Heterogeneous Fault Tolerance for Software

Xiao Liu

Master of Science
School of Informatics
University of Edinburgh
2021

Abstract

Microprocessors are vulnerable to bit-flip that may cause transient errors in computing. Current mainstream solutions for fault tolerance usually require custom hardware to implement redundancy. There are already some attempts to use software ways like compiler support to build fault tolerance on general programs. However, these software schemes wasted too many resources in logging and didn't make use of the new parallelism in checking. This project proposed a new software scheme for fault tolerance based on two novel architectures proposed by Ainsworth and Mitropoulou. By using a software way to exploit new parallelism in checking code and targeting specific algorithms to reduce the overhead of logging, the new approach cannot only avoid the cost of custom hardware, but also achieve better efficiency and latency compared with traditional software schemes.

Acknowledgements

I want to thank my supervisor Sam Ainsworth for his insightful comments, guidance and encouragement throughout the project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Xiao Liu*)

Table of Contents

Chapter 1

Introduction

1.1 Motivation

With the progress of the processor production technology, the size of transistors is constantly shrinking so that more transistors can be integrated on a single chip. But this also leads to an increased risk of bit-flip faults in modern microprocessors[19]. Bit flip faults indicate transient changes in the state of transistors in CPU, such as changing from 0 to 1, and vice versa. It may corrupt the data and cause errors or malfunctions in programs. Transient faults usually occur when the processor is affected by specific conditions like severe vibration, high temperature or cosmic rays[43]. The reduction in the size of transistors also exacerbates this risk.

The risk of bit-flip faults severely threatens the safety and security of programs running on affected processors. For those scenarios that have high demands for the robustness and confidentiality of computing, transient faults in microprocessors have always been a great concern. For example, in automotive industry and aerospace industry, transient faults may cause errors and even crashes in software running on electronic equipment[26, 6]. Previous research also revealed that bit-flip faults may compromise the security of cryptographic algorithms like AES and RSA. Attackers managed to hack the key of AES by deliberately introducing transient faults into the protected device, which is called 'Differential Fault Analysis'[15]. There are also attempts to make use of transient faults to break into an SGX enclave[32] in Intel CPUs and attack RSA signatures[8].

The effective detection and recovery of transient faults in microprocessors has always been a research focus during the past few decades, various solutions have been proposed to solve it. The current mainstream in industries is lock-stepping[11], which

runs two copies of the same program on two separate cores, then synchronizes and compares result of every instruction to detect any hardware faults. However, this approach involves duplicating the processor core and customizing hardware to implement redundancy, which is too expensive.

To reduce the cost of duplicating hardware, redundant multi-threading(RMT) has been proposed to implement redundancy in a cheaper way. It is achieved by making use of simultaneous thread technology, which runs two copies on two simultaneous threads: a main thread and a checker thread instead of duplicating the processor cores. The state-of-art in RMT is Comet proposed by Mitropoulou et al.[30], which is totally based on compiler support without duplicating or customizing the hardware. Comet is a software scheme and free of hardware cost, but it sacrifices some performance to support general programs. Comet logs all the data in a program, even those unchanged, to implement fault tolerance, which will slow down the main thread.

Previous research[35] has proved that the checker thread could be split and run in parallel, which could lower the overhead of energy as well as improve overall performance. The novel architecture in this field is parallel error detection proposed by Ainsworth et al.[1]. This technology could split a strict sequential program into several parallelizable parts based on strong induction principle, then running the segmented checker threads in parallel on energy-saving and slower cores, while using a power-hungry core to process the main thread. It could achieve lower checking latency and overall performance in comparison with previous software approaches like Comet, and is also expected to reduce the overhead of energy since smaller cores are more energy-efficient. However, this architecture involves custom hardware, thus more expensive than software schemes that are almost free.

The two novel approaches have their own pros and cons[30, 1]. COMET does not require custom hardware and is therefore almost free, but not efficient enough. Parallel error detection exploits new parallelism in checker code to achieve better performance, while its custom hardware leads to a higher cost. It seems feasible to combine the advantages of both to propose a more superior solution. Specifically, the new solution should achieve parallel error detection totally based on software to avoid custom hardware. It will also target specific algorithms rather than general programs to reduce unnecessary logging and improve overall performance.

In summary, there are two main research questions in this project:

1. Is it feasible to exploit the parallelism in a checker thread totally based on software without any custom hardware?

2. By only targeting specific algorithms and forwarding the protected data more finely, will it help lower the overhead of logging compared to supporting general programs?

1.2 Objectives

This project aims to use a software way to implement parallel checker threads without additional hardware support. Additionally, we also want to reduce the overhead of logging through reading data directly from memory when explicit logging is unnecessary. We will realize and develop this architecture on several sequential algorithms to produce example programs for software parallel error detection, for instance, Cipher Block Chaining (CBC) encryption mode of AES.

In order to realize this goal and demonstrate the advantages of our method over traditional methods such as Comet, the following objectives must be achieved:

1. Implement software-based parallel error detection on one or several strict sequential algorithms like CBC without custom hardware.

2. Compile and run the examples programs of different algorithms on a heterogeneous device which has two types of cores: power-hungry and power-saving cores, bind checker threads to power-saving cores.

3. Evaluate the overall performance of our approach based on running time, overhead of logging and possibly energy, then compare it with several traditional methods like Comet.

1.3 Contribution

This project proposed a software solution for fault tolerance based on two novel architectures. It provided a software way to implement parallel fault tolerance on sequential algorithms without any custom hardware. The new approach has been proved to be more efficient than traditional software schemes in most cases through actual experiments. It can not only achieve lower latency for checking by exploiting new parallelism, but also avoid the additional cost of hardware and lower the overhead of energy. The new technique also uses more efficient hardware than existing software schemes: the parallelism allows it to run on energy-efficient ”little cores” even though they are slow.

Chapter 2

Background

2.1 Related work

2.1.1 Lock-stepping

As the current mainstream in relative industries, lock-step is widely used and there are many successful applications, for example, Arm Cortex-R series processors[11]. The basic idea of lock-step is very simple, which just runs the program twice on two separate processor cores. The result of every instruction is synchronized between two cores and compared to ensure the correctness. Recovery steps will be conducted when any errors detected[10]. Traditional lock-step architectures use two cores to implement redundancy. Recently, a triple-core design[22] has been proposed to reduce the latency of recovery when detecting transient faults. Both of the dual-core and triple-core lock-step architectures involve duplicating hardware and processor cores, which is very expensive and requires custom hardware.

2.1.2 Redundant multi-threading

To reduce the cost of duplicated processor cores and achieve the same redundancy, some researchers turned to the simultaneous multi-threading(SMT) technology[33], using duplicated threads instead of cores to implement fault tolerance. Rotenberg et al.[38] proposed a new method for error detection called AR-SMT based on SMT, which spawns two simultaneous threads on a single core to avoid duplicated cores. The results of instructions on two threads are compared at the end of every cycle, which may slow down the program and cannot make the most of the core's computing resources. Inspired by AR-SMT, SRT[36] was developed to improve the overhead

of synchronization by using a finer way to reschedule resources and synchronize between two threads instead of comparing the result of each instruction. SRT managed to outperform AR-SMT at least 16% in experiments.

However, both SRT and AR-SMT can only cover soft faults and will ignore those permanent faults since the two threads are run on the same core. Austin et al. proposed a new architecture called DIVA[4] to detect both soft and hard errors. DIVA added a simple in-order core to run the checker thread and detect faults in an out-of-order core, which also adds to the cost as it involves custom hardware. Based on SRT, Mukherjee et al. further extends it and developed a chip-level redundant architecture called CRT[31]. It applied a dual-core processor to run main and checker threads separately, which is expected to improve performance of multi-threaded workloads. It also managed to detect those hard faults since the two threads are targeted to separate cores. Compared with lock-step, RMT hardware methods can reduce the cost of duplicated hardware with the help of SMT technology, but they still require custom hardware to implement simultaneously threads.

2.1.3 Software schemes

Both RMT and lock-step involve custom or duplicated hardware, some researchers tried to build totally software-based schemes for fault tolerance that is free of any hardware costs. SWIFT[37] provides a software solution for instruction-level fault tolerance based on modifications of compiler. It can be directly run on regular processors without any custom hardware. The idea of it is to duplicate the instructions in general programs, then execute them twice and compare results. Since SWIFT only uses a single thread, it doesn't make use of the parallelism in CMP structure. Inspired by this, SRMT[41] split the main and checker code into two threads running on two processor cores in parallel, which is expected to achieve better speed on CMP processors. DAFT[25] further reduced the cost of synchronization on CMP processors by running the main and checker threads asynchronously.

As the novel architecture in this field, COMET[30] proved that the overhead of synchronization between checker and main threads is the major bottleneck in existing software schemes. To relieve this issue, COMET developed a mechanism to generate a series of code at the stage of compiling to accelerate it. However, it still wastes many resources in logging to support general programs, which can be further improved to reduce the overhead within the main thread.

2.1.4 Parallel error detection

Prior research has proved that it is possible to make use of the new parallelism in checker code itself to improve the overall performance. The first attempt is the architecture proposed by Rashid et al.[35], which exploits the parallelism in checking to segment the checker thread into parallel ones, then runs them on a device with multiple cores. This can reduce the latency of checking as well as achieve lower overhead of energy since multiple smaller cores are more efficient than a single powerful core.

There are also some attempts to build fault tolerance based on heterogeneous devices. Necromancer et al.[3] used some functionally dead cores to assist and accelerate a smaller light core by providing high-level hints. DIVA[4] cited before also combined two different types of cores: in order and out of order to help verify the pipeline of instructions.

By exploiting the new parallelism in checker thread with the help of heterogeneous processors, Ainsworth et al.[1] proposed a novel architecture for fault tolerance that is both efficient and consumes less energy. This is also part of the basis of our project. This method managed to run the checking code of even sequential programs in parallel with the assistance of strong induction principle. Those parallel checker threads will be deployed on the little cores of a heterogeneous system, which can achieve better performance as well as reduce energy overhead.

However, this technology involves additional hardware and is more expensive than software schemes. Thus, our project aims to use a software way to achieve parallel fault tolerance on heterogeneous cores. We also want to reduce the overhead of logging compared with Comet by only targeting specific algorithms and conducting the synchronization in a finer way.

2.2 Heterogeneous system

Our approach will make use of the heterogeneous processors to reduce the overhead of energy. Heterogeneous devices consist of different kinds of cores, and today's big. LITTLE system[20] is one of them. There are two types of cores in a big. LITTLE system: smaller cores which are simple and in-order and bigger ones which are powerful and out-of-order.

Those bigger cores are power-hungry and much faster than little cores. However, little cores are more energy-efficient and suitable for parallel workloads. For instance,

4 little cores running in parallel may be more powerful than a single big core and they consume less energy to achieve better performance[18]. For those workloads which are parallelizable, running them on multiple smaller cores is better.

Since our method can split the checker thread of even sequential algorithms into parallelizable segments. Binding checker threads to those little cores in a big. LITTLE system will improve the overall latency as well as save power. For the main thread of sequential algorithms, we will target it to a powerful big core since it is not parallelizable. By running main and checker threads on two different types of cores, we cannot only reduce the latency of checking, but also achieve it with a lower cost of energy.

Chapter 3

Software parallel error detection

3.1 Target algorithm

This project aims to build a better software scheme for error detection without custom hardware, as well as make use of the new parallelism in checking and reduce the overhead of logging by targeting specific algorithms.

The selection of target algorithms in this project is based on two principles: sequential and specific. We only target sequential algorithms in this project since our approach exploits new parallelism in checking to improve latency. For those algorithms which are already parallelizable, there is no need to exploit new parallelism anymore. Only for those strictly sequential algorithms, can we show the advantages of our method over traditional ones. In addition, to avoid unnecessary logging, our approach only targets specific algorithms rather than general ones like Comet[30]. This is because we can forward modified data more finely in specific programs to reduce the overhead of logging rather than log all the involved data like Comet.

Based on the above two principles, we chose 5 specific algorithms to achieve fault tolerance in this project: AES-CBC, integer sorting, Zlib compression, RSA public key and Blackscholes. They were chosen not only because they are representative and widely used in the fields of encryption and scientific computing, but also because they are susceptible to transient faults. For example, some researchers have discovered that when an attacker performs a ”differential fault analysis” [15], a bit flip failure may cause AES to leak secrets. We hope to implement fault tolerance on these five algorithms, based on which to show the feasibility of software parallel error detection and its advantages over traditional solutions like Comet.

3.2 New parallelism in checking code

The basic idea of fault tolerance is to make use of redundancy to detect transient errors. The easiest way is to just execute the instructions twice, then compare the results to ensure correctness[24]. Traditional RMT methods like Comet[30] implements redundancy by creating a single checker thread to execute the program twice. However, prior research[35] has shown that there is new parallelism even in strict sequential algorithms, thus the checker thread itself could be split and run in parallel. The new parallelism could be exploited based on strong induction principle as described by parallel error detection[1].

In a strict sequential algorithm, every step depends on the output of previous step, for example, CBC, zlib, RSA and so on. Thus, strict sequential algorithms usually cannot be run in parallel due to the data dependence. However, when it comes to the checker thread, the data dependence can be removed based on a strong induction principle. To elaborate further, suppose there is a sequential algorithm E, and there are three data variables A, B, C that will be processed sequentially in E. E(A) means all the operations on A in program. Therefore, the data dependence between A, B, C can be shown as below.

$$E(A) -> E(B) -> E(C) \tag{3.1}$$

In the checking code of above algorithm, the data dependence between A, B, C can be removed by exploiting strong induction principle. Once the main thread has finished computing E(A) and E(B), the checker thread cannot only use E(A) to recalculate and verify E(B), but also use E(B) to produce and verify E(C) at the same time by assuming E(B) is correct. That's to say, the checking of E(B) and E(C) could be run in parallel by assuming E(B) is correct. This concept could be generalized to all the sequential operations. By supposing there is no error in previous operation, the checking of a series of sequential operations could be parallelized. When all checking of all the operations has been completed, the correctness of all the whole algorithms could be verified based on strong induction principle[1]. As such, the data dependence in general sequential programs has been removed.

Based on the above concept, the checker thread of sequential algorithms could be parallelized to improve the overall performance. We will use a main thread to run a sequential algorithm, then create multiple parallel checker threads to verify the results of every segmented block. If all the operations are checked to be correct, then there is no transient fault in the whole process. Otherwise, any faults will be detected by

finding the first block that reports error.

3.3 General architecture

Although we will implement fault tolerance on specific algorithms, the overall architecture of parallel error detection is the same for all programs. We will describe the general architecture of software fault tolerance in this section, then implement it on 5 different sequential algorithms in remaining chapters.

3.3.1 Segmentation and synchronization

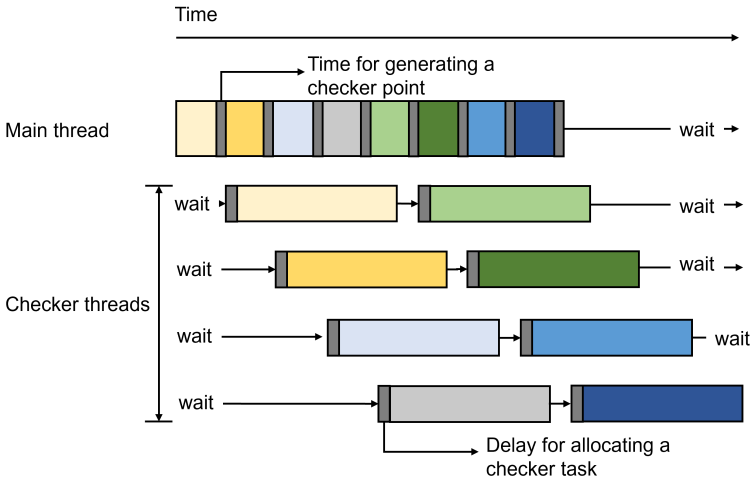


Figure 3.1: Sequence diagram of the main thread and checker threads

This project uses a totally software scheme to achieve parallel error detection on specific sequential algorithms. This is achieved by splitting the main thread into multiple segments, then running those checker threads in parallel to check them based on strong induction principle(see figure 3.1). The main thread executes these sequential segments in order. It will spawn a new checker task at the end of each segment, then prepare any necessary data for the checker thread to consume and verify. We will run the main thread on a powerful out-of-order core, while binding these checker threads to some smaller and energy-saving cores. Therefore, the main thread will run several times faster than a single checker thread, and the execution of these checker threads will overlap in time.

At the start of each segment, the main thread will make copies of the data to be modified in this round. When the computing of a segment is finished, the main thread

will forward the original data as well as the computed results to the checker thread. The checker thread will assume the data before modification is correct, then use it to recalculate and verify the correctness of final results of the corresponding segment.

3.3.2 Fault detection

The checking result of each segment will be stored in an array(see section 3.3.4). If any transient errors occur during this process, it will only be detected in its segment. Note that all the checker threads run in parallel, it means that the transient faults may spread to later segments since the checking of previous segments may not have been finished when the checking of latter blocks starts. If an error occurred in the previous segment, the next segment will assume the wrong result to be correct, then use it verify remaining segments. Thus, the later checker threads may not be able to detect the error since the original data has been corrupted[1].

As such, to detect the accurate place where the fault occurs, we have to find the first segment which reports error. This is achieved by setting semaphores between main and checker threads[39], when a checker thread detects a fault, it will notify the main thread, then error information will be logged and all the threads will be forced to stop.

3.3.3 Automatic task allocation and balance

A thread pool with a limited size is used to manage these checker threads instead of creating each checker thread manually. This is because destroying and recreating a thread is very expensive[27], and the max number of threads on a single core is also limited. Thus, we decided to develop a thread pool integrated with a task queue to reuse these threads and allocate checker tasks automatically.

At the start of the protected program, the main thread as well as a checker thread pool with a given number of threads will be created. These checker threads are reusable and bound to different little cores at the creation of thread pool. A single checker thread will be in charge of multiple discrete segments. We do not need to spawn a new checker thread for every segment, which can greatly reduce the overhead of creating and destroying threads.

Using a thread pool and task queue to manage checker tasks can also help balance the workloads between different little cores[42]. New checker tasks will be allocated to free threads first. Every time when the main thread has finished dealing with a

segment, it will push a new checker task to the queue in thread pool. When a checker thread is free, it will automatically remove the task from the queue for execution. If all the checker threads are occupied, these checker tasks will be queued and wait for checker threads to consume.

3.3.4 Data flow between threads

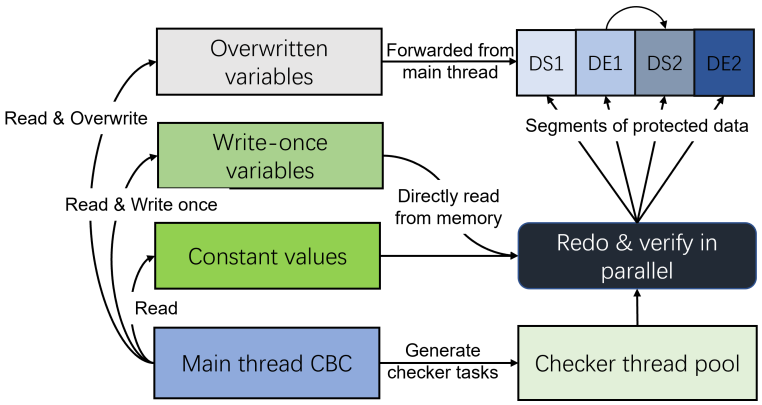


Figure 3.2: Parallel fault tolerance architecture for general programs

Compared with traditional RMT methods which log all the data like Comet, our scheme applies a finer way to transfer data between main and checker threads. For a given program, the data inside it can be divided into three types according to how often they are modified: constant values, write-once variables and overwritten variables(see figure 3.2).

Constant values refer to variables that will not be modified and are unchanged during the whole process, so the checker thread can directly read it from the main thread without making copies. Write-once variables refer to the data that will only be altered once by the main thread, which means that the data is unchanged since the start of a checker thread. Therefore, the checker thread can also read it directly. Note that we assume the memory is already protected by ECC, which means that the transient errors will only occur in the process of computing[34]. Thus, the constant values which is directly read will be considered to be correct in default. For those overwritten variables, if an error occured during computation and corrupted the output data. It will be detected when the checker threads repeat the same steps to compute and compare.

Overwritten variables are more special, they are expected to be modified for multiple times by the main thread, which indicates that overwritten variables may still be altered when its checker threads start. Thus, the main thread needs to make copies of

overwritten variables at the start and end of each segment, then forward it to checker threads.

By targeting specific algorithms, we only need to log the overwritten data and the other two types of data can be directly fetched from memory. This can greatly reduce the overhead of logging compared with traditional methods like Comet, where all the data is logged. Since we only target specific algorithms, we can manually select the overwritten data to be forwarded in a finer way.

3.4 Running time model

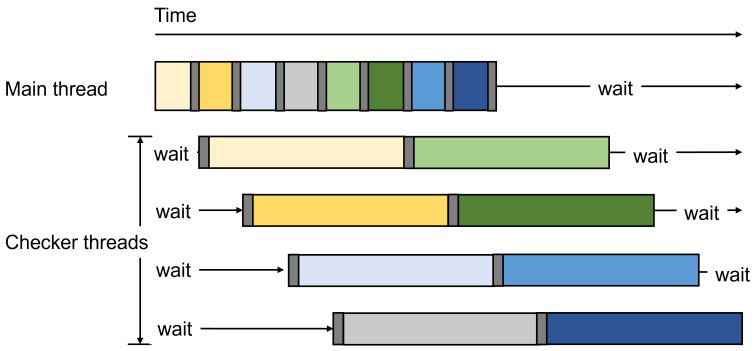


Figure 3.3: Congested sequence diagram of the main thread and checker threads

From the sequence diagram proposed in the general architecture(see figure 3.1), we can also derive the running time as well as latency of checking in theory. Since checker threads are running on those little cores, they will spend much longer than the main thread to process a single segment. When the checker thread is slower enough, checker tasks generated by main thread may be congested in the task queue since all the checker threads are occupied as shown in figure 3.3. Actually, figure 3.1 and 3.3 show two different cases of the running time model: idle and congested.

The checker tasks generated by main thread will be allocated to checker threads in order. When checker threads are idle, new tasks can be consumed immediately and the check thread will be vacant before the next task arrives. In this case, the overall running time is only determined by the running time of main thread and the latency of last segment(see figure 3.1). When it comes to the congested case, theses checker threads are so slow that they cannot finish checker tasks before the next task arrives. That's to say, the checker tasks will be congested and piled in the queue. For the congested case(see figure 3.3), the overall running time is much longer due to the

latency of each checker thread.

In order to quantitatively represent the overall running time in the two cases, we have derived a general formula to calculate it.

$$T(N,m) = \begin{cases} N*G + \frac{BASE}{N}*k + BASE + d + k*G & m \geq k \\ \frac{N}{m}*(d+k*G) + \frac{m}{N}*BASE + m*G + \frac{k}{m}*BASE & m < k \end{cases} \tag{3.2}$$

where N is the number of segments, m is the number of checker cores, G is the cost of generating a checker point in main thread, d means the delay for allocating a checker task in thread pool. BASE is the unprotected version's running time; k represents the relative performance of the main thread, which indicates how many times the main thread is faster than a single checker thread. T(N,m) represents the overall running time for different number of segments and cores.

When $m \geq k$, the checker threads are idle and can catch up with the checker tasks spawned by main thread. T(N,m) is only determined by the execution time of last segment. The minimum value of overall running time when $m \geq k$ is at

$$N = \sqrt{\frac{BASE}{G}*k} \tag{3.3}$$

When $m < k$, checker tasks will be piled and congested. The latency of each checker thread will be accumulated, which will result in longer overall running time. Compared with the case of $m \geq k$, the overall running time is slowed down by the accumulated latency. The minimum value is both determined by N and m.

$$m = \sqrt{\frac{BASE}{G}*k} \tag{3.4}$$

$$\frac{N}{m} = \sqrt{\frac{BASE}{(d+k*G)}} \tag{3.5}$$

The above formula is directly derived from the theoretical model, it will also be tested and verified later through the implementation on five specific algorithms. Based on the general architecture, we will describe how to implement parallel fault tolerance on specific algorithms in details in the next chapter.

Chapter 4

Implementation on specific workloads

In this chapter, we will describe how we implemented software parallel error detection on the five specific algorithms selected before. It briefly introduces the principle and intent of each algorithm, how to protect different types of data in the process, and some pseudo-code to help explain the parallel fault tolerance.

4.1 Cipher Block Chaining

Cipher block chaining(CBC) is one of the main encryption modes in AES algorithm. AES is chosen for its wide use and vulnerability to transient faults that may be exploited by 'Differential fault analysis'[15]. To handle plaintext with uncertain length, AES will divide the text into blocks with fixed length, then apply different operation modes to encrypt them[2]. The two most representative modes in AES are Electronic codebook (ECB) and Cipher Block Chaining (CBC).

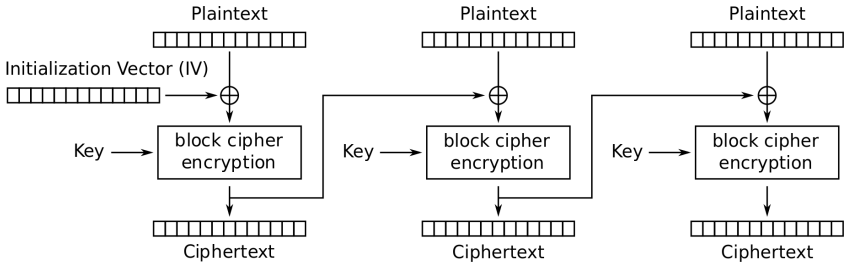


Figure 4.1: CBC encryption mode in AES[40]

We will only target CBC mode in AES instead of ECB for the following reasons. First, ECB is not very safe since it applies the same encryption to every block, which

may leak the patterns in data[21]. Second, CBC is a sequential algorithm which cannot be run in parallel(see figure 4.1). In CBC mode, every block's encryption relies on previous block's result[40]. The dependence of data forces CBC to be sequential, while ECB is already parallelizable since there is no any data dependence there. Our approach can exploit new parallelism in strictly sequential programs, which is our advantage over traditional methods. As such, we will implement parallel fault tolerance on CBC instead of ECB.

4.1.1 Protected data

There are 4 variables involved in CBC algorithm(see figure 4.1): plaintext given by users, ciphertext storing the output of every block's encryption, initial vector storing the output of previous block and round key used to encrypt current block.

As stated in section 3.3.4, the data in general programs can be divided into 3 classes. In CBC algorithm, the plaintext and the key is a constant string and will not be changed during the whole process. The cipher text array used to store the output will be modified in every round of encryption. However, the cipher text produced by previous segment will not be modified again in later blocks, so it is an write-once variable. As such, the checker threads in CBC can directly read plaintext and cipher text of previous segment from the memory. The initial vector will be updated in every encryption block and they will be overwritten by later segments, so the main thread has to make copies of them at the start and end of each segment, then forward them to the corresponding checker thread.

Main thread	**Checker thread pool**	
cipher[i] = E(P, Iv, key)	**cipher[i]' = E(P, Iv, key)**	
Iv = cipher[i]	**Iv' = cipher[i]'**	
Enqueue Iv	**Dequeue Iv**	
cipher[i+1] = E(P, Iv, key)	**cmp cipher[i], cipher[i]'**	**cipher[i+1]' = E(P, Iv, key)**
Iv = cipher[i+1]	**cmp Iv, Iv'**	**Iv' = cipher[i+1]'**
Enqueue Iv		**Dequeue Iv**
		cmp cipher[i+1], cipher[i+1]'
		cmp Iv, Iv'

Table 4.1: Software parallel error detection code on CBC

The only data that requires to be logged in is Iv. This variable is very tiny, usually

no more than several bytes. Thus, it is expected that the cost of generating a checker point in CBC is quite small.

To implement parallel fault tolerance on CBC, the original CBC code will be segmented into many slices. At the end of each segment, the overwritten variable Iv will be copied and forwarded to the checker thread through a queue as shown in table 4.1. Once the corresponding checker thread retrieved Iv, it will verify it as well as ciphertext by executing the encryption on this segment again. Ciphertext, key and plaintext can be directly read from main thread, as stated in previous section.

4.2 NAS Integer Sort

NAS IS(integer sort) is an algorithm in NAS Parallel Benchmarks[5] used to evaluate the performance of supercomputers. In this workload, the keys themselves in an array are used as index to calculate the number of each key and generate distinct population[17]. This workload was chosen because of its memory-constrained latency. The actual computing of this algorithm is very tiny, just traverse the array and increase the corresponding key. However, each round in the loop involves accessing two large arrays. We hope to use this workload to illustrate the shortcomings of our method when dealing with memory-intensive workloads.

Algorithm 1 NAS Integer Sort[17]

```
1: function INTEGER_SORT(buff1, buff2)        ▷ Where buff1, buff2 - array
2:     n = len(buff1)
3:     for i = 0 to n − 1 do
4:         buff1[buff2[i]]+=1
5:     end for
6: end function
```

4.2.1 Protected data

There are only two key arrays involved in IS: key_buff_1 and key_buff_2. key_buff_1 is a overwritten variable which will be updated in every round. The size of it is very large, about 2 MB. This means that the overhead of logging is considerable if we want to build parallel fault tolerance on this workload. The main thread must make copies

of the large array, then forward them to the corresponding checker thread, which may slow down the main program greatly when there are many checker points.

Main thread	Checker thread pool	
buff1[buff2[i]]++	buff1'[buff2[i]]++	
Enqueue buff1	Dequeue buff1	
buff1[buff2[i+1]]++	cmp buff1, buff1'	buff1''[buff2[i+1]]++
Enqueue buff1		Dequeue buff1
		cmp buff1, buff1''

Table 4.2: Software parallel error detection code on NAS-IS

IS is very special compared to other workloads like CBC and miniz where the data to be logged is very tiny. This workloads spends lots of time in memory accessing instead of computing, which leads to higher cost of generating a single checker point and lower the parallelism that could be exploited based on strong induction principle. To show the advantages as well as cons of our approach, we deliberately chose IS as one of the target algorithms. By implementing software parallel error detection on IS, we will figure out the performance of our architecture on different types of workloads and explore the applicable conditions of parallelism from more angles.

4.3 Zlib compression

Zlib is a famous algorithm used for data compression on many platforms like Linux and IOS. We will target Deflate[13], which is a main specification standard of zlib to implement fault tolerance. We will only cover the key sequential loop used for generating output buffer instead of the whole program which includes some steps for initialization and freeing variables.

The key principle of deflate compression is to find those duplicated patterns in data, then replace these items with custom marker bits to save space. Deflate algorithm applied vanilla hash chaining to match repeated strings in data that could be compressed, which is a sequential loop that constantly updates the dictionary and hash chaining in the process. The dictionary and hash chaining are used to store the strings that have been encountered before and those duplicated strings to be compressed.

Algorithm 2 Zlib Deflate[12]

```
1: function DEFLATE(data, buffer, hash, dict) ▷ Where data, buffer, hash, dict - array
2:     for i = 0 to len(data) − 1 do
3:         hash, dict = P(data[i])
4:         buffer[i] = F(hash, dict)
5:     end for
6: end function
```

4.3.1 Protected data

The data involved in vanilla hash chaining are dictionary, hash chaining, original data as well as the compressed buffer. Dictionary and hash chaining are overwritten variables, the compressed buffer is write-once variable and the original text is a constant string. The main thread has to copy the dictionary and hash chaining at the start and end of each segment, then store them for the checker threads to consume. Note that the size of dictionary and hash chaining is usually more than hundreds of bytes, especially when dealing with large amount of data. Thus, the overhead of logging in zlib is greater than that in CBC, which will also increase the cost of generating a single checker point as we stated in the running time model.

Main thread	Checker thread pool	
hash, dict = P(data[i])	**hash', dict' = P(data[i])**	
buffer[i] = F(hash, dict)	**buffer[i]' = F(hash', dict')**	
Enqueue hash, dict	**Dequeue hash, dict**	
hash, dict = P(data[i+1])	**cmp hash, hash'**	**hash', dict' = P(data[i+1])**
buffer[i+1] = F(hash, dict)	**cmp dict, dict'**	**buffer[i+1]' = F(hash', dict')**
Enqueue hash, dict	**cmp buffer[i], buffer[i]'**	**Dequeue hash, dict**
		cmp hash, hash'
		cmp dict, dict'
		cmp buffer[i+1], buffer[i+1]'

Table 4.3: Software parallel error detection code on Zlib

The checker threads will reparse the original data to update hash and dictionary again, then generate the compressed buffer based on the them. At the end of a checker thread, hash, dictionary and output buffer will be compared with those on the main thread to ensure the correctness of results.

4.4 Blackscholes

Our approach is also expected to work well on workloads that are already parallel because there is no data to be logged in parallel segments, which leads to a lower cost of generating checker points. To prove this, we also targeted a thread-level parallel workload called Blackscholes in parsec-benchmark[7]. It is a differential equation used to calculate how the value of options change as the price of underlying assets varies. This algorithm is thread-level parallelizable as there is no any data dependence in the loop. Each round in the loop can be allocated to a single thread. Thus, error detection on it is just like executing each round twice, then compare the results.

Algorithm 3 Blackscholes[28]

```
1: function PROCESS(rate, volatility, sptprice, strike, otime)
2:                                  ▷ Where rate, volatility, sptprice,strike, otime - array
3:     for i = 0 to len(rate) − 1 do
4:         price[i] = diffEquation(rate[i],volatility[i],sptprice[i],strike[i],otime[i])
5:     end for
6: end function
```

4.4.1 Protected data

Since there is no data dependency in Blackscholes, the main thread doesn't need to log any data during the process. The checker threads can directly read the input from the main thread and execute the fault tolerance procedure. There are two ways to do checking: comparing results at the end of each checker thread, or checking the final array when all threads are finished. We applied the first way to check each segment separately, which is hard to code but faster as it doesn't leave all the checking for the last thread to execute.

Main thread	Checker thread pool	
price[i] = diff(r,v,s,t,o)	**price[i]' = diff(r,v,s,t,o)**	
price[i+1] = diff(r,v,s,t,o)	**cmp price, price'**	**price[i+1]' = diff(r,v,s,t,o)**
		cmp price, price'

Table 4.4: Software parallel error detection code on blackscholes

Although there is no overhead of logging in this workload, the cost of generating a checker point still exists. This is because the main thread has to notify the corresponding checker thread in every round, which will take a tiny amount of time. When the number of segments is large enough, the cumulative cost will be very great.

4.5 RSA public crypto

RSA is an asymmetric encryption algorithm, which uses a private key and a public key to decrypt and encrypt separately. To show the generality of parallel fault tolerance, we will also apply out proposal on RSA to exploit the new parallelism. We will focus on the public key encryption algorithm of RSA-OAEP, which applies a random padding before encrypting data to ensure the security[29].

RSA is often used to process a limited length of data. It can only encrypt data that doesn't exceed the size of key, including the padding and header data, whose size is 11 bytes for PKCS[23]. Thus, the maximum size of data is 245 bytes for RSA 2048 and 501 bytes for RSA4096. Due to the small size of data, the parallelism that could be exploited in a single RSA block is also limited. Unlike previous sequential algorithms which could be divided into hundreds of segments, the encryption loop in RSA can only be split into several rounds, and this cannot exploit much parallelism.

Due to this, aside from building fault tolerance on a single RSA block, we will also try to use RSA in a way like ECB encryption to encrypt large data. We will divide the data which exceeds the limit into multiple blocks, then encrypt them separately using RSA-OAEP. RSA-OAEP could be used to encrypt large data divided by blocks and will not suffer the issue like ECB. This is because it will conduct a random padding of data before encryption, thus will not leak the patterns of data since it produces different results for multiple same blocks.

4.5.1 Single block

When running RSA-OAEP on a single block, it will do padding at first by generating random values to fill the blanked bits, then follow the RSA public encryption formular to calculate ciphertext: $C = m^e \mod n$. This is integrated in a sequential loop with only several rounds. Thus, there is only little parallelism in this workload to be exploited.

Algorithm 4 RSA OAEP[16]

```
1: function PUBLIC_ENCRYPT(input, cipher, pkey)  ▷ Where input1, cipher[i] - array, pkey - struct
2:     random_pad(input)
3:     for i = 0 to key_len/16 do
4:         cipher[i] = exp(input, pkey→e)
5:         cipher[i] = mod(cipher, pkey→n)
6:     end for
7: end function
```

Main thread	Checker thread pool	
cipher[i] = exp(input, pkey→e)	cipher[i]' = exp(input, pkey→e)	
cipher[i] = mod(cipher, pkey→n)	cipher[i]' = mod(cipher, pkey→n)	
cipher[i+1] = exp(input, pkey→e)	cmp cipher[i], cipher[i]'	cipher[i+1]' = exp(input, pkey→e)
cipher[i+1] = mod(cipher, pkey→n)		cipher[i+1]' = mod(cipher, pkey→n)
		cmp cipher[i+1], cipher[i+1]'

Table 4.5: Software parallel error detection code on RSA-OAEP

For RSA-OAEP on a single block, the involved data are the public key, original data and the cipher buffer. All of them are write-once or constant strings that don't need to be copied. Thus, the overhead of logging will be very minor in this workload. However, there is an issue that the number of segments is also very small, no more than 16 for RSA4096. This prevents us from improving the overall performance by exploiting the parallelism. As such, we came across another way to make use of RSA that can show the advantages of parallelism better.

4.5.2 Long text

RSA-OAEP can also be used to encrypt data that exceeds the length of key. This is achieved by dividing the data into multiple blocks with fixed length, which is a bit like ECB and CBC modes in AES. However, RSA-OAEP will not suffer the issues that ECB has since it uses random padding to generate different results for even the same input data, which can hide the patterns in protected data. By dividing the large data into blocks, we can gain much more parallelism than that in a single block.

For every single block in the long text, the algorithm will firstly generate random

padding values to fill the blanked bits of the original data, then use public key to encrypt the block. The computing of random values cannot be protected as it is unrepeatable and comes from the input of sensors or the real time of system. Actually, there is no need to protect the generating of random values in this workload. Since the random values are only written to the blanked bits after the end of valid data, the original data will never be corrupted no matter what kind of errors occur in the random padding. We can always recover the correct data by truncating the padded string.

This is based on an assumption that the memory is already covered by ECC, and so only data that is computed on (read and/or written back) is vulnerable to errors[34]. That isn't always strictly a requirement – we will pick up an error on the input data if it occurs after the first run but before the second, and so we can prevent fault analysis attacks even without ECC. The property we guarantee is that, assuming the input data is accurate (IE hasn't got any faults in it) within memory at the time of our first read, the padded results will also be correct. ECC is common these days anyway.

Algorithm 5 RSA OAEP LONG TEXT

```
1: function PUBLIC_ENCRYPT_LONG(input, cipher, pkey)   ▷ Where input,
   cipher - array, pkey - struct
2:     for i = 0 to input_len/block_len do
3:         random_pad(input[i])
4:         cipher[i] = encrypt(input[i], pkey)
5:     end for
6: end function
```

Main thread	Checker thread pool	
random_pad(input[i])		
cipher[i] = encrypt(input[i], pkey)	cipher[i]' = encrypt(input[i], pkey)	
random_pad(input[i+1])	cmp cipher[i], cipher[i]'	
cipher[i+1] = encrypt(input[i+1], pkey)		cipher[i+1]' = encrypt(input[i+1], pkey)
		cmp cipher[i+1], cipher[i+1]'

Table 4.6: Software parallel error detection code on RSA-OAEP for long text

Chapter 5

Experiments and results

To analyse the efficiency of our approach and find out how much parallelism we can exploit for different workloads, we deployed our example programs of parallel fault tolerance on an Odroid XU4 to collect the running time statistics of our approach. Odroid XU4 is an ARM big.LITTLE device consisting of 4 Cortex A15 and 4 Cortex A7 cores. A15 cores are out-of-order processors and power-hungry, A7 cores consume less energy and execute instructions in order. A Cortex A15 is 5x the area and 5x the power consumption, and that's even before we downclock the A7 to 600MHz from 1.5GHz, where power consumption is cubic in clock frequency[9]. The main thread of program is running on a powerful A15 core and those parallel checker threads are targeted to four A7 cores.

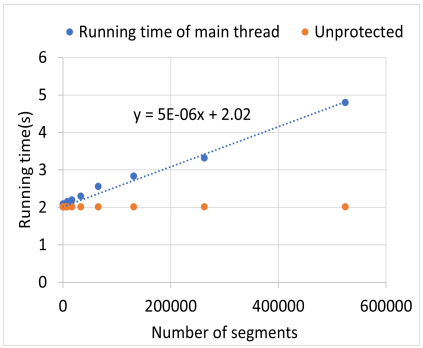
We conducted experiments to explore our approach's efficiency and its advantages as well shortcomings compared with traditional methods. It can be divided into 3 main parts: the cost of generating checker tasks, the slowdown of our approach relative the unprotected version and its performance compared with nZDC[14].(We also tried to compare with Comet, but its public source is incomplete and we cannot recreate it.)

5.1 Cost of generating checker points

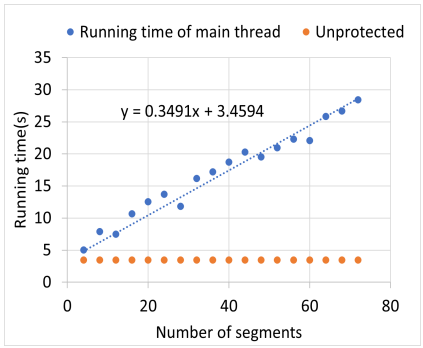
The cost of generating a single checker point refers to the delay in the main thread for preparing data and spawning a new checker task, which is an important parameter in our running time model. As the number of segments grow, the main thread is expected to be slower, and the slowdown caused by generating checker points is proportional to the number of segments. It can be represented in the following equation.

$$RM = BASE + N * G \tag{5.1}$$

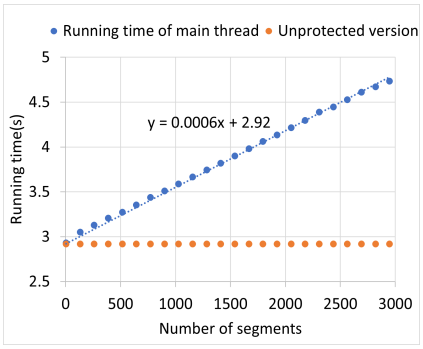
RM means the running time of main thread, BASE is the unprotected version's speed, G is the cost of generating a checker point and N is the number of segments.



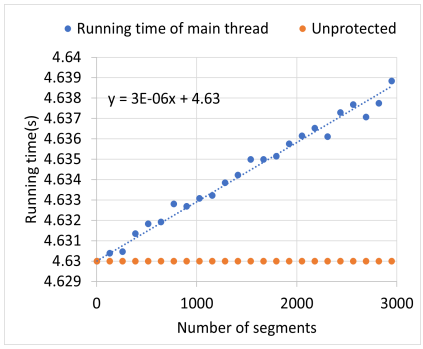
(a) CBC



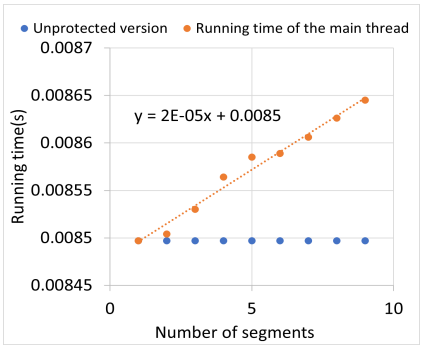
(b) NAS-IS



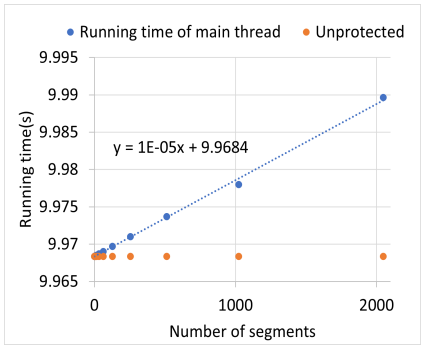
(c) ZLIB Compression



(d) BLACKSCHOLES



(e) RSA-OAEP



(f) RSA long text

Figure 5.1: Cost of generating checker points on six different workloads. The runing time of main thread(RM) is in blue and the unprotected program's(BASE) is in orange. There is also a linear fitting line in each subfigure, in which G is the slope and BASE is the intercept. G indicates the delay in the main thread to generate a single checker point.

To get the accurate value of G and BASE, we timed the main thread when there were different number of segments, then used linear fitting to calculate the cost. This was achieved by just segmenting the main thread but not conducting the checking. For each workload, we collected the data about how the running time of main thread varies with different number of segments, which is shown in Figure 5.1. By doing linear fitting on them, we can calculate the cost of generating a single checkepoint(G) and the unprotected version's time duration(BASE).

For example, in CBC workload, the only data that needs to be synchronized between main and checker threads is IV, which is only 16 bytes. Thus, the cost of generating a single checker point(G) is very tiny in theory. This can also be proved through actual experiments(see Figure 5.1(a)).. From the linear fitting curve, two parameters G and BASE can be known: $G = 5E - 6$, $BASE = 2.02$. However, for NAS Integer sort, G is expected to be much higher since two large arrays are copied in every round. G in IS workload is 0.3491 as shown in Figure 5.1(b).

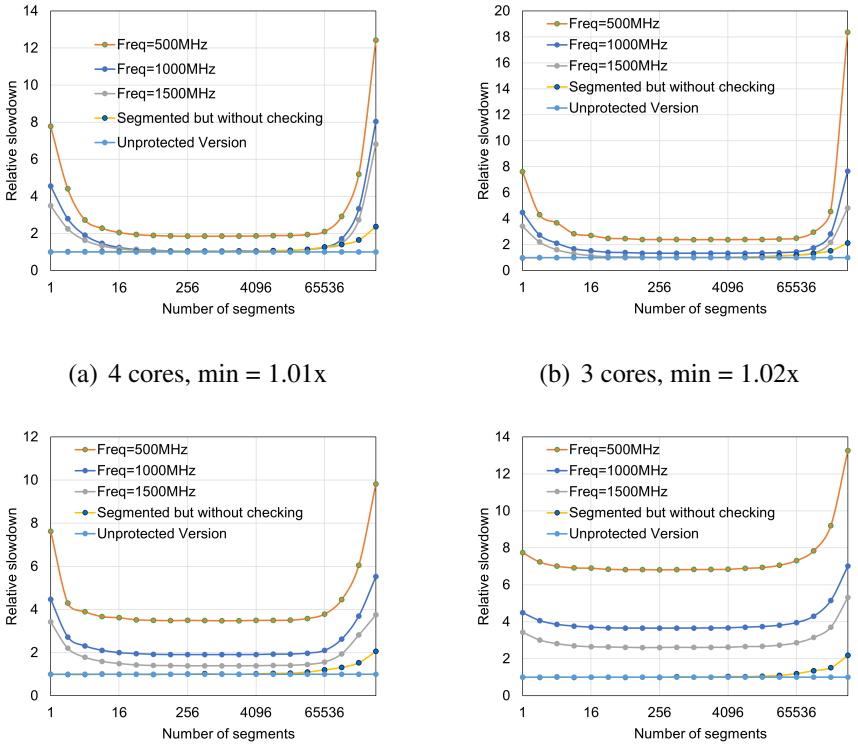
5.2 Slowdown due to fault tolerance

Slowdown is the ratio of the running time of protected version and the original program, which is a primary metric to show the efficiency of fault tolerance. The slowdown of our approach were explored in several angles.

First, its running time on different workloads were recorded and compared with that of the unprotected versions to calculate the relative slowdown and latency. This is achieved by setting timers when programming as well as using linux time tools. Second, we tried to tune different core and segmentation configurations to figure out how to make the best use of parallelism. This involves changing the size of segments, number of cores and munually downgrading the maximum frequency of checker cores. For example, we tried using only 2 A7 cores to run parallel error detection instead of 4 cores in default. We also downgraded the frequency of A7 from 1.5GHz to 500MHz and 1000MHz by modifying cpu governor to analyse how the checker cores's frequency will affect the parallelism and overall performance.

We used the statistics from experiments to plot the slowdown graph of different workloads and they are shown in the following of this section. We will discuss how much parallelism we can exploit when implementing fault tolerance on different workloads and show the pros as well as cons of our approach based on them.

5.2.1 CBC



(a) 4 cores, min = 1.01x

(b) 3 cores, min = 1.02x

(c) 2 cores, min = 1.38x

(d) 1 core, min = 2.31x

Figure 5.2: CBC: Slowdown relative to unprotected version of CBC with varying number of checker cores, segments and frequencies, where slowdown is the ratio of the running time of our approach and the unprotected CBC. We tried to run the parallel fault tolerance with different number of cores from 1 to 4 to test how much throughput is needed. The minimum slowdown with different number of cores is also listed at the bottom of each figure.

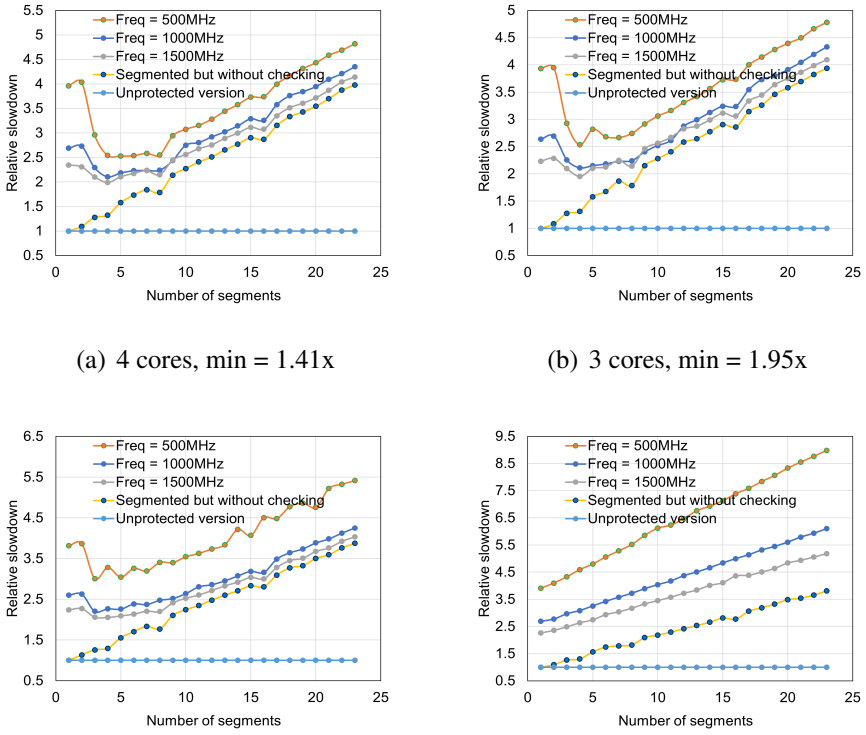
As the number of segments increase, the main thread will be slightly slower due to the cost of generating checker points. However, the latency of last segment also decreased at the same time because less work is left at the end. The overall running time is determined by both of the two factors: cost of checking and the benefits of parallelism, so it will achieve a balance somewhere. According to the slowdown graph on CBC(see figure 5.2), the overall running time will decrease at first, then grow again when the number of segments continue to grow, which corresponds to the running time model proposed before.

The slow down graph also shows the effects of different frequencies of checker cores. We manually downgraded the frequency of A7 from 1500MHz to 1000MHz

and 500MHz. It was found that higher frequency will lead to lower running time in general, but it still follows the patterns of our running time model. Actually, higer frequency means lower relative performance(k) in our model, which represents how many times the main core is faster than the checker core.

The number of checker cores will also determine the throughput of our approach, as show in slowdown graph, the minimum slowdown that our approach can achieve will decrease when the number of checker cores decline from 4 to 1.

5.2.2 NAS-IS



(a) 4 cores, min = 1.41x

(b) 3 cores, min = 1.95x

(c) 2 cores, min = 2.06x

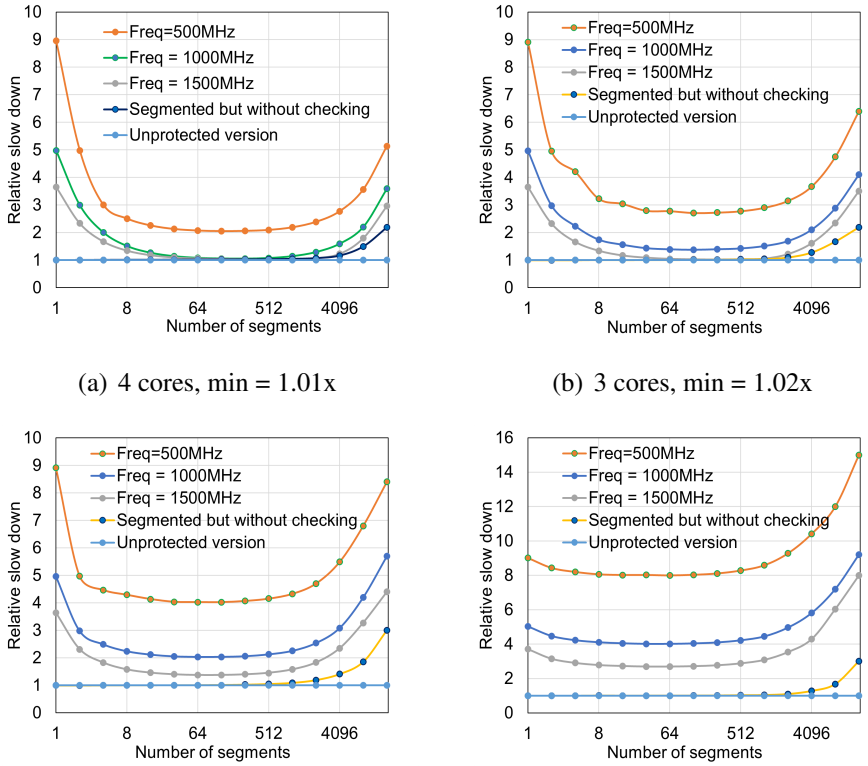
(d) 1 core, min = 2.26x

Figure 5.3: NAS-IS: Slowdown relative to unprotected version of zlib compression with varying number of checker cores, segments and frequencies. Due to the higher cost of generating checker points, the benefits of parallelism contributed by the number of segments is reduced in this workload. The minimum slowdown of fault tolerance on IS is 1.41x when using 4 checker cores with a frequency of 1500MHz and 4 segments.

NAS-IS is a memory-bound workload, which needs to copy two large arrays for every segment. Thus, the cost of generating a single checker point in IS is much larger than

that in CBC. As shown in figure 5.1(b), G = 0.3491s, BASE= 3.459s. Since the cost of generating checker points is very high in IS, the overall running time only declines for a while, then grows sharply as the number of segments increase. To achieve the minimum slowdown, the number of segments should be 4 according the experiments.

5.2.3 Zlib compression



(a) 4 cores, min = 1.01x

(b) 3 cores, min = 1.02x

(c) 2 cores, min = 1.37x

(d) 1 core, min = 2.69x

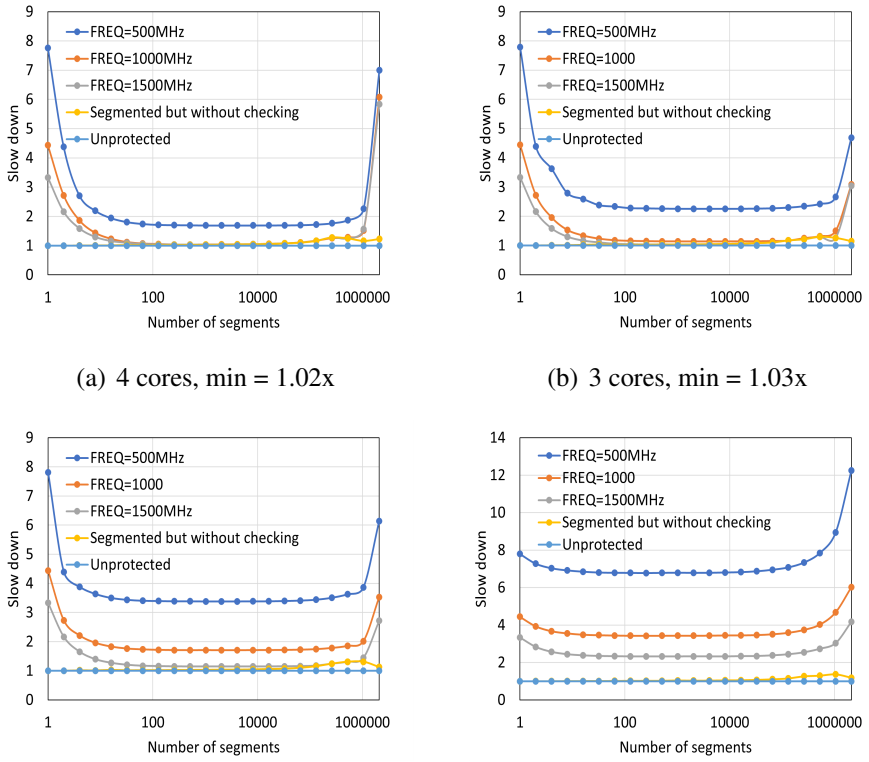
Figure 5.4: ZLIB: Slowdown relative to the unprotected version of zlib compression with varying number of segments, cores and frequencies. In this workload, using 3 checker cores could achieve the similar performance with using 4 cores.

For zlib compression, the data to be synchronized is bigger than that in CBC, but much smaller than that in IS. The cost of generating a single checker point is 0.00006 according to figure 5.1(c). For this workload, we also plotted its slowdown graph(see figure 5.4) and did the same analysis as in previous ones based on our running time model.

Figure 5.4 indicates that using 3 cores with a frequency of 1500MHz could catch up the throughput of the main thread while using 4 cores with a frequency of 500MHz

doesn't. Based on this, we found that the slowdown is determined by three factors: the throughput of checker cores, the benefits of parallelism and the cost of generating checker points. These 3 factors are all integrated in our model, where the number(m) and frequency(F) of checker cores represents their performance, the number of segments(N) reflects the level of parallelism and cost of generating checker points(G) is also included.

5.2.4 Blackscholes



(a) 4 cores, min = 1.02x

(b) 3 cores, min = 1.03x

(c) 2 cores, min = 1.14x

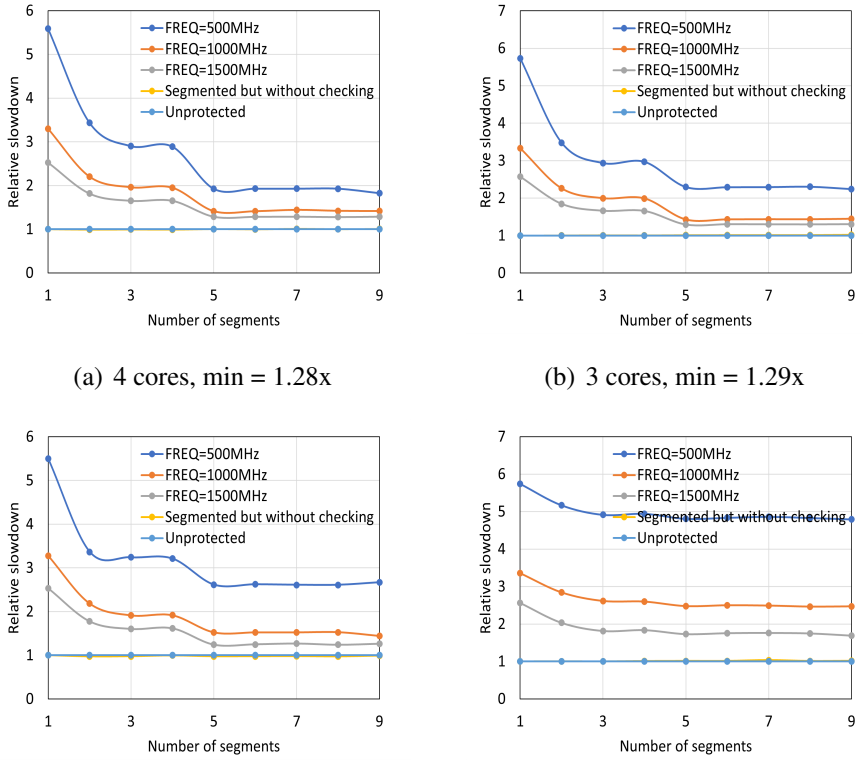
(d) 1 core, min = 2.31x

Figure 5.5: Blackscholes: Slow down relative to unprotected version of fault tolerance on blackscholes. Since the cost of generating checker points is quite little, the slowdown curve is very flat as the number of segments grow. The number of segments that could achieve the minimum slowdown is also much larger, close to around 2000.

In this workload, the cost of generating checker points is very minor since there is no data to be copied when creating checker threads in blackscholes, which is quite different from previous workloads like zlib or CBC, which still has a small amount of overwritten data. According to the experiments in section 5.2.1, the G in blackscholes

is only $3E-7$, which is very close to zero. It means that we can split the main thread into more segments to exploit the parallelism with very little overhead.

5.2.5 RSA single block



(a) 4 cores, min = 1.28x

(b) 3 cores, min = 1.29x

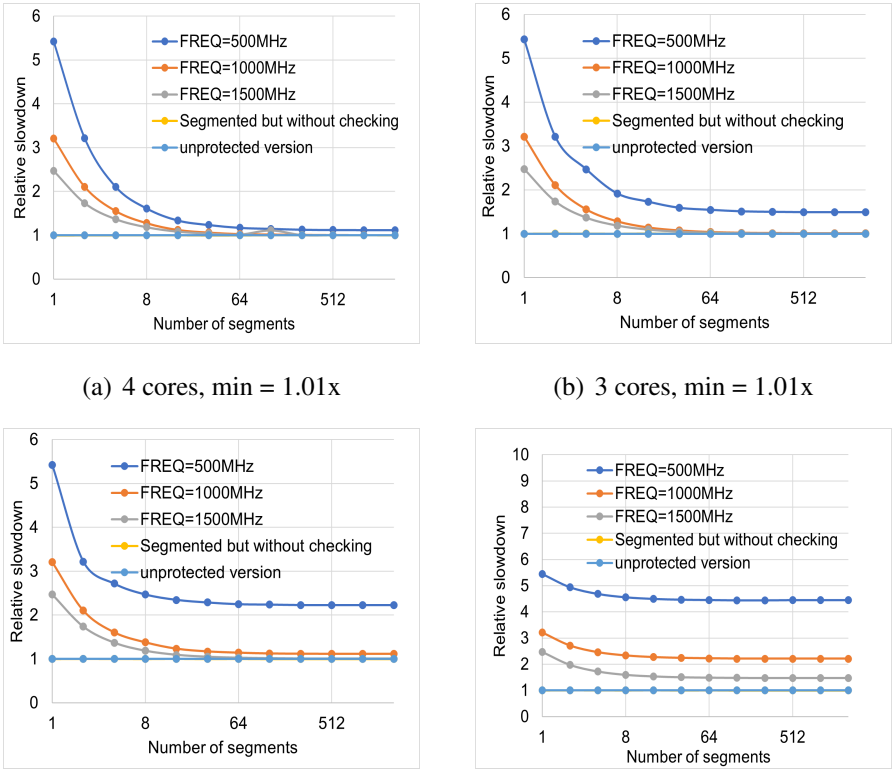
(c) 2 cores, min = 1.31x

(d) 1 core, min = 1.72x

Figure 5.6: RSA single block: Slowdown of fault tolerance on RSA public encryption for single block relative to the unprotected version. There is very little parallelism in RSA single block that can be exploited since it can only be divided into 9 segments at most. Due to this, although G is very small in this workload, the minimum slowdown that our approach can achieve is limited to 1.28x, which is quite high compared with CBC and Blackscholes.

The slowdown graph of RSA single block caters to the analysis we did in section 4.5.1, where we stated that the maximum number of segments in this workload is limited and this will prevent us from exploiting the parallelism. Thus, we came up with another workload which applies RSA public encryption on longer text without length limit. We expect to exploit much more parallelism in RSA long text(see section 5.2.6).

5.2.6 RSA long text



(a) 4 cores, min = 1.01x

(b) 3 cores, min = 1.01x

(c) 2 cores, min = 1.01x

(d) 1 core, min = 1.47x

Figure 5.7: RSA long text: Slowdown of fault tolerance on RSA public encryption for long text relative to the unprotected version. Due to the limitation of the granularity of the segmentation(in a single RSA encryption unit), the main thread can only be divided into 1024 segments at most, which limits the parallelism that could be exploited in this workload. The slowdown continues to decline as the number of segments grow but doesn't increase again, which indicates that the balance with minimum slowdown isn't achieved yet.

5.2.7 Summary

Based on the slowdown graphs on the above six workloads, we can find that our approach works especially well on both sequential and parallelizable workloads. However, there are two factors that may hinder us from exploiting the new parallelism: first is the cost of generating checker points(G), second is the maximum segments of a workload. Two examples are NAS IS and RSA single block, where G is too high or the maximum N is too small.

5.3 Verification of running time model

We proposed a running time model for fault tolerance on general programs in section 3.4. This model is theoretical and needs to be analysed in actual experiments. There are five core parameters in this model: G - the cost of generating a checker point in main thread, k - relative performance of the checker core relative to the main core for a specific workload, BASE: the running time of the unprotected algorithms, N: the number of segments in main thread, m: how many checker cores are run in parallel.

Once we have figured out the specific parameters for each workload, we can derive its running time in theory and explore the best configuration for parallel fault tolerance. This will be mainly based on the following equations proposed in section 3.4. For the two cases: idle and congested, the best N for exploiting parallelism is shown as below.

$$N = \begin{cases} \sqrt{\frac{BASE}{G} * k} & m \geq k \\ \sqrt{\frac{BASE}{(d+k*G)}} * m & m < k \end{cases} \tag{5.2}$$

For each workload, if we know the accurate values of G, BASE, k and d, we will be able to calculate the best number of segments in theory to exploit parallelism as well. We can also use it to verify the correctness of our model by monitoring the actual best number of segments in experiments. To achieve this, we have conducted experiments to calculate G and BASE as discussed in section 5.1. The other two parameters k and d could also be measured in actual experiments by seting timers. As such, we got the values of k, d, BASE and G for all the workloads and used them to verify our model. In the below analysis for verification, we will use N-T to represent the number of segments derived with formula, N-E is the actual result in experiments.

m	F(MHz)	k	BASE(s)	G(s)	d(s)	N-T	N-E
4	500	6.782	2.02	5E-6	0.00004	661.3	512
4	1000	3.5	2.02	5E-6	0.00004	1189.1	1024
4	1500	2.49	2.02	5E-6	0.00004	1003.0	1024

Table 5.1: CBC: The number of segments to achieve minimum slowdown of fault tolerance in theory and experiments. N-T is very close to the N-E in experiments. For the two different cases: congested and idle, our model can both explain them well and give reasonable results.

m	F(MHz)	k	BASE(s)	G(s)	d(s)	N-T	N-E
4	500	2.93	3.4594	0.3491	0.00004	5.38	8
4	1000	1.63	3.4594	0.3491	0.00004	4.01	4
4	1500	1.23	3.4594	0.3491	0.00004	3.49	4

Table 5.2: NAS-IS: The number of segments to achieve minimum slowdown of fault tolerance on NAS IS in theory and experiments. Compared with CBC, the number of segments that can achieve the minimum slowdown in IS is much smaller. This is because the cost of generating checker points is too high in this workload, which prevents us from exploiting the parallelism by spawning more segments.

m	F(MHz)	k	BASE(s)	G(s)	d(s)	N-T	N-E
4	500	7.95	2.92	0.0006	0.00004	98.6	128
4	1000	3.97	2.92	0.0006	0.00004	139.1	128
4	1500	2.65	2.92	0.0006	0.00004	113.6	128

Table 5.3: ZLIB: The number of segments to achieve minimum slowdown of fault tolerance on zlib in theory and experiments. The cost of generating a checker point in ZLIB is 0.0006, which is a medium value between that in CBC and IS. Its best number of segments in theory is also smaller than that in CBC but bigger than that in IS.

m	F(MHz)	k	BASE(s)	G(s)	d(s)	N-T	N-E
4	500	6.76	4.63	3E-6	0.00004	1108.57	1024
4	1000	3.43	4.63	3E-6	0.00004	2300.78	2048
4	1500	2.32	4.63	3E-6	0.00004	1892.22	2048

Table 5.4: Blackscholes: The number of segments to achieve minimum slowdown of fault tolerance on blackscholes in theory and experiments. It can be noticed that G in blackscholes is the smallest among the six workloads, thus the best number of segments is also much larger than that in other workloads. This is because lower cost of checker points means we could spawn more segments to exploit the parallelism.

m	F(MHz)	k	BASE(s)	G(s)	d(s)	N-T	N-E
4	500	4.58	0.0085	2E-5	0.00004	32.15	9(max)
4	1000	2.3	0.0085	2E-5	0.00004	31.26	9(max)
4	1500	1.53	0.0085	2E-5	0.00004	25.5	9(max)

Table 5.5: RSA single block: The number of segments to achieve minimum slowdown of fault tolerance on RSA single block in theory and experiments. Since BASE is very minor in this workload, the best number of segments is also very small although the cost of generating checker points is little.

m	F(MHz)	k	BASE(s)	G(s)	d(s)	N-T	N-E
4	500	4.42	9.97	1E-5	0.00004	1376.42	1024(max)
4	1000	2.2	9.97	1E-5	0.00004	1481.0	1024(max)
4	1500	1.47	9.97	1E-5	0.00004	1210.61	1024(max)

Table 5.6: RSA long text: The number of segments to achieve minimum slowdown of fault tolerance on RSA long text in theory and experiments. Note that the maximum number of segments is limited to 1024 in this workload due to the segmentation granularity. Based on the N-T derived from our running time model, we can also find that the balance to achieve minimum slowdown exceeds the limit of segments in this workload. If we break the main thread into blocks with finer granularity, it is expected to reduce the slowdown further.

Based on the above statics and analysis on different workloads, we can find that the theoretical values derived in our running time model is reasonable and caters to the results in actual experiments. For any workloads with given k, G, d and BASE, our model could predict the best number of segments(N-T) to exploit parallelism under two different cases: idle and congested. In addition, from the above examples, we can also know that higher cost of generating checker points usually leads to lower parallelism that can be exploited, which can be explained in our formular 5.2.

5.4 Comparison with traditional schemes

The intention of our project is to build a better scheme for fault tolerance based on two novel architectures, so we also compared our method with traditional software schemes like nZDC.(We didn't choose Comet since its public source code is incomplete.) nZDC was not targeted for Aarch32 and thus cannot be run on odroid XU4. It only works on Aarch64[14]. Thus, we also cross-compiled our programs using llvm-gcc in nZDC to run and compare them with parallel fault tolerant ones on Aarch64 platforms. We deployed the nZDC protected version and our approaches on a Raspberry pi 3B+ to collect running time statistics, then used them to calculate relative slowdown to the unprotected version(see figure 5.7). The cpu of pi 3B+ is ARM Cortex-A53, which is a quad-core processor with a frequency of 1.4GHz.

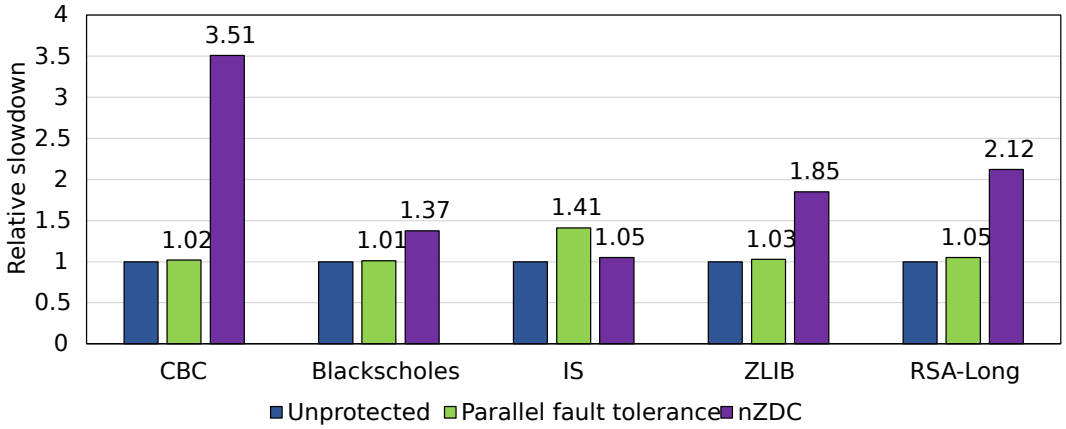


Figure 5.8: Slowdown of parallel error detection on different workloads compared with nZDC and the unprotected version. The unprotected version's slowdown is set to be 1 and in blue. Our approach's slowdown relative to the unprotected version is in green and the nZDC protected version's slowdown is in purple.

For different workloads, nZDC protected version is usually from 1.5x to 3x slower than the original version except for IS, while our approach can reduce the latency to 1.01x with appropriate configuration of parallelism. The geomean slowdown of nZDC for the above workloads is 1.82x, while ours is 1.09x. This shows the advantages of our approach over traditional software schemes like nZDC.

The IS workload was deliberately selected to show the cons of our approach. It is highly bound to the memory latency instead of the frequency of processors as most of work in IS is to access data rather than do computing. For nZDC or Comet, the repetition of compute or logging is easy to hide on an out-of-order processor because

the computation is memory bound. nZDC is only 1.05x slower in our experiments and Comet is 1.25x slower according to their paper. However, our in-order checker cores struggle with the latency hiding of the workload, and the checkpoints we take are expensive, hence why we show it as a worst case. For these memory-bound workloads, the high cost of generating checker points may prevent us from exploiting the new parallelism to improve overall performance.

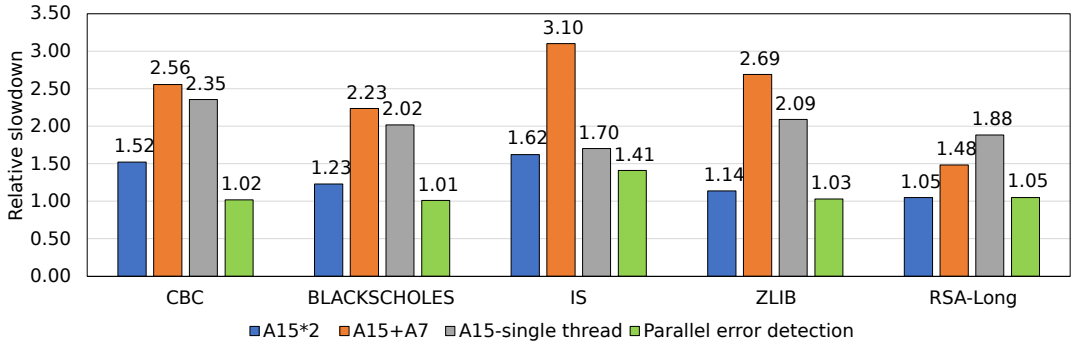


Figure 5.9: Slowdown of parallel error detection on different workloads compared with running twice on each thread(A15+A15 or A15+A7) and single thread(A15) on in-order processors.

Most other works[30, 14] implicitly suggest that the relevant comparison is to running the workload twice and comparing the results. It's intuitively expected that it will be 2x slowdown if run in the same thread, and 1x if offloaded (and so we either want to do better than that, or get very close and have more energy efficiency through parallelism). In reality, it will probably be a fair bit higher for several of these workloads. Although we don't have the source code of Comet, we slightly transformed our approach to run twice on each thread on two different cores to simulate the performance of Comet on in-order processors. We also tried to run the checking on the same thread to imitate nZDC.(see figure 5.9)

It shows that running checking on a separate thread(A15+A15) is from 1.05x to 1.5x slower than the unprotected version and running them on the same thread is from 1.70x to 2.35x slower. Even on IS we do considerably better in performance, and with extra parallelism (thus energy saving) from running on multiple A7s.

Chapter 6

Conclusions

6.1 Discussion

In this project, we have successfully built parallel fault tolerance on five different workloads using a pure software way without any custom hardware. This proves the feasibility of software parallel error detection and provides realistic examples for the promotion of the parallel fault-tolerant mechanism of general procedures in the future.

In regarding of the software parallel fault tolerance mechanism, we also proposed a theoretical model that can derive the running time and latency of general programs. With the model, we managed to derive the number of segments which exploits the parallelism best and achieves minimum slowdown. We also conducted actual experiments to compare and verify the results of our model. Experiments have proved that the slowdown curves on different workloads can be well in line with our theoretical model, and the experimental results are very close to our theoretical values.

We also compared the efficiency of our method with the traditional solution - nZDC to figure out its pros and cons. For most workloads, software parallel error detection could achieve almost the same speed as the unprotected version. For some special workloads with intensive memory accesses like IS, the high cost of generating checker points is the weakness of our approach on out-of-order cores. However, on in-order cores, experiments have shown that our approach is better in performance compared with running checking on a separate thread like Comet or the same thread like nZDC even on IS.

6.2 Limitations and future work

This project targets specific algorithms and we have to build fault tolerance on each program separately, which is very time-consuming and involves manually modifying the code. A more general framework of software parallel error detection could be developed in the future, which can automatically determine which kind of data to be copied or directly read without manually specifying. Since we have implemented parallel error detection on five different workloads, the experience we gained from this project can help develop a general parallel fault tolerance framework, which can automatically generate parallel error detection code on any given program. This may be based on the compiler support, which is similar to Comet[30] or nZDC[14], but will exploit new parallelism and apply finer checking schemes on data as described in section 3.3.4.

To generate parallel fault tolerance code at the stage of compiling, three components must be realized: automatic segmentation of main thread, classification of variables and integration of a thread pool. Segmentation of main thread will be based on the granularity given by users. Just divide a sequential loop into blocks, then insert checking code between them to spawn checker threads and transfer data. The thread pool is also easy to build, it is used as a container of the checking tasks. The maximum size of it can be adjusted when compiling. The most challenging task is to automatically classify all variables into three types as we described in figure 3.2, then apply different checking schemes on them: explicit checking or implicit checking. This may involve counting the number of writes to this variable in the entire program before and after checking, then do classification based on the rules described in section 3.3.4.

Bibliography

[1] Sam Ainsworth and Timothy M Jones. Parallel error detection using heterogeneous cores. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 338–349. IEEE, 2018.

[2] Sultan Almuhammadi and Ibraheem Al-Hejri. A comparative analysis of aes common modes of operation. In *2017 IEEE 30th Canadian conference on electrical and computer engineering (CCECE)*, pages 1–4. IEEE, 2017.

[3] Amin Ansari, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Necromancer: Enhancing system throughput by animating dead cores. *ACM SIGARCH Computer Architecture News*, 38(3):473–484, 2010.

[4] Todd M Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 196–207. IEEE, 1999.

[5] David H Bailey, Eric Barszcz, Leonardo Dagum, and Horst D Simon. Nas parallel benchmark results. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):43–51, 1993.

[6] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 170–177, 2003.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[8] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[9] Peter Clarke. How arm's cortex-a7 beats the a15. *EE Times*, 2013.

[10] Ádria Barros de Oliveira, Gennaro Severino Rodrigues, and Fernanda Lima Kastensmidt. Analyzing lockstep dual-core arm cortex-a9 soft error mitigation in freertos applications. In *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*, pages 84–89, 2017.

[11] Ádria Barros de Oliveira, Gennaro Severino Rodrigues, Fernanda Lima Kastensmidt, Nemitala Added, Eduardo LA Macchione, Vitor AP Aguiar, Nilberto H Medina, and Marcilei AG Silveira. Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors. *IEEE Transactions on Nuclear Science*, 65(8):1783–1790, 2018.

[12] Peter Deutsch. Rfc1951: Deflate compressed data format specification version 1.3, 1996.

[13] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC 1950, May, 1996.

[14] Moslem Didehban and Aviral Shrivastava. nzdc: A compiler technique for near zero silent data corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[15] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on aes. In *International Conference on Applied Cryptography and Network Security*, pages 293–306. Springer, 2003.

[16] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. Rsa-oaep is secure under the rsa assumption. In *Annual International Cryptology Conference*, pages 260–274. Springer, 2001.

[17] Thomas Grün and Mark A Hillebrand. Nas integer sort on multi-threaded shared memory machines. In *European Conference on Parallel Processing*, pages 999–1009. Springer, 1998.

[18] Marcus Hähnel and Hermann Härtig. Heterogeneity by the numbers: A study of the {ODROID} xu+ e big. little platform. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*, 2014.

[19] Peter H. Hochschild, Paul Jack Turner, Jeffrey C. Mogul, Rama Krishna Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Proc. 18th Workshop on Hot Topics in Operating Systems (HotOS 2021)*, 2021.

[20] Simon Holmbacka and Jörg Keller. Workload type-aware scheduling on big. little platforms. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 3–17. Springer, 2017.

[21] Kas Raygaputra Ilaga, Christy Atika Sari, and Eko Hari Rachmawanto. A high result for image security using crypto-stegano based on ecb mode and lsb encryption. *Journal of Applied Intelligent System*, 3(1):28–38, 2018.

[22] Xabier Iturbe, Balaji Venu, Emre Ozer, Jean-Luc Poupat, Gregoire Gimenez, and Hans-Ulrich Zurek. The arm triple core lock-step (tcls) processor. *ACM Transactions on Computer Systems (TOCS)*, 36(3):1–30, 2019.

[23] Jakob Jonsson and Burt Kaliski. Public-key cryptography standards (pkcs)# 1: Rsa cryptography specifications version 2.1. Technical report, RFC 3447, February, 2003.

[24] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, 10(2):124, 1985.

[25] Yun Zhang Jae W Lee and Nick P Johnson David I August. Daft: Decoupled acyclic fault tolerance. 2010.

[26] Aiguo Li and Bingrong Hong. Software implemented transient fault detection in space computer. *Aerospace science and technology*, 11(2-3):245–252, 2007.

[27] Yibei Ling, Tracy Mullen, and Xiaola Lin. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review*, 34(2):42–55, 2000.

[28] James D MacBeth and Larry J Merville. An empirical examination of the black-scholes call option pricing model. *The journal of finance*, 34(5):1173–1186, 1979.

[29] Evgeny Milanov. The rsa algorithm. *RSA laboratories*, pages 1–11, 2009.

[30] Konstantina Mitropoulou, Vasileios Porpodas, and Timothy M Jones. Comet: Communication-optimised multi-threaded error-detection technique. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10. IEEE, 2016.

[31] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th annual international symposium on computer architecture*, pages 99–110. IEEE, 2002.

[32] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[33] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-sensitive scheduling for smt processors, 2000.

[34] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture*, pages 291–302. IEEE, 2005.

[35] M Wasiur Rashid, Edwin J Tan, Michael C Huang, and David H Albonesi. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 315–325. IEEE, 2005.

[36] Steven K Reinhardt and Shubhendu S Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*, pages 25–36. IEEE, 2000.

[37] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift: Software implemented fault tolerance. In *International symposium on Code generation and optimization*, pages 243–254. IEEE, 2005.

[38] Eric Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*, pages 84–91. IEEE, 1999.

[39] Vivek Sarkar. Synchronization using counting semaphores. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 627–637, 1988.

[40] M Vaidehi and B Justus Rabi. Design and analysis of aes-cbc mode for high security applications. In *Second International Conference on Current Trends In Engineering and Technology-ICCTET 2014*, pages 499–502. IEEE, 2014.

[41] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 244–258. IEEE, 2007.

[42] Liangzhou Wang and Chaobin Wang. Producer-consumer model based thread pool design. In *Journal of Physics: Conference Series*, volume 1616, page 012073. IOP Publishing, 2020.

[43] Nicholas J Wang, Justin Quek, Todd M Rafacz, et al. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks, 2004*, pages 61–61. IEEE Computer Society, 2004.