section boundaries, the consumer cannot start reading from a section that has not been totally written; and the producer cannot start filling a section that has not been entirely read (except the first time). This separation allows that both threads access the queue simultaneously without locking each other, provided they work on different sections. A multi-section queue tries to solve problems such as cache ping-pong and false sharing [24].

The main novelty of Lynx queue is using just two instructions per enqueue and dequeue operations; that is the read/write of the data and advancing the dequeue/enqueue pointer. This is performed by taking advantage of memory protection systems available in commodity processors and operating systems. Each section of the queue is followed by a protected memory zone, non-readable and non-writable, which are called red zones. This red zones serve as the synchronization mechanism triggers. Every time the enqueue/dequeue pointer reaches a red zone at the end of the section, it will access protected memory. Therefore, a segmentation fault signal is raised which is managed by a custom exception handler where the synchronization takes place. By doing this trick, they remove the synchronization cost from the critical path of execution and reduce the overall number of instructions executed [23] [24].

The authors from COMET actually personalize the Lynx queue in order to make it more efficient for the redundant multi-threading case. They sacrifice general use of the modified queue for performance gains. The error detection code generation is performed automatically as a RTL [1] pass in the back-end of GCC.

While the Lynx Queue helps a lot in the inter-thread communication, the fact that is a multi-section queue means that the communication pattern must be asynchronous, and so the producer never awaits for the consumer check. This is done in this case to prevent deadlocks. If the producer has to pause for the consumer but is in the middle of filling its section, it will wait forever, since the consumer cannot start reading values of such section until it is completely filled. Having this scenario means that there is no protection against volatile memory accesses. In fact, the authors from [23] do recognize that such operations are protected in the state of the art solutions, but then they do not mention mention how this problem is solved in their paper.

Decoupled Acyclic Fault Tolerance (DAFT) [37] is another example of the redundant multi-threading schemes, where the threads are scheduled to different cores. The authors present a non-speculative version with synchronous communication pattern. Before any memory operation in the leading thread, the address and value are sent to the trailing thread, compared against the corresponding duplicate values and then a check is sent back to the leading thread. While all this happens, the leading thread has to wait (by spinning) for such check before continuing. In this version of DAFT, there is a lot of performance overhead because of the inter-thread communication.

The same authors then present a speculative version with a semi-synchronous communication pattern. Basically, the idea is to speculate that the operations performed by the leading thread are without-errors and therefore allowing it to continue freely as much as possible. That permits the leading thread to advance on some instructions without a proof of correctness, removing the busy waiting from the critical path of the application and also reducing communication bandwidth. Zhang *et al* (authors from DAFT) point out that volatile variable accesses still need some kind of check before execution, in order to avoid catastrophic effects. So, they implement ILR in the leading thread only for volatile memory accesses and their dependencies. For any other operation, the leading thread is allowed to make forward progress, producing values that will be later checked by the consumer, but not having to wait for them [37].

However, the approach comes with the penalty of possibly having an incorrect execution of the program due to misspeculation. Since the leading thread is allowed to continue on some operations without a check, it might be unaware that a transient fault has already corrupted

[1] RTL stands for Register Transfer Language. The last part of the GCC compiler work is done in this low-level intermediate representation [1]

important data. The authors try to solve this issue by injecting artificial handlers in the code, so in case an exception is raised by the leading thread on operations that were speculated to be correct, they can be properly managed. There is another downside of the solution, the fact that there might be several dependencies for volatile memory accesses, that could depend as well on other values. That situation will cause the leading thread to have ILR in a lot of its code; which would defeat the whole purpose of having a redundant multi-threading approach, this has already been discussed on Section 2.3.2.2. By performing such mechanisms and other optimizations, DAFT manages to still be deliver a correct execution of the program and decrease performance overhead significantly compared to their non-speculative version [37].

Another Redundant Multi-Threading approach is presented in *Compiler-managed software-based redundant multi-threading for transient fault detection* [34]. As well as DAFT, they analyzed the high cost of having a synchronous communication pattern and conclude it involves too much performance overhead to be a manageable solution. The authors, also identify that volatile memory accesses must be confirmed correct before the leading thread can execute them. In order to detect such operations, they rely on variable attributes available to the compiler and only in those cases the leading thread awaits for the trailing thread's confirmation. Basically, they propose another semi-synchronous communication pattern that only synchronizes with the trailing thread on volatile memory accesses. Sadly, they ran into the same problems DAFT does when allowing the leading thread to continue on some operations without a check. Sometimes exceptions might be triggered due to soft errors and the authors decide (as well as in DAFT) to install artificial handlers on the application in order to deal with those signals.

Both DAFT and *Compiler-managed software-based redundant multi-threading for transient fault detection*, propose a Redundant Multi-Threading approach with semi-synchronous communication pattern. The difference between them is how they deal with memory volatile accesses. The latter option, makes the leading thread wait for a confirmation from the trailing thread in such cases. On the other hand, DAFT decides to implement ILR in the leading thread instead of having to synchronize both threads. Both options run into problems when they choose to allow the leading thread continue on some operations without a proof of correctness.

COMET [23] focuses on optimizing the inter-thread communication in a redundant multi-threading approach with asynchronous communication pattern. They compare their results against the state of the art solutions, or so they claim, and get a higher performance on average by reducing the number of instructions executed. However, they actually perform the tests against a technique similar to DAFT [37] and *Compiler-managed software-based redundant multi-threading for transient fault detection* [34], which both are semi-synchronous RMT solutions and therefore provide protection for volatile memory accesses. But, since COMET does not protect against volatile variable accesses because of its asynchronous communication, it seems that is not a fair comparison.

Sadly, the optimizations described in [23, 37] are not applicable in our HPC context. In DAFT

[37], the authors propose to move to an asynchronous communication pattern by protecting the *volatile* variables and the variables they depend on using ILR instead of synchronizing the threads. However, there are many volatile variables in HPC application, and thus, applying such a solution in the HPC context would probably lead to fall back to full ILR. The authors of COMET [23] propose to use a very efficient SPSC queue to improve RMT performance. This SPSC queue algorithm divides the queue into multiple sections that are processed as batches of messages to reduce as much as possible the number of instructions required to insert items in and remove items from the queue. However, such a batching algorithm implies that an asynchronous communication model has to be used between the *leading* and the *trailing* thread[2]. Since the number of volatile variables is large in the applications we target, using such an asynchronous communication pattern would not be safe.

3.4 Improving Redundant Multi-Threading

The main factor that limits the performance of existing RMT solutions is the cost of communicating between threads to compare the output of their execution [23]. In software-based RMT, a *leading* thread has to send the result of each operation it executes to its sibling *trailing* thread that detects soft errors by comparing the values it receives with the result of its own operations. The single-producer/single-consumer (SPSC) queue involved in this communication is central to the performance of RMT. The first improvement we propose is to run sibling replicas on SMT threads of the same physical core, to allow for more efficient communication between them.

The second direction we study to improve the performance of RMT is to limit the amount of data that is exchanged between the leading and the trailing thread. To achieve this goal we first propose *variable aggregation*, that is, to combine several values produced by the leading thread into a single value and to send this single value to the trailing thread (this thread also groups the same variables together, so there is no mismatch). The second technique we evaluate is *selective checking*, that aims at reducing the number of variables that are checked for soft-error detection. More specifically, in this thesis, we evaluate the performance of a solution that only checks variables that correspond to communication with the *outside world*.

3.4.1 Using SMT for Soft Error Detection

The use of Hyper-Threading in Soft Error detection techniques in general has, to our extent, not been investigated properly. There are however, some papers published a while ago (2000 and 2002) that propose hardware solutions leveraging from a Simultaneous Multi-Threading (SMT) Processor. *Transient fault detection via simultaneous multithreading* [27], suggest how a SMT processor can be tweaked and improved, in order to be able to create a Simultaneous and

[2] Using batches with a *semi-synchronous* communication pattern could lead to a deadlock in the case where the *leading* thread is waiting for an acknowledgment from the *trailing* thread while the *trailing* thread is waiting for the current batch to be full to start processing data.

Redundantly Threaded (SRT) processor. A SRT processor would be able to efficiently execute an application with redundancy in order to detect soft errors.

Transient-fault recovery using simultaneous multithreading [33], continues the work done by Reinhardt *et al* [27] to extend a SRT processor to a Simultaneously and Redundantly Threaded processor with Recovery (SRTR). The authors propose how would be best to add error recovery to a STR processor. Both solutions are hardware-based, which are out of the scope of the current thesis.

We aim at leveraging SMT for soft-error detection as proposed in the seminal paper by Reinhardt and Mukherjee [27]. However, contrary to the solution described in [27], our solution is solely implemented at the software level. Making use of wasted CPU cycles through SMT threads can allow for efficient RMT-based SDC detection.

Specifically on Redundant Multi-Threading approaches, Hyper-Threading has neither been investigated appropriately. The only mention we know of is in *Compiler-managed software-based redundant multi-threading for transient fault detection* [34], in a single experiment with their semi-synchronous communication pattern. They authors compare hyper-threads that share the L1 cache (config 1), against threads on different cores within the same cluster that share a L4 cache (config 2) and threads on different clusters with different L4 caches (config 3). The physical configuration of their test machine made the second configuration perform best, the configuration 1 came in second place (but not by much) and the configuration 3 behaving much worse than the other two. It is important to notice that such paper was published in the beginning of 2007, more than 10 years ago. For that, and other reasons explained in Section 4.2, we believe that a more detailed investigation of Hyper-Threading in Redundant Multi-Threading soft-error detection techniques is necessary.

3.4.2 SPSC queues

As discussed previously, the SPSC queue used to communicate between the *leading* and the *trailing* thread is central to the performance of RMT. The SPSC queue problem has largely been studied even outside the context of RMT [10, 18]. The key point to achieve high performance with an SPSC queue is to reduce the number of control instructions that a thread should execute during an enqueue or a dequeue operation [24].

Batching has been proposed to improve the performance of SPSC queues [19, 24]. The main goal of batching is to allow the threads to run a sequence of operations (enqueue or dequeue) corresponding to the size of one batch with a reduced number of control instructions. As discussed previously, the use of fixed-size batches as in [24], is not suitable for a semi-synchronous communication pattern between the producer and the consumer as required by our RMT approach to handle stores to volatiles variables, as it would be prone to deadlocks.

The solution proposed in by Wang et al. [34] relies on flexible batching. More specifically, their main ideas are *Delayed Buffering* and *Lazy Synchronization*. As such, we name this queue, the

DB-LS queue. The producer batches a few values before updating the index that is visible to the consumer. This allows improving performance as it reduces the number of cache line transfers between the two threads. Such a solution is still able to deal with volatile stores because in this case the producer can make a batch visible before it is full. We present the *DB-LS* queue in more details in Section 5.1.

In the rest of this work, we use the *DB-LS* queue as a basic block for RMT, because according to results reported on the paper, is the most efficient existing solution for implementing a semi-synchronous communication pattern within our context of HPC applications.

3.4.3 Reducing the scope of soft-error detection

One of the contributions of this thesis is to study how much *selective checking*, that is, limiting the number of values in a program that are checked for soft-error detection, can improve the performance of RMT.

This idea has already been proposed in the context of ILR [17, 36]. These papers propose different techniques to identify the main variables of a program that need to be protected for a program to be correct. The work presented in [21] aims at achieving the same goal. These works are complementary with the contribution we present: In this paper, we study the impact of *selective checking* by applying a *simple* heuristic, that is, applying soft-error detection only to the store instructions to volatile variables. In the future, we could try to apply more advanced techniques to decide which variables to protect, such as the ones introduced in these works.

Chapter 4

Research Description

In this chapter the research description is presented. The problem definition, proposed solution, justification, hypothesis, objectives, scope and limitation, and testing methodology are explained.

4.1 Problem Definition

Redundant Multi-Threading is a replication-based technique to detect soft errors. The original application code is duplicated in two threads, the producer or leading thread and the consumer or trailing thread. Both threads execute almost the same code and every time a value needs to be checked, there is data exchange from the producer to the consumer and or vice versa. The latter thread, performs additional checks in order to compare both values and be sure that no soft error has happened. In such schemes, communication between threads happens frequently and that represents the main performance bottleneck of such solutions.

There are 3 types of communication patterns in RMT solutions, synchronous, asynchronous and semi-synchronous. In the first one, the leading thread must wait for the trailing thread confirmation on every memory operation, making the performance overhead unmanageable. RMT schemes with asynchronous communication patterns try to minimize the data exchange between the two threads. They let the producer make forward progress all the time without a check from the consumer. When there are no volatile memory accesses such schemes provides enough soft-error-detection with low performance overhead [23]. However on the other hand, when there are volatile memory accesses, an asynchronous communication pattern does not offer sufficient protection. RMT approaches with semi-synchronous communication patterns take into account volatile memory accesses and still try to provide low inter-thread data exchanging. The basic idea is to allow the leading thread continue without a proof of correctness as much as possible, while delivering safe execution of volatile memory access.

Although there are several variations of a Redundant Multi-Threading soft error detection technique, in general they incurs a lot of performance overhead because of the highly frequent inter-thread communication.

4.2 Justification

4.2.1 Volatile Variable Accesses

Several authors [23] [34] [37] claim that applications have few volatile variable accesses and that RMT with synchronous communication pattern is unrealistic because it adds so much performance overhead. In COMET the authors even decide not to protect at all against such instructions. But we have found that volatile variable accesses are not too uncommon. Many real HPC applications use programming models like MPI or OpenMP, where data needs to be transfered between nodes or threads regularly. Each one of these function calls includes at least one volatile memory access that should be confirmed to be correct before execution. As an example of this, there are several programs included in the Mantevo benchmark project [1]. The Mantevo project has a set of mini applications (kernels) that are representative in the HPC programs. These small software examples perform the most common work that HPC real software do as well [13]. For example, the HPCCG mini app is described as follows:

"*Many engineering applications require the implicit solution of a nonlinear system of equations where the vast majority of time as problem size increases is spent in some variation of a conjugate gradient solver. As a result, any miniapp focusing on this area will necessarily have a conjugate gradient solver as the dominant computational kernel. MiniFE or HPCCG is a miniapp that mimics the finite element generation, assembly and solution for an unstructured grid problem*"

Such software does represent a lot of engineering HPC applications. In this case the problems is solved with an iterative algorithm that approximates the solution. If the application is executed with MPI, on every round of the loop that data is refined there is a send/receive instruction to neighbors nodes and every one of these communication instructions represents at least one volatile store. On the other hand, if it is run in a single machine with multiple threads, on every iteration there is also communication among threads; which again are volatile stores.

Another example is CoMD or MiniMD: a molecular dynamics simulation, also a miniapp from the Mantevo project. It is explained as follows:

"*The MiniMD application is miniature version of the molecular dynamics (MD) application LAMMPS. The source for MiniMD is less than 3,000 lines of C++ code. Like LAMMPS, MiniMD uses spatial decomposition MD, where individual processors in a cluster own subsets of the*

[1] https://mantevo.org/

simulation box. And like LAMMPS, MiniMD enables users to specify a problem size, atom density, temperature, timestep size, number of timesteps to perform, and particle interaction cutoff distance."

CoMD is an MPI applications with a lot of communication between particles through MPI function calls, as stated before each one of this calls is a volatile memory access point which has to be check for correctness before execution.

Such mini applications serve as an example to demonstrate that volatile variable accesses are not that uncommon in HPC software and therefore any Redundant Multi-Threading mechanism should be aware of them. Therefore we believe that a safe approach is to have a solution with semi-synchronous communication pattern, that on every volatile memory access must allow the trailing thread to catch up with the leading one, in order to provide confirmation that no soft error has happened; similar to the solution provided by [34]. As said before, this scheme also has high inter-thread communication costs, which we believe can be mitigated with the use of Hyper-Threading. Since hyper-threads share the cache L1, data communication between them is faster that having the threads on different cores. Our other two improvement proposals can be of great help reducing the overall performance overhead.

4.2.2 Redundant Multi-Threading with Hyper-Threads

We only know of one experiment that has tested how hyper-threads behave in a Redundant Multi-Threading technique[34]. In such tests, further explained in Section 3.4.1, the hyper-thread option came in really close, even better in some cases, to the best configuration (having two threads in the same cluster sharing the L4 cache). Also the last option of two threads in different clusters not sharing a L4 cache behave significantly worse than the two other options. We believe the following reasons justify a more detailed investigation of Hyper-Threading in Redundant Multi-Threading approaches:

- Such experiments were performed a long time ago (more than 10 years ago), in which some implementation details of Hyper-Threading may have been improved by Intel.

- The specific physical characteristics of the test machine cannot be generalized to all supercomputers.

- The technique tested was not specifically designed to run on hyper-threads, therefore there might be some modifications that might decrease the overhead.

- The overall time performance of the technique with hyper-threads does not necessarily has to be better, compared against threads on different cores, to justify its use. Hyper-Threads run on a single core, therefore if a hyper-threaded version of the technique performs similar to threads on different cores, means that the resource utilization overhead of the system can be significantly improved.

4.3 Solution

Some HPC applications cannot fully take advantage of Hyper-Threading for performance improvement due to algorithmic properties; different data access patterns make up for different performance gains. Whether the original application can benefit from HT or not, we propose to use these additional resources for fault tolerance instead.

Second, we present variable aggregation to group several values together and use this merged value to speed up detection of soft errors. Instead of sending every single data that needs checking through the SPSC queue, we can accumulate multiple values into a single variable. Therefore reducing the amount of traffic that goes through the queue, while still being able to detect soft errors in any of the values.

Third, we introduce selective checking to decrease the number of checked values to a minimum by manually identify the volatile memory access points of the application where checks are strictly necessary. The last two techniques reduce the overall performance overhead by relaxing the soft error detection scope. By doing this the overhead can be reduced significantly.

4.4 Hypothesis

Applying the proposed improvements to a Redundant Multi-Threading detection technique with semi-synchronous communication pattern, shows that it can be a viable solution (less than 2x overhead) for soft-error detection at extreme scale.

4.5 Objectives

4.5.1 General Objective

The main objective of this thesis is to improve a Redundant Multi-Threading technique with semi-synchronous communication pattern via the 3 optimizations proposed, to show than it can be a viable solution (less than 2x) for soft error detection at extreme scale.

4.5.2 Specific Objectives

1. Design and implement the way Hyper-Threading can improve a RMT with semi-synchronous communication pattern approach.

2. Design and implement variable aggregation improvement for a RMT with semi-synchronous communication pattern approach.

3. Design and implement selective checking improvement for a RMT with semi-synchronous communication pattern approach.

4. Evaluate the performance improvement of each of the optimizations proposed.

4.6 Scope and Limitations

The current thesis focus only soft error detection, specifically on a Redundant Multi-Threading approach. Since checkpointing techniques have been commonly used for soft error correction, we believe our scheme can easily be further combined with such a technique in order to provide complete soft error management.

4.6.1 Restrictions and Assumptions

It is assumed main memory is already protected by mechanisms like ECC against bit flips. Also, only one of these faults is expected to happen per execution of the application (a Single Event Upset, SEU). Both assumptions are commonly made in the literature [6] [15] [23] [34] [37].

4.7 Test Methodology

The final software and experiments will be run in different machines on the Grid'5000. Grid'5000 is a large-scale and versatile testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing including Cloud, HPC and Big Data. Next are some features about Grid'5000: [2]

- provides access to a large amount of resources: 1000 nodes, 8000 cores, grouped in homogeneous clusters, and featuring various technologies: 10G Ethernet, Infiniband, GPUs, Xeon PHI.
- highly reconfigurable and controllable: researchers can experiment with a fully customized software stack thanks to bare-metal deployment features, and can isolate their experiment at the networking layer
- advanced monitoring and measurement features for traces collection of networking and power consumption, providing a deep understanding of experiments
- designed to support Open Science and reproducible research, with full traceability of infrastructure and software changes on the testbed

In order to try to provide HPC representative results with our experiments, we will be using applications part of the Mantevo Project (already mentioned in Section 4.2). Such applications are small examples of real HPC software. For each application there in an execution that we call the *baseline*, which represent the program executed without soft-error detection. Then the replicated version will have 3 versions, no optimization, variable aggregation and selective checking. Needless to say that in the last two our proposed improvements will be applied. Finally,

[2] https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home

for each version there will be two sub-versions where a core-threaded-version, in which the leading and trailing thread will be pinned to different cores. Lastly, a hyper-threaded-version, in which both threads will be pinned to the same core, hence exploiting the hyper-threading capabilities available in the chip.

Chapter 5

Our solution

In this chapter we present the RMT SPSC queue algorithm used as our baseline. Secondly, we explain the way we modify applications to deliver soft error detection and finally present the details of each optimization we propose to improve the RMT performance overhead.

5.1 RMT synchronization

This section presents the RMT synchronization technique we use as a basis for our evaluation. We start by presenting the *DB-LS* queue. We describe the way different types of variables are protected. Finally we present an example of how we modify programs for soft-error detection.

5.1.1 Thread synchronization mechanism

The real difference in SPSC queues is how the FIFO policy is implemented. Usually it is accomplished at the level of the enqueue and dequeue functions, making sure everything is correct before actually executing the operation. In their simplest form, the *Produce* function waits until the next entry of the queue (accessed via enqueue pointer) is empty, writes it with the new value and advances the enqueue pointer one position. The *Consume_Check* method, on the other hand, holds until its next entry in the queue (accessed via deuque pointer) is not empty, reads the value from it and compares it against the parameter, then it moves the dequeue pointer; if the values differ it should report a soft error. Sadly, this is costly because it implies a lot of standing by for the other thread to reach a certain state, before actually writing or reading the queue. So, the waiting implementation determines the overall performance of the queue.

As discussed in Section 3.4.2, we use the *DB-LS* queue [34] to implement the communication between replicas in our RMT technique because of its efficiency and that is applicable to our HPC

```
method dbls_enqueue(data)
{
    content[enqLocal]=data
    enqLocal = (enqLocal+1) % QSIZE

    if(enqLocal % UNIT==0)
    {
        while (enqLocal == deqCached)
            deqCached = deqIndex

    enqIndex = enqLocal
    }

}
```

(a) Enqueue operation

```
method dbls_dequeue()
{
    if(deqLocal % UNIT==0)
    {
        deqIndex = deqLocal

        while(deqLocal == enqCached)
            enqCached = enqIndex
    }

    data= content[deqLocal]
    deqLocal = (deqLocal+1)  % QSIZE
    return data
}
```

(b) Dequeue operations

Figure 5.1: DB-LS queue synchronization mechanism

context. We briefly introduce the *DB-LS* queue algorithm.

The *DB-LS* queue is based on a circular buffer and uses 3 different variables on each thread. On the *leading* thread for example the variables are:

- *enqLocal* stores the index of the last element pushed to the queue. This is a local variable not shared between threads.
- *enqIndex* is the shared index to the last element pushed to the queue, which is only updated when enough data has been accumulated.
- *deqCached*, is a local index corresponding to the last value of the shared *deqIndex* read by the *leading* thread.

The *trailing* thread uses 3 variables in a similar way (*deqLocal*, *deqIndex* and *enqCached*). Figure 5.1 shows the implementation of the enqueue/dequeue operations. The optimization lies in two key aspects, delayed buffering and lazy synchronization. Delayed buffering means that data are buffered on the producer with the help of a local index and only when enough values (`UNIT` operations) have been enqueued, the data are made visible to the other thread by updating the shared variable *enqIndex* (the same logic is applied to dequeue operations). Lazy synchronization means that the algorithm avoids checking directly shared variables on each enqueue/dequeue operation, but iterates based on local indexes. A deeper explanation is provided in [34].

5.1.2 Interface of our technique

Figure 5.2 presents the interface of the queue used for communication between the leading and the trailing threads. We omit details about cache alignment, queue size and optimization-specific

```
1  typedef struct queue{
2     Type content[QSIZE]
3      // Leading thread values
4      int enqIndex, enqLocal, deqCached
5      // Trailing thread values
6      int deqIndex, deqLocal, enqCached
7      bool volState
8      Type volValue
9  }
10 method Produce(Type val)
11 method Consume_Check(Type val)
12 method Produce_Volatile(Type val)
13 method Consume_Volatile(Type val)
14 method Produce_Direct(Type val)
15 method Type Consume_Direct()
```

Figure 5.2: Interface of the SPSC queue

variables for the sake of clarity. The array *content* is the circular buffer used to store the data inserted in the queue. The queue has size `QSIZE` and manipulates objects of type `Type`. In practice a different queue can be instantiated for each basic data type used by the application. The variables on lines 4 and 6 are used to implement the *DB-LS* queue algorithm. Finally, the last variables (*volState* and *volValue*) are used during a synchronous communication before a store to a volatile variable.

The interface of the queue to communicate between the leading thread and the trailing thread includes 6 methods. The first two methods are used for asynchronous communication, that is, when the the leading pushes a value and does not need an acknowledgment from the trailing thread before resuming its execution: *Produce()* pushes a new value to the queue and *Consume_Check()* reads the next value from the queue and checks for a soft error by comparing with the value computed by the trailing thread. The next two functions are the special cases of enqueue and dequeue operations for values used for store instructions to volatile variableq, *i.e.*, they implement the synchronous communication. *Produce_Volatile()* enqueues a value and waits for an acknowledgment from the trailing thread before returning. *Consume_Volatile()* is the corresponding dequeue function that informs the leading thread when the data has been consumed. Finally, *Produce_Direct()* procedure is used to sent data from the main thread to its replica without any soft-error verification. Its counterpart is *Consume_Direct()*. These methods are used to send to the *trailing* thread, the result of volatile memory operations that return a value and that are only executed by the *leading* thread. The use of all these methods is illustrated through an example in Section 5.1.4.

5.1.3 Synchronous communication for volatile store operations

Figure 5.1 explains how asynchronous communication is implemented between the threads but, as mentioned before, stores to volatile variables are a special case that requires synchronous communication. The leading thread must wait for an acknowledgment from its replica, because it is about to execute something that may have irremediable effects.

Each synchronous exchange makes the producer spin-wait for the consumer's acknowledgment allowing it to continue with the operation. Since the former might have already enqueued several "normal" values, it has to hold until all of them are verified for correctness. Therefore, a *volatile store* implies that the whole execution so far is checked for integrity.

We recall that these writes to volatile variables should not be replicated, *e.g.*, we would not want to print twice a value to the console, or send data twice to a neighbor node. We must only make sure that any data used as parameters to these function calls are free of soft errors, before the leading thread executes them. Other RMT solutions also skip replicating these instructions [15, 23, 34, 37].

There is a risk of deadlock when checking a *volatile store*, because the *DB-LS* queue is used for asynchronous communication. The leading thread allows its replica to know that new data is in the queue only after every `UNIT` operations have been executed. Before that, the index *enqIndex* is not updated and therefore, the trailing thread is unaware of such data. If the consumer catches up to the point where it last saw the producer (*enqCached*) and the latter has not updated *enqIndex* ever since, then the replica will spin for a different *enqIndex* value. But, if there is a store to a volatile variable to be checked before the producer reaches a new `UNIT` limit, then it will also spin-wait for the replica to validate all recent entries. Sadly the consumer will not be able to validate new data, because the producer did not update the *enqIndex* variable.

It is unclear in [34] how Wang *et al.* deal with this scenario. So, we must be careful in order to have a deadlock-free synchronization mechanism while using the *DB-LS* queue. To avoid the system to hang, before every *volatile store* each thread calculates how many steps are needed to reach a new `UNIT` limit (we call this number S). Then, we logically advance both threads S steps ahead (by updating the indexes). This makes the leading thread publish all local new data and update its real position. Both threads reach the same point and skips the same number S of operations, thus avoiding any deadlock. Figure 5.3 shows the implementation of the synchronization before a store to a volatile variable. Note that the variables *volValue* and *volState* defined in Figure 5.2 are used for the synchronous communication.

It is common in programs that several store operations considered as volatile are done in the same code portion. For instance, it can be the case when an array or several variables are used as input to a library function call such as an MPI library. We note that in those cases only the last produced value needs to be treated as volatile, because checking a *volatile store* will imply that everything else so far is also validated.

5.1.4 Code transformation

We illustrate how the interface introduced in Figure 5.2 is used when two threads execute the same code in a RMT technique to detect data corruptions. Figure 5.4 shows a simple replication

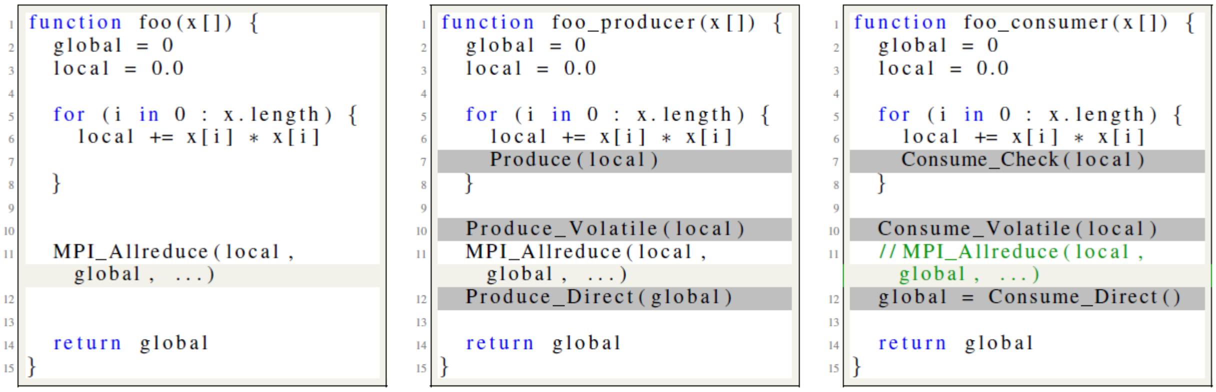
```
method Produce_Volatile(val)
{
    S = (UNIT - (enqLocal%UNIT)) % UNIT
    if (S > 0)
    {
        enqLocal = (enqLocal+S) % QSIZE
        deqCached = deqIndex
        enqIndex = enqLocal
    }

    volValue = val
    volState = 0

    while (volState == 0) {}
}


method Consume_Volatile(val)
{
    while (volState == 1)  {}

    if (not equal(val, volValue))
        Report_Soft_Error()

    volState = 1
    S = (UNIT - (deqLocal%UNIT)) % UNIT

    if (S > 0)
    {
        deqLocal = (deqLocal+S) % QSIZE
        deqIndex = deqLocal
        enqCached = enqIndex
    }
}
```

Figure 5.3: Implementation of synchronous communication

scenario. Figure 5.4(a) shows the original code. Figure 5.4(b) and Figure 5.4(c) show the code of the producer (*leading*) thread and the consumer (*trailing*) thread, respectively. In the source code there is a *for* loop that calculates a value and stores it in the variable *local*. Then it is shared among all ranks of the MPI application with an *MPI_AllReduce()* operation, which leaves the final result in the *global* variable. At last, the function returns this latter variable.

For each non-volatile variable computed by the program, a call to *Produce()* is added in the leading thread and a call to *Consume_Check()* in the trailing thread. This is illustrated with the first highlighted code lines in Figures 5.4(b) and 5.4(c), where each value computed inside the *for* loop is checked for soft errors.

After the last iteration of the *for* loop, the *local* variable should be considered as a volatile



```
1  function foo(x[]) {
2    global = 0
3    local = 0.0
4
5    for (i in 0 : x.length) {
6      local += x[i] * x[i]
7
8    }
9
10
11   MPI_Allreduce(local,
       global, ...)
12
13
14   return global
15 }
```

(a) Original Code

```
1  function foo_producer(x[]) {
2    global = 0
3    local = 0.0
4
5    for (i in 0 : x.length) {
6      local += x[i] * x[i]
7      Produce(local)
8    }
9
10   Produce_Volatile(local)
11   MPI_Allreduce(local,
       global, ...)
12   Produce_Direct(global)
13
14   return global
15 }
```

(b) Producer Code

```
1  function foo_consumer(x[]) {
2    global = 0
3    local = 0.0
4
5    for (i in 0 : x.length) {
6      local += x[i] * x[i]
7      Consume_Check(local)
8    }
9
10   Consume_Volatile(local)
11   //MPI_Allreduce(local,
       global, ...)
12   global = Consume_Direct()
13
14   return global
15 }
```

(c) Consumer Code

FIGURE 5.4. Code replication example

variable since it is then used as a parameter in the call to the function `MPI_Allreduce()`. Thus, the final value stored in the variable *local* has to be checked using the *Produce_Volatile()* or *Consume_Volatile()* functions. This ensures that the value stored in *local* is tested for correctness before running the MPI function call.

Finally, note that the call to the external library function `MPI_Allreduce()` is only executed by the leading thread. Hence the value returned by the function should be sent from the leading thread to the trailing thread. The *Produce_Direct()* and *Consume_Direct()* methods are used for this purpose (at lines 12).

5.2 Techniques to improve performance of RMT

This section describes the strategies and optimizations we propose to improve the performance of RMT techniques in the context of HPC applications. In Section 6, we evaluate the impact of these optimizations by applying them to the basic RMT solution we presented above. The main goal of these enhancements is to improve the performance and to reduce the cost of the communication between the leading thread and the trailing thread.

The first idea we propose to improve the performance of communication is to leverage SMT to host a thread and its replica on the same physical core. Then, we propose two ideas that aim at limiting the amount of data exchanged between replicas: *variable aggregation* and *selective checking*. We present these ideas in this section.

5.2.0.1 The use of simultaneous multithreading

HPC applications usually try to take advantage of multicore architectures by performing computations in parallel. Most recent processor architectures additionally implement simultaneous multithreading (Hyper-Threading Technology in the case of Intel processors). SMT allows to have 2 or more threads concurrently executing in the same physical core [27]. It makes a single physical processor appear as (at least) two logical processors. The physical execution resources are shared and the architecture state is duplicated for the virtual processors.

Because it is important to utilize all available resources, the use of SMT threads would seem obvious for HPC applications. Sadly, not all programs scale up properly with this feature. Different data access patterns make up for different performance effects. Whether the program is CPU or memory bound impacts significantly how much benefit can SMT deliver. The limited scalability of some HPC workloads with SMT threads has been highlighted in several studies [26, 29, 30].

Regardless of whether SMT threads can be used to solve the original application problem or not, they can help speed up soft-error detection. Hence, we propose to use SMT in our context to improve the performance of RMT. Since the leading thread and the trailing thread need to communicate very frequently, it can be good for performance to place them as close as possible on a multicore processor. We propose to accomplish that by placing (and pinning) the trailing thread in the same physical core where the leading one resides. Exchanging data between threads that share the first cache level can be of great aid at reducing the performance degradation induced by RMT synchronization.

5.2.0.2 Variable Aggregation

Another way to reduce the performance impact of communication between the replicas is to reduce the amount of data exchanged between the threads.

We can significantly reduce the queue stress by grouping several values together before enqueue/dequeue operations. We refer to this approach as *variable aggregation*. It is based on an idea presented in [23]. The authors state that two values can be unified through an operation, if the result still allows soft error detection on either value. The aggregate `(+)` operator[1] is one of this kind. For example, if two values need to be checked, we can add them together in both threads and just compare their result. If one value in a thread gets corrupted then its output will differ from the one in the other thread, and the soft error would still be detected.

In this paper, we study a generalization of *variable aggregation* where more values can be grouped together to further reduce the number of enqueues and dequeues. Aggregating multiple values together weakens soft-error detection. First, if more than one value is corrupted, there is a small chance that, due to the aggregation, 2 bit-flips or more would compensate each other and go undetected. Second, there is a risk of overflow when computing the aggregated value. Third, even if a SDC is detected, we lose the information about which specific value was corrupted. Note that this last point is not a major issue in case recovery is based on a checkpointing mechanism,

[1] We chose the (+) operator because it does not require to cast variables to other types, as it would be the case for bit-wise operators such as `XOR` for instance.

```
method VA_Produce(val)
{
    pGroupVal += val

    if (pIter++ % GRANULARITY == 0 )
    {
        Produce(pGroupVal)
        pGroupVal = 0
    }

}
```

(a) VA_Produce routine

```
method VA_Consume(val)
{
    cGroupVal += val

    if (cIter++ % GRANULARITY == 0)
    {
        Consume_Check(cGroupVal)
        cGroupVal = 0
    }

}
```

(b) VA_Consume routine

Figure 5.5: Implementation of *variable aggregation* optimization

it might not be important to know which of the last *n* instructions was corrupted, but to know that at least one of them is, so the recovery can begin.

One important implementation detail is that the grouping granularity should be a power of 2, because modulo operations can be implemented in a very efficient way using bit-wise operations for such values.

Figure 5.5 shows how the *variable aggregation* optimization is implemented. This strategy can only be applied in the case of asynchronous communication. We replace *Produce()* and *Consume_Check()* function calls by wrappers to those functions that handle variable aggregation. If a synchronous communication is initiated through *Produce_Volatile()*, the current *groupVal* is immediately pushed to the queue even if not enough data have been aggregated.

5.2.0.3 Selective checking

Another more aggressive optimization that we propose is to only check for soft errors at certain points of the application. As previously explained, *volatile stores* are the point where a thread could make a data corruption visible to the *outside world*. As such, we propose to only check values (address and data) that are used for store instructions to volatile variables. We refer to this optimization as *selective checking*.

In the examples of Figures 5.4(b) and 5.4(c), selective checking implies that the intermediate values generated for the variable *local* inside the loop (at line 7) are not checked anymore: only the final value at the end of the loop is checked for correctness. Therefore a lot of checks will be removed.

Since this optimization reduces the number of computed values that are checked for soft errors, it also weakens the detection mechanism. We think that checking all store to volatile variables should allow detecting any error that would impact the final result of the application. One new issue could be that the time between the corruption and its detection would increase due to this mechanism. Running experiments to better understand these points is part of our future work.

A similar idea has been explored on a ILR approach on ESoftCheck [36]. The authors try to identify the minimum number of checks necessary to detect a soft error and still be able to recover the application. It uses a set of compiler optimizations techniques in order to automatically identify what needs to be replicated. For example it attempts to discover redundant checks, those that are postdominated by another nearby check. To our extent, the idea of selective checking has not been studied on a RMT solution.

Chapter 6

Evaluation

In this chapter we present our evaluation. We start this by describing the implementation of RMT used for our tests and the testbed. We evaluate the impact of the optimizations we propose using two benchmarks from the Mantevo project.

6.1 Implementation

To evaluate the efficiency of our optimizations, we chose to implement RMT directly in the code of the tested applications by modifying the code of these applications. Automatically replicating an application for RMT using for instance a compiler approach has already been done in other works [23, 34, 37], and it could also be done for our solution. The goal of thesis paper is to assess the gain that can be obtained thanks to the optimizations we propose. As such, we chose to delay the implementation of an automatic tool to future work.

Hence, in the experiments presented in this paper, we modified the code of the tested applications to create two threads out of the main thread of the applications and to call the *Produce()* and *Consume()* functions defined in the previous section appropriately.

The code to implement our technique (SPSC queue, comparison of values, etc.) has been implemented in C. Special care has been taken to properly align variables and to pad the data structures to avoid any false sharing.

6.2 Setup of the experiments

All experiments presented below are run on a node equipped with 2 Intel Xeon E5-2630L v4 processors (*Broadwell* architecture) and 128 GB of RAM. Each processor features 10 physical cores, that is, 20 physical cores in total and 40 logical cores when Hyper-Threading is enabled.

The machine runs Debian 9.5 with Linux kernel 4.9.0-7-amd64, GCC version is 6.3.0-18 for Debian and Open MPI 2.0.2.

Evaluations are executed with two applications included in the Mantevo benchmark suite: HPCCG and CoMD. The first one is representative of workloads from finite element methods. It takes matrix dimensions as arguments used to calculate the global problem size. CoMD is a reference implementation of typical classical molecular dynamics algorithms. It can also take arguments that impact the total number of atoms used in the simulation [13]. We use the MPI version of these two applications.

Unless otherwise stated, the global problem size used for running each application is the following: a total number of atoms of $2.4565 * 10^6$ in CoMD and a global problem size of $1.28 * 10^8$ in HPCCG. All results presented in this section are average execution times over 10 runs of each application. The histogram graphs include error bars, showing the standard deviation, however they represent in general less than 1% of variation, so they are hard to spot.

6.2.1 Scaling of HT in the original applications

The main goal of this experiment is to observe how hyper-threading impacts the original applications. We tested the scalability of the two applications when using hyper-threading by comparing the performance using 20 MPI ranks and 40 MPI ranks, this is exactly the point where the processors begins to use all logical cores and therefore hyper-threads. We observed a limited performance improvement of just about 18% for CoMD and even a performance degradation of 4.2% for HPCCG. Figure 6.1 helps to show this behavior.

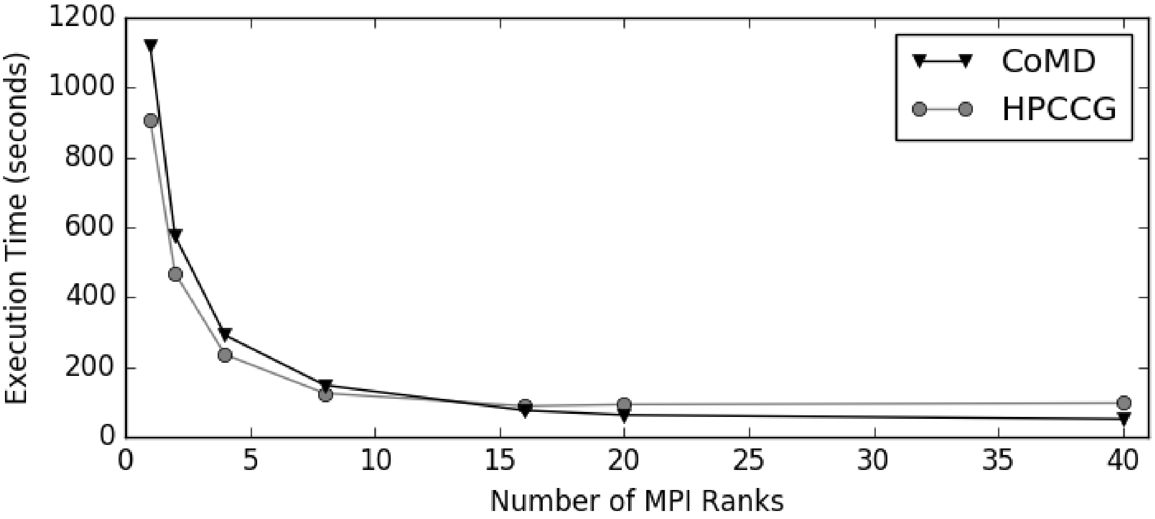
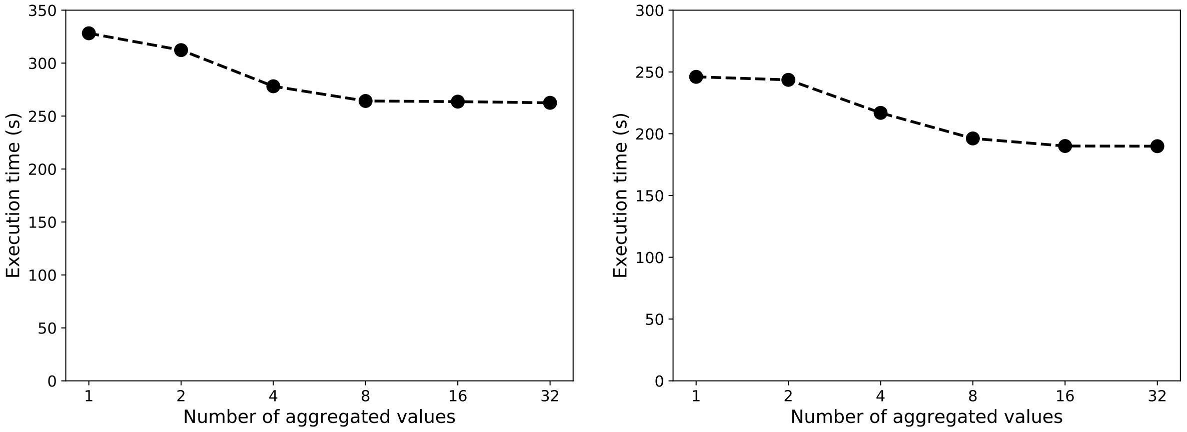


Figure 6.1: Scaling of applications with multiple ranks

6.3 Impact of aggregation granularity on performance

The next experiment we run aims at studying the impact of the number of values we aggregate together on the efficiency of the *variable aggregation* optimization. Figure 6.3 shows the result for the two applications. The values in the graphics are the execution times of the replicated execution of each application, as more values are aggregated it would be expected that the performance would improve, however we can see that aggregating more than 16 values does not lead to any significant gain. Therefore, for all other experiments we use this value for aggregation.



(a) CoMD

(b) HPCCG

Figure 6.2: Impact of variable aggregation granularity on RMT performance

We can normalize the values of the last two figures based on the execution time of the not-replicated execution. By doing so we can calculate how much performance overhead the replicated execution incurs in each application. As show in Figure 6.3, CoMD is the one that experiments more performance overhead (more than 6x), but by applying variable aggregation more than 1x is gained. HPCCG on the other hand starts around 2.8x and this optimization is able to reduce the overhead to almost 2x.

6.4 Performance impact of the optimizations

In this experiment we study the performance impact of *variable aggregation* and *selective checking* on RMT in both applications. For each configuration, we also compare the performance when the *leading thread* and the *trailing thread* are on the same core (labeled *Hyper-threads*), and when they are on different cores (labeled *Different Cores*). As mentioned in Section 6.2 the test machine's architecture has two Intel Xeon X5-2630L processors each one with 10 physical cores and 20 logical cores. In the first one, the cores have the following numbering 0/20, 2/22, ..., 16/36, 18/38, each pair representing the two logical cores of each physical core. That means that the

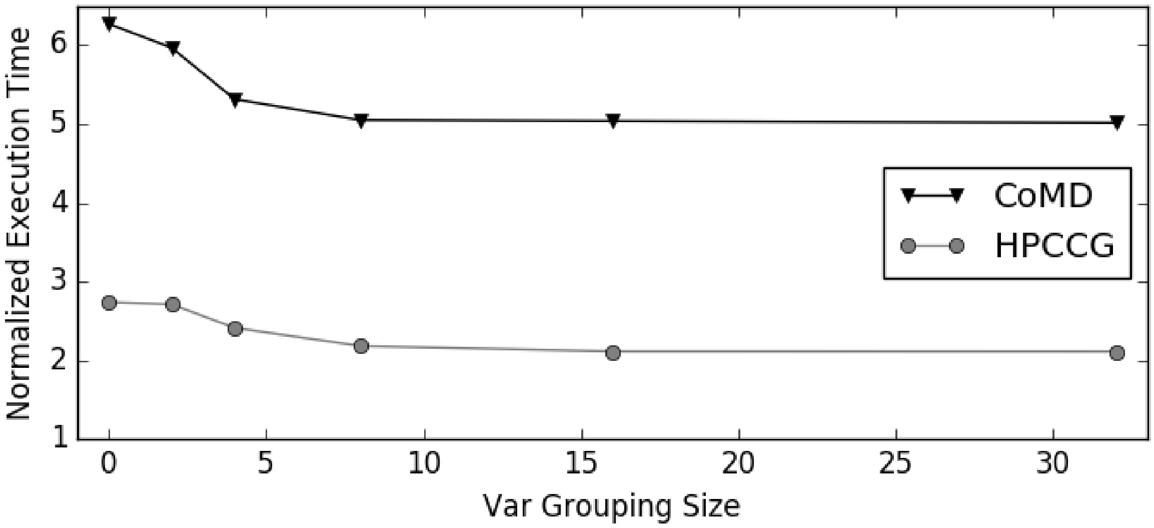


Figure 6.3: Normalized execution time with difference variable aggregation granularity

first physical core (of the first processor) has the logical cores 0 and 22, and so on. In the second processor, the cores are numbered in the same fashion like this 1/21, 3/23, ..., 17/37, 19/39.

Figure 6.4 shows the *Different cores* configuration, where we pin each leading and trailing thread to a different physical core but on the same processor. This configuration would be the most likely if no pinning is manually specified, meaning if the operative system is the one responsible to assign threads to cores. In the figure, each pair of colors represents the leading and trailing threads.

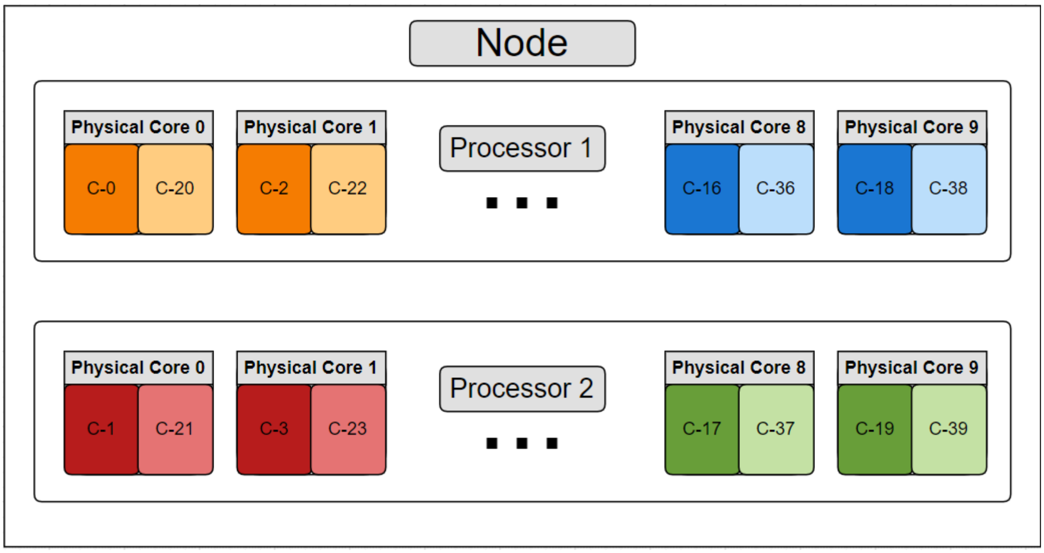


FIGURE 6.4. Different cores configuration, each pair of colors represent a pair of leading and trailing threads.

Figure 6.5 on the other hand shows the *Hyper-threads* configuration, where we pin each leading and trailing thread to the same physical core. This configuration allows the sibling replicated threads to share the core resources. Again, in the figure, each pair of colors represents the leading and trailing threads.

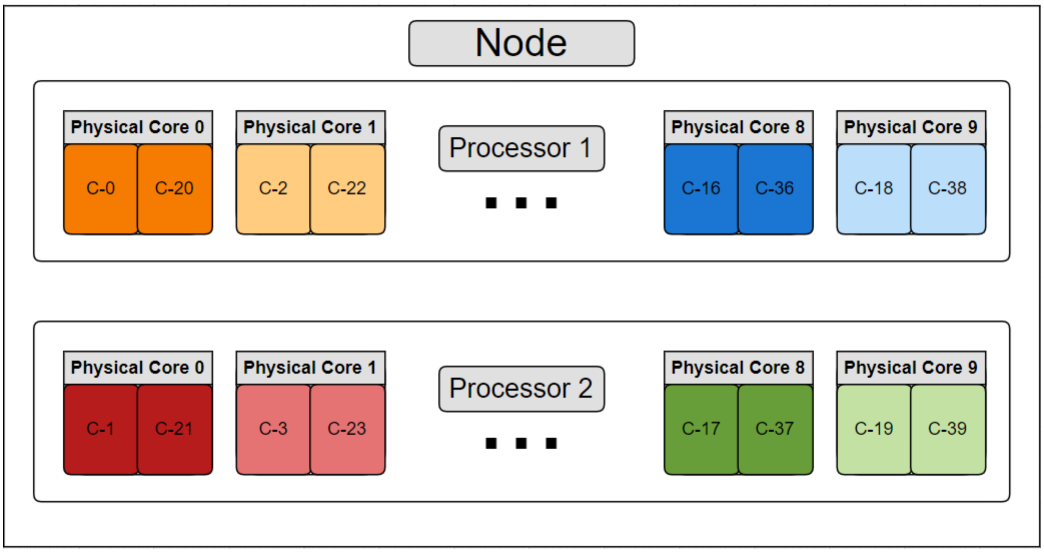
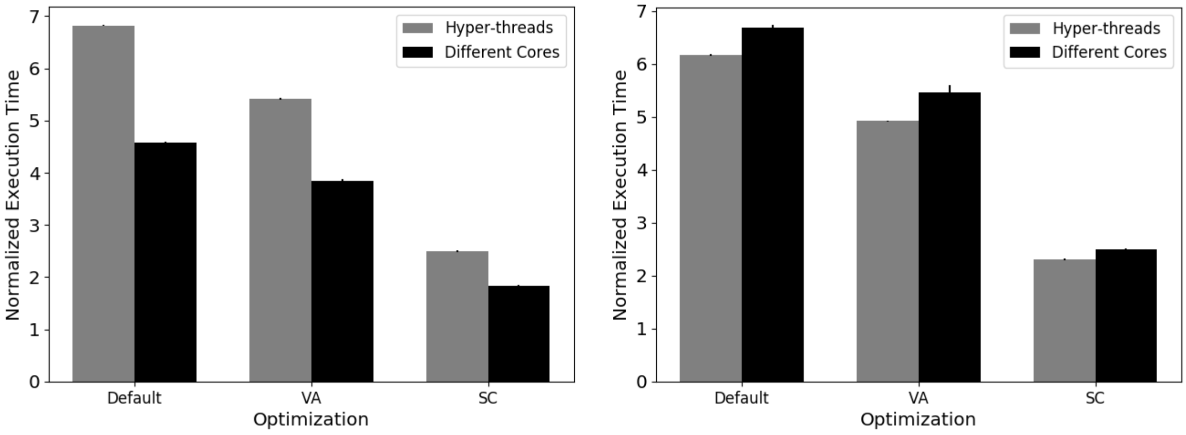


FIGURE 6.5. Hyper-threads configuration, each pair of colors represent a pair of leading and trailing threads.

Figures 6.6 and 6.7 study the performance impact of *variable aggregation* and *selective checking* on RMT in both applications. The default configuration is when no optimization is applied. Then each optimization is evaluated independently. The results are presented as a normalize execution time taking the execution time of the unmodified application as reference.

Figures 6.6(a) and 6.7(a) show the performance when running only 1 MPI rank. The problem size is then changed as follows for each application: $1.08 * 10^5$ atoms for CoMD and a problem size of $8.1 * 10^6$ for HPCCG. Evaluation with one rank implies that when the replicas are run on hyper-threads of the same core a single core is used, whereas when replicas are executed on different cores, two cores are used. In this case, we can observe that the impact of each optimization on the performance is significant. However, running replicas on different cores is faster but uses 2 cores instead of one.

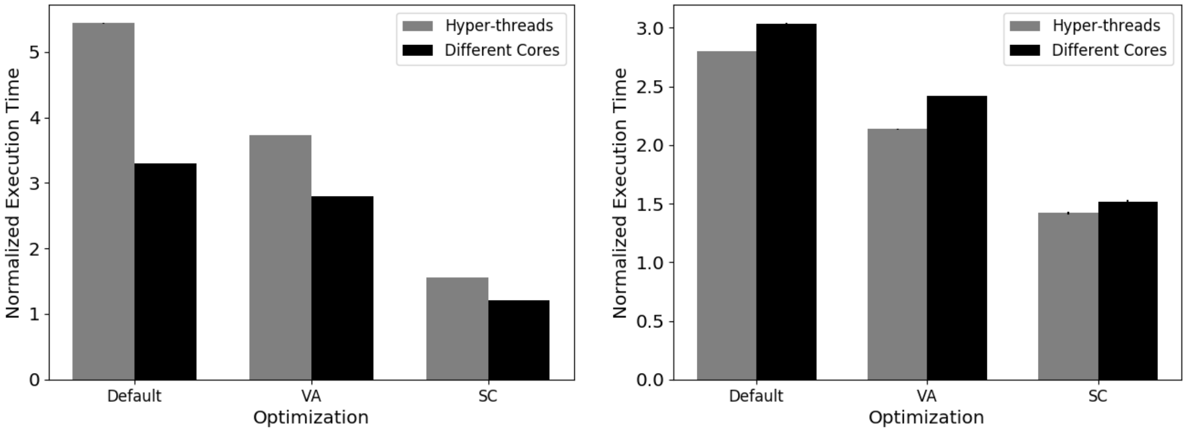
Figures 6.6(b) and 6.7(b) show the performance when running 20 MPI ranks, that is 40 threads in total when RMT is applied. In this case, in the configuration named *hyper-threads*, threads are pinned so that each core hosts the two replicas of the same rank. In the configuration named *Different Cores*, threads are pinned so that half of the cores host the *leading* threads and the other half host the *trailing* threads. If core A hosts the *leading* thread of two ranks, the *trailing* thread of these two ranks are also hosted on the same core B. Furthermore, pinning is done in such a way that cores A and B are on the same NUMA node.



(a) CoMD single rank

(b) CoMD all ranks

Figure 6.6: Performance in all configurations for CoMD



(a) HPCCG single rank

(b) HPCCG all ranks

Figure 6.7: Performance in all configurations for HPCCG

In the experiments with 20 ranks, it is better to have both the leading and replica threads on the same core; just by choosing where to place the replica thread can imply a performance boost. This option outperforms the other configuration even though they are using the same amount of resources. In all levels of optimizations on both applications those improvements range from 6.2% to 11.6%.

In the experiment with 20 ranks, *variable aggregation* on CoMD reduces by 18,75% the overhead of RMT while for HPCCG it shows a 22.73% improvement. *Selective checking* is the optimization that delivers the best performance. For CoMD it improves the performance of RMT by 62.18% and for HPCCG by 58.56%.

Finally the best configuration in the executions with 20 ranks is with *selective checking* and replicas placed on hyper-threads of the same core. In the case, CoMD is 2.2x slower than a non replicated execution, and HPCCG is only 1.42x slower. These results make us think that RMT can be a viable solution for soft-error detection in HPC applications.

6.5 Problem size impact on performance overhead

The objective of this experiment is to show how the best replicated execution behaves when varying the original problem size. Table 6.1 is a comparison on HPCCG using 20 ranks, between the baseline execution, that is the execution without RMT, and the best replicated configuration (*selective checking* when placing the replicas on hyper-thread of the same core). The first row of Table 6.1 is the global problem sizes considered. The second row shows the mean execution times in seconds of the baseline while increasing the problem size. The third row shows the mean execution times of the best replicated execution. Finally, the last row shows the overhead normalized to the baseline. We see that the overhead of 1.41x observed in Section 6.4 remains valid for different problem sizes.

Table 6.2 is the equivalent for CoMD. The first row of Table 6.2, must be multiplied by 10^6 (for space reasons omitted) to get the different number of atoms used. Once again the 2.22x performance overhead observed in Section 6.4 remains valid for different problem sizes.

Sizes	$4 * 10^6$	$6 * 10^6$	$8 * 10^6$	$1 * 10^7$	$1.2 * 10^7$
Baseline (sec)	57.6	87.4	116.1	145.7	175.5
Rep. (sec)	81.9	122.9	164.3	205.9	247.8
Overhead	**1.42x**	**1.41x**	**1.41x**	**1.41x**	**1.41x**

Table 6.1: HPCCG - Impact of the problem size on performance.

Sizes $*10^6$	2.048	2.4565	2.916	3.4295	4
Baseline (sec)	52.6	62.9	75.6	88.3	101.8
Rep. (sec)	118.3	139.8	166.4	193.9	225.3
Overhead	**2.25x**	**2.22x**	**2.20x**	**2.19x**	**2.21x**

Table 6.2: CoMD - Impact of the problem size on performance.

6.6 Pause instruction as a minor optimization

In addition to all optimizations evaluated in this section, we tested a solution to improve the performance of the SPSC queue in the case where replicas are hosted on the same physical core. Namely, we tried reducing the impact of the spin-wait loops involved in the synchronous communication using the *“pause”* instruction. This instruction is specifically designed for processors supporting hyper-threading, to help speeding up performance when executing such code patterns. It provides a hint to the processor that the code sequence is a spin-wait loop, avoiding memory order violation and preventing the pipeline flush. In addition, it also de-pipelines the loop, preventing it from consuming execution resources excessively. By synchronizing using *“pause”*, it is expected that spin-wait loops yield the core’s execution units to the other hyper-thread thus providing a better resource utilization overall [14]. However, experiments with both applications running 20 ranks with RMT showed only 1.5% of performance improvement on average.

Chapter 7

Conclusion and Discussion

In this chapter we present our conclusions for the current thesis dissertation. An analysis about the objectives and hypothesis is presented, in order to give a formal closure to the work. Finally, we provide a discussion on several topics that we did not have time to accomplish.

7.1 Conclusion

This thesis presents several optimizations to improve the performance of software-based RMT for soft-error detection. Our results first show that leveraging SMT threads can help improve the performance of RMT by placing the replicas of one thread on the same physical core. We also propose two optimizations that aim at reducing the amount of data exchanged between the replicas of one thread. *Variable aggregation* and *selective checking* significantly improve the performance of RMT at the cost of weakening soft-error detection. In the best configuration, that is when *selective checking* is applied and replicas of a thread are run on the same physical core, the performance overhead of RMT can be as low as 1.42x, making it a candidate solution to help detecting SDC at extreme scale.

Our next step for this work is to run experiments with fault injection to precisely quantify the degradation in error detection introduced by the proposed improvements of RMT. Still, we think that our optimizations can provide interesting new trade-offs between the reliability of soft-error detection and performance.

7.2 Thesis objective analysis

Regarding each specific objective we believe we have accomplished all of them. We were able to implement every single optimization we proposed and evaluate their performance impact on RMT. The main objective is achieved through the specific ones and even though for CoMD the performance overhead is not currently less than 2x, it is close to the desired value. So, we feel that we were able to demonstrate for some cases that RMT is a viable soft error detection solution at extreme scale.

7.3 Discussion

For now, our technique is not applied automatically to the application's code. So, decisions such as what is cataloged as a volatile access or which values must be directly shared between threads are performed manually. However, such operations can be automated eventually, by trying to identify information available as compile time (such as *volatile* attributes in variables of C programs). Also, programming hints can be provided in order to ease the automation process.

We also plan to include other applications to the analysis. More applications will reveal different data access patterns and different performance overheads, which will make the study more complete. More testing scenarios are also to be included in the future, different architectures might yield different results.

One idea that might help improve performance of RMT techniques, is to take advantage that leading and trailing threads move at different speeds. In this scenario, the leading thread is faster than its replica, because the latter has to make extra computations to validate results. One could try to take advantage of the situation by allowing the consumer to read values from the queue without performing sync checks; i.e, asking if the index has already been produced. Since most of the times the producer should have pushed new data to the buffer, most of times the soft-error checks will match. Special consideration must be made for the mismatches, where the replica gets ahead. Counting enqueues/dequeues operations can be a way of identifying those cases.

BIBLIOGRAPHY

[1] *GCC: gnu compiler collection.*
`http://gcc.gnu.org`.
Accessed: 20/08/2017.

[2] A. AVIZIENIS, J.-C. LAPRIE, B. RANDELL, AND C. LANDWEHR, *Basic concepts and taxonomy of dependable and secure computing*, IEEE transactions on dependable and secure computing, 1 (2004), pp. 11–33.

[3] L. BAUTISTA-GOMEZ, F. ZYULKYAROV, O. UNSAL, AND S. MCINTOSH-SMITH, *Unprotected computing: a large-scale study of dram raw error rate on a supercomputer*, in High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for, IEEE, 2016, pp. 645–655.

[4] E. BERROCAL, L. BAUTISTA-GOMEZ, S. DI, Z. LAN, AND F. CAPPELLO, *Lightweight silent data corruption detection based on runtime data analysis for hpc applications*, in Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2015, pp. 275–278.

[5] ——, *Toward general software level silent data corruption detection for parallel applications*, IEEE Transactions on Parallel and Distributed Systems, 28 (2017), pp. 3642–3655.

[6] J. CALHOUN, M. SNIR, L. N. OLSON, AND W. D. GROPP, *Towards a more complete understanding of sdc propagation*, in Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2017, pp. 131–142.

[7] C. CONSTANTINESCU, *Trends and challenges in vlsi circuit reliability*, IEEE micro, 23 (2003), pp. 14–19.

[8] S. FENG, S. GUPTA, A. ANSARI, AND S. MAHLKE, *Shoestring: probabilistic soft error reliability on the cheap*, in ACM SIGARCH Computer Architecture News, vol. 38, ACM, 2010, pp. 385–396.

[9] D. FIALA, F. MUELLER, C. ENGELMANN, R. RIESEN, K. FERREIRA, AND R. BRIGHTWELL, *Detection and correction of silent data corruption for large-scale high-performance com-*

puting, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, p. 78.

[10] J. Giacomoni, T. Moseley, and M. Vachharajani, *Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue*, in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM, 2008, pp. 43–52.

[11] F. Haas, S. Weis, T. Ungerer, G. Pokam, and Y. Wu, *Fault-tolerant execution on cots multi-core processors with hardware transactional memory support*, in International Conference on Architecture of Computing Systems, Springer, 2017, pp. 16–30.

[12] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*, vol. 21, ACM, 1993.

[13] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, *Improving performance via mini-applications*, Sandia National Laboratories, Tech. Rep. SAND2009-5574, 3 (2009).

[14] R. Intel, *Intel r 64 and ia-32 architectures. software developer‚Äôs manual. volume 3a*, System Programming Guide, Part, 1 (2010).

[15] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, *Haft: Hardware-assisted fault tolerance*, in Proceedings of the Eleventh European Conference on Computer Systems, ACM, 2016, p. 25.

[16] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer, *Elzar: Triple modular redundancy using intel avx (practical experience report)*, in Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on, IEEE, 2016, pp. 646–653.

[17] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, *Ipas: Intelligent protection against silent output corruption in scientific applications*, in Proceedings of the 2016 International Symposium on Code Generation and Optimization, ACM, 2016, pp. 227–238.

[18] L. Lamport, *Concurrent reading and writing*, Communications of the ACM, 20 (1977), pp. 806–811.

[19] P. P. Lee, T. Bu, and G. Chandranmenon, *A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring*, in Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–12.

[20] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, et al., *Hyper-threading technology architecture and microarchitecture*, Intel Technology Journal, 6 (2002), pp. 4–15.

[21] H. Menon and K. Mohror, *Discvar: Discovering critical variables using algorithmic differentiation for transient faults*, in Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, 2018, pp. 195–206.

[22] 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*, IEEE Transactions on Device and Materials Reliability, 5 (2005), pp. 329–335.

[23] K. Mitropoulou, V. Porpodas, and T. M. Jones, *Comet: communication-optimised multi-threaded error-detection technique*, in Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, ACM, 2016, p. 7.

[24] K. Mitropoulou, V. Porpodas, X. Zhang, and T. M. Jones, *Lynx: Using os and hardware support for fast fine-grained inter-core communication*, in Proceedings of the 2016 International Conference on Supercomputing, ACM, 2016, p. 18.

[25] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, *Acr: Automatic checkpoint/restart for soft and hard error protection*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 7.

[26] L. Porter, M. A. Laurenzano, A. Tiwari, A. Jundt, W. A. Ward Jr, R. Campbell, and L. Carrington, *Making the most of smt in hpc: System-and application-level perspectives*, ACM Transactions on Architecture and Code Optimization (TACO), 11 (2015), p. 59.

[27] S. K. Reinhardt and S. S. Mukherjee, *Transient fault detection via simultaneous multithreading*, 28 (2000).

[28] 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, IEEE Computer Society, 2005, pp. 243–254.

[29] S. Saini, J. Chang, and H. Jin, *Performance evaluation of the intel sandy bridge based nasa pleiades using scientific and engineering applications*, in International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Springer, 2013, pp. 25–51.

[30] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas, *The impact of hyper-threading on processor resource utilization in production applications*, in High

Performance Computing (HiPC), 2011 18th International Conference on, IEEE, 2011, pp. 1–10.

[31] V. SHARMA, G. GOPALKRISHNAN, AND G. BRONEVETSKY, *Detecting soft errors in stencil based computations*, tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), 2015.

[32] G. UPASANI, X. VERA, AND A. GONZÁLEZ, *Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery*, in ACM SIGARCH Computer Architecture News, vol. 42, IEEE Press, 2014, pp. 37–48.

[33] T. VIJAYKUMAR, I. POMERANZ, AND K. CHENG, *Transient-fault recovery using simultaneous multithreading*, in ACM SIGARCH Computer Architecture News, vol. 30, IEEE Computer Society, 2002, pp. 87–98.

[34] C. WANG, H.-S. KIM, Y. WU, AND V. YING, *Compiler-managed software-based redundant multi-threading for transient fault detection*, in Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, 2007, pp. 244–258.

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

[36] J. YU, M. J. GARZARAN, AND M. SNIR, *Esoftcheck: Removal of non-vital checks for fault tolerance*, in Code Generation and Optimization, 2009. CGO 2009. International Symposium on, IEEE, 2009, pp. 35–46.

[37] Y. ZHANG, J. W. LEE, N. P. JOHNSON, AND D. I. AUGUST, *Daft: decoupled acyclic fault tolerance*, International Journal of Parallel Programming, 40 (2012), pp. 118–140.