POWER-AWARE RESILIENCE FOR EXTREME SCALE COMPUTING

A Dissertation
Submitted to
the Temple University Graduate Board

---

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

---

by
Anis Alazzawe
August 2019

Examining Committee Members:
Krishna Kant, Advisory Chair, Computer and Information Sciences
Justin Y. Shi, Computer and Information Sciences
Chiu C. Tan, Computer and Information Sciences
Albert Kim, External Member, Electrical and Computer Engineering

©

by

Anis Alazzawe

2019

All Rights Reserved

ABSTRACT

POWER-AWARE RESILIENCE FOR EXTREME SCALE COMPUTING

Anis Alazzawe

DOCTOR OF PHILOSOPHY

Temple University, August 2019

Dr. Krishna Kant, Chair

The increase in processing power provided by successive generations of high performance computing platforms has made it possible, to tackle a diverse range of large problems in many different fields, that would not have been feasible otherwise. Exascale computing is on the horizon and it brings with it unique opportunities and challenges. Applications running on exascale systems will run into many errors due to the vast number of components in these systems. Traditional recovery methods such as checkpointing alone will not be sufficient to allow these applications to finish execution in a reasonable amount of time, and in some instances they will not be able to finish execution at all. This is because the number of errors will occur so often that they are expected to occur during the recovery process itself. Two primary issues that need research to make running applications on exascale systems viable is methods to provide scalable resilience and managing energy consumption. Managing

the energy usage of a resilience method is vital because these systems will be a huge energy draw and energy usage is expected to the largest cost of running these systems.

The research path we have taken, to introduce an energy efficient resilience method for exascale systems, is as follows:

First, we introduced slicing as an energy efficient resilience method. In this phase of the research we presented a model of program structure and error propagation of data corruption and showed how slicing can be used to detect these errors. Slicing is a technique that can be used generate all the parts of a program, as an executable, that influence the computation of a given a variable. It is traditionally used for analysis in debugging, maintenance, testing, of software. Using this model we derive properties that show how slicing can be used to provide high confidence that a program has run without corruption errors. The results show that a high error detection can be obtained using only a small increase in power usage.

Second, we introduced Slice Swarms for high performance computing (HPC) application resilience. In this phase of the research we scaled slicing to HPC environments. We developed a model that would allow us to reason about the use of multiple slices in an HPC environment. We showed that using multiple slices would provide the ability to to detect more errors in applications that don't have extreme inter-dependencies among it's variables while requiring

only a nominal amount of extra energy to run. We also showed the best way to distribute these slices across the variables of the application, given specific energy constraints.

Finally, we tackled the challenge of providing energy efficient resilience to HPC applications with regular structure. The largest computing systems routinely run into silent data corruption (SDC) as part of its normal operation. The number of SDCs will increase drastically as computing systems approach the exascale mark, forcing a need to reconsider the resilience approach taken to counteract the effects of unmitigated data corruption errors. Yet any resilience method must be sensitive to both resource and energy requirements. HPC applications often have a regular structure that can be exploited for providing resilience more efficiently. We explore the propagation of data corruption errors caused in stencil computation, an iterative kernel with structured communication pattern that is found in a wide variety of scientific and engineering problems. The key insight, is that SDCs and corruption of data that they cause have localized impact in these types of applications and recovery does not require the use of every process to recompute the application state. We present a resilience mechanism, mimic replication, for resilience against SDC errors through dynamic reexecution of select processes. We then provide an analytical model that allows tradeoff between resource and energy consumption and resilience.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my dissertation director, Professor Krishna Kant, for the direction and strong support he has given me over the course of my research. He has consistently worked to provide me opportunities for growth. I would also like to thank the members of my dissertation committee for the fruitful discussions and valuable feedback.

I would like to acknowledge the many colleagues I had the opportunity to interact with over the years and especially members of L339 (362). They have enriched my time at Temple University. It has been fun working on the various projects, and I truly enjoyed the many discussions including those on the color of grass, rabbits, tones, and the very consequential "what is the purpose of life?"

Most importantly to my parents, family, and many friends who have been extremely supportive and patient, I would like to tell them: Alhamdulillah, it paid off!

YSOIML, NA, AHKMT

"... My Lord, enable me to be grateful for Your favor which You have bestowed upon me and my parents and to do righteousness of which You approve. And admit me by Your mercy into [the ranks of] Your righteous servants."

(The Holy Quran, 27:19)

TABLE OF CONTENTS

ABSTRACT iv

ACKNOWLEDGEMENT vii

DEDICATION viii

LIST OF FIGURES xii

LIST OF TABLES xiv

1 INTRODUCTION 1
- 1.1 Power-Aware Program Resilience Through Slicing 1
- 1.2 HPC Application Resilience Using Slice Swarms 2
- 1.3 Resilience by Exploiting Regular Structure in HPC Applications 3

2 POWER-AWARE RESILIENCE THROUGH SLICING 4
- 2.1 Introduction 4
  - 2.1.1 Faults and soft errors 5
  - 2.1.2 Program Slicing 7
  - 2.1.3 Resilience Through Program Slicing 9
- 2.2 Program and Fault Models 10
  - 2.2.1 Program Model 10
  - 2.2.2 Internals of a Function 11
  - 2.2.3 Fault Model 12
  - 2.2.4 Error propagation 13
  - 2.2.5 Types and effects of structure 14
- 2.3 Evaluation of Slicing Based Resilience 15
  - 2.3.1 Generation of Functions 17
  - 2.3.2 Generation of Programs 17
  - 2.3.3 Generation of Slices 18
  - 2.3.4 Fault Injection Method 21

2.3.5 Power management 22
2.3.6 Characterization 23
2.3.7 Experimental setup 24
2.3.8 Parameters 26
2.4 Results 28
2.5 Related Work 30

3 HPC APPLICATION RESILIENCE USING SLICE SWARMS 33
3.1 Introduction 33
3.2 Background 36
3.2.1 Program Slicing Based Resilience 36
3.2.2 System level error estimation 38
3.2.3 Energy consumption and slices 42
3.3 Model and Analysis 43
3.3.1 HPC processes and slices 44
3.3.2 Characterizing the recomputation time under single checkpoint 48
3.3.3 Optimizing energy usage under multiple slices 55
3.3.4 System level resilience and I/O handling 56
3.4 Method and Results 57
3.4.1 Experimental parameters 58
3.4.2 Detection of errors 59
3.4.3 Energy usage under different number of slices 61
3.4.4 Reliability of coverage 64
3.4.5 Time to completion 65
3.5 Related Work 67

4 MIMIC: FAST RECOVERY FROM DATA CORRUPTION ERRORS IN STENCIL COMPUTATIONS 70
4.1 Introduction 70
4.2 Definitions and Model 74
4.2.1 Preliminaries 74
4.2.2 Model 76
4.2.3 Recovery model 79
4.2.4 The cost of recovery 84
4.3 Architectural considerations 86
4.3.1 Lag and Latency 86
4.3.2 Number of detectors 87
4.3.3 Energy consumption of recovery 88
4.4 Evaluation 89
4.4.1 Recovery 91
4.4.2 Latency 93

4.4.3 Energy Consumption 94
4.5 Related Work 96
4.5.1 Detecting data corruption 96
4.5.2 Energy aware recovery 97
4.5.3 Process failure and replication 97
4.6 Conclusion 99

5 CONCLUSION **100**

REFERENCES **105**

LIST OF FIGURES

2.1 A program can be sliced on a variables such that the generated slice is a an executable subset of the program that returns the same value as calculated in the original program 7
2.2 Solid arrows represent calls and dashed arrows represent indirect dependencies. 12
2.3 A program slice to be performed on the return value of the C7 function. 16
2.4 The corresponding program slice on the program in figure 3.1 19
2.5 The energy needed to evaluate a slice in terms of the size of the slice. 23
2.6 The fraction of errors detected as a function of the slice to program size. 25
2.7 The fraction of errors detected as a function of the extra energy needed under two different p-state sets. 27

3.1 A program slice to be performed on the return value of the C7 function. 36
3.2 The corresponding program slice on the program in Fig. 3.1 . 37
3.3 Architecture layout of slice placement 44
3.4 Total execution time for simple application with an error at slice point 50
3.5 A process with three corresponding slices on variables v_2, v_5, and v_6. Solid circles represent the variables that are required by the slice. 54
3.6 The percentage of errors detected when the energy usage of the slices is constrained to 60% 59
3.7 The percentage of errors detected when the energy usage of the slices is constrained to 60% and the dependence between functions is low 60

3.8 The percentage of errors detected when the energy usage of the slices is constrained to 60% and the dependence between functions is high . . . 60
3.9 Error detection under 20% energy constraints . . . 62
3.10 Error detection under 40% energy constraints . . . 62
3.11 Error detection under 60% energy constraints . . . 63
3.12 Error detection under 80% energy constraints . . . 63
3.13 10^3 core dedicated to application execution . . . 65
3.14 10^4 core dedicated to application execution . . . 66
3.15 Time to completion . . . 67

4.1 Seven processes that can only communicate with its neighbor. Knowing which process detects a corruption can be used to limit the possible time and sources of that error. The red in the left figure represents the origin of the corruption and the orange is the processes that can detect it. The green are those processes that can be originator of the corruption. . . . 80
4.2 The total recovery cost given a system error rate of 0.01 for Mimics running at 10%, 30%, and 70% frequency . . . 90
4.3 The total recovery cost given a system error rate of 0.001 . . . 90
4.4 The recovery time as a function of the detection interval given 0.01 errors/second . . . 91
4.5 Energy usage of the mimics normalized by energy required by CR for varying detection interval for a system error rate of 0.001 errors/second . . . 95
4.6 For a higher error rate of 0.01 errors/second the system using mimics require less energy . . . 96

LIST OF TABLES

2.1 P-State voltage, frequency pairs for two CPUs. 28
3.1 Voltage/frequency pairs for the P-State 58
4.1 Configuration 92

CHAPTER 1

INTRODUCTION

1.1 Power-Aware Program Resilience Through Slicing

Resilience against soft errors in software is a growing research concern and the trend is expected to continue as the density of transistors increases. In this chapter we propose a power-aware method to resilience, using program slicing as a mechanism to detect errors in running executables and to provide confidence in the result of program outputs.

We present a model of program structure and fault propagation. Using the model we derive properties that show how slicing can be used to ensure that parts of a program ran without errors. From this we show how the overall resilience of a program can be improved. We test this method in a simulated

environment and show that a small increase in power usage can lead to better resilience of programs.

1.2 HPC Application Resilience Using Slice Swarms

Resilience in High Performance Computing (HPC) is a constraining factor for bringing applications to the upcoming exascale systems. Resilience techniques must be able to scale to handle the increasing number of expected errors in an energy efficient manner. Since the purpose of running applications on HPC systems is to perform large scale computations as quick as possible, resilience methods should not add a large delay to the time to completion of the application.

In this chapter we introduce a novel technique to detect and recover from transient errors in HPC applications. One of the features of our technique is that the energy budget allocated to resilience can be adjusted depending on the operator's resilience needs. For example, on synthetic data, the technique can detect about 50% of transient errors while only using 20% of the dynamic energy required for running the application. For a 60% energy budget, an application that uses 10k cores and takes 128 hours to run, will only require 10% longer to complete.

1.3 Resilience by Exploiting Regular Structure in HPC Applications

One commons class of HPC applications is stencil computation. These applications are composed of processes which compute kernels and have predictable communication patterns among these kernels. Assuming that we have acceptance tests for these applications, we can speed up the recovery of the application. This is done by exploiting the fact that the propagation of errors is limited by the communication patterns.

To this end we setup up companion processes to each of the primary processes. The purpose of these companion processes is to speed in the recovery of the application when an error is detected. These companion processes run at a lower frequency than the primary processes which reduces their energy consumption. When an error is detected by means of an acceptance test, a select number of these companion processes run at the maximum frequency. The number of these processes can be bounded based on the structure of the stencil computation.

CHAPTER 2

POWER-AWARE RESILIENCE THROUGH SLICING

2.1 Introduction

The incidence of soft errors [10, 46] in semiconductor technology continues to rise and is already becoming very challenging to deal with in HPC environments. In the past, the primary issue was shrinking feature size of the chips which leads to effects such as alpha particle strikes. While such effects continue, there are two other factors that further increase the soft error rates. One is the need to continually downscale the switching voltage in order to

contain the power consumption of the chips. This brings additional switching reliability issues that may not always be detected and thereby lead to system failure. The other is the increasing variability in transistor and gate parameters due to lithographic difficulties, which in turn may overstress certain parts of logic and thus lead to errors. In this paper, we explore a program-slicing based approach to detecting and correcting soft errors without a substantial increase in the power budget. We show that the approach can complement existing techniques in improving the resilience of HPC systems. To the best of our knowledge, this is the first paper that exploits slicing for power-aware resilience.

2.1.1 Faults and soft errors

The pressure for ever smaller components has two effects on the components that requires resilience measures to deal with soft errors. The first is that having the electronic components reach sizes that can be measured in the width have atoms leads to physical anomalies such as quantum tunneling. The second is that the manufacturing process for these smaller components leads to irregularities between the components such as large variability between cores or memory banks on the same system.

Both of these issues are problematic in terms of resilience. While the first issue has been well known the second is becoming quite serious because

of lithographic challenges, as discussed for the current 14nm technology in [57, 12]. The variability typically has close to Gaussian distribution with the standard deviation about 5% of the mean. In a billion transistor chip, this means that millions of transistors could have rather large deviations from the mean. If these transistors are not driven properly, they could result in errors that may manifest at logic, subsystem, or software level.

The need for energy efficiency dictated by unsustainable power densities has led to another factor in the increase in soft errors. Since the active power consumption varies as square of the voltage, one approach to reduce energy usage is by using lower operating voltage. The operating voltage of upcoming chips will be either close to the threshold or below threshold leading to the so-called near-threshold and sub-threshold computing. In this regime, errors are unavoidable and are traditionally handled at the circuit level through the detection of violation of timing constraints.

The more established methods [26] for detecting the violation of timing constraints is to retry operations that violate them. Newer methods move towards greater asynchrony by stalling progress when signals don't arrive in time and therefore would cause a timing error [30, 35]. Other methods rely on dynamic control of voltage to handle errors, such as quick boost to voltage to handle slow signals [40]. Since the power consumption is proportional to the square of the voltage, these methods can achieve huge power savings in spite

```
//program                //slice              //slice
main()                   new_Top1()           new_Top2()
{                        {                    {
 v1 = calc1();            v1 = calc1();        v1 = calc1();
 v2 = calc2(v1);          return v1;           v2 = calc2(v1);
 v3 = calc3();           }                     return v2;
 result = v1+v2+v3;                           }
 return result;
}
```

Figure 2.1: A program can be sliced on a variables such that the generated slice is a an executable subset of the program that returns the same value as calculated in the original program

of the additional complexities associated with handling the errors.

With the increasing sophistication of the resilience methods, such as moving from the closed-loop method like Razor to open-loop methods such as Bubble-Razor or Blade, and with the the increase in the process variations, there will be a corresponding rise in likelihood that soft errors will not be caught or properly handled. This may either result in silent errors or exceptions raised to the software.

2.1.2 Program Slicing

Program slicing [59, 54] has a long history in software engineering. At its essence, program slicing deals with isolating a part of a program for fur-

ther analysis. Program slicing has a wide variety of applications including debugging, maintenance, testing, and giving insights to the developer.

Categorization of slicing falls under two broad forms, static and dynamic slicing. Static slices are slices generated at compile time, while dynamic slices are slices that take a specific execution context and generate slices based on the resulting execution graph. For the rest of this paper we will only consider static slices.

Static slices can be defined by a 2-tuple composed of a given program point p and a variable v to slice on. The type of slices that are generated are either backward slices or forward slices. Backward slices are runnable programs that include all the instructions that affect the selected (p, v). Forward slices generated on (p, v) contain those instructions that are affected by variable v at point p, and are not usually executable. In this paper we use backward slicing to generate executables from a program such that result will be a variable v. As we develop the program and fault models, it may seem that limiting the slicing process to a single variable is restrictive. Ideally, a slice should be built using a set of related variables that collectively define some crucial aspect of the function and results in a small slice size. However, finding such a set can be very challenging and beyond the scope of this paper.

2.1.3 Resilience Through Program Slicing

The addition of resilience properties to a system must be evaluated against the overhead of achieving such resilience. For large HPC applications, typically running on a large number of cores, the primary concern in terms of overhead is power consumption rather than computing power. In this paper we propose to use program slicing as a resilience method where the slices can make use of dynamic voltage and frequency scaling (DVFS) for energy management.

There are two ways that program slicing can be useful in providing resilient properties to a system. At the software level, we can guide slicing based on the importance of variables computed by the application. That is we can view the correctness of certain variables as being more important than other variables. On the hardware level, the program can be sliced on variables that excersize all the challenging timing situations, and if these variables match those computed by the program as output then we can be fairly confident that we have avoided silent errors.

In this paper we investigate the ability to detect and correct soft errors in software by a redundant execution of program slices which are expected to require less energy than running a redundant execution of the entire program. Of course, executing only a slice provides less assurance regarding the correctness of a program, and we wish to study the correctness vs. computation time and energy tradeoff. Section 2 introduces some definitions and the

model that we develop. Section 3 discusses the methods used for generating different parts of the experiment and the setup of the simulation environment for running the experiments. Section 4 will present the results and provides some interpretations. In section 5 we compare our work to previous research on software based approaches for dealing with soft errors. Finally, we will put the results in context, discuss the tension between resilience and energy requirement, and discuss future work.

2.2 Program and Fault Models

2.2.1 Program Model

The programs we are concerned with are those that can be easily converted into a functional form. That is, there are no shared states between functions and they have no side effects. A realization of function f is a function $s : \mathcal{I} \rightarrow \mathcal{O}$ where $\mathcal{I}$ is a set of input variables to the function and $\mathcal{O}$ is the set of output variables. The distinction between f and s is that multiple s's may represent the same f but are run in different parts of the program hierarchy. For the rest of this paper we will refer to the realization of functions as functions.

A program S can be viewed as a non-empty set of functions $\{s_1, ..., s_n\}$. For any program S, we define the children of a function to be all the functions it calls directly. We define $Ch : S \rightarrow \mathcal{P}(S)$ where $\mathcal{P}$ is the powerset, to be the

mapping that will take an element of S and will return the children of that element. We also define the mapping Top : S→ $Boolean$ in terms of Ch.

$$Top(x) = \begin{cases} True, & \text{if } x \notin Ch(s)\, \forall s \in S \\ False, & otherwise \end{cases}$$

Under this program model, for any program S and any two functions $s_i \in S$ and $s_j \in S$ where $i \neq j$, there is a constraint that $Ch(s_i) \cap Ch(s_j) = \emptyset$. That is a function can only be a included in the children set of at most one function.

2.2.2 Internals of a Function

In the previous section we gave the definition of a function. In this section we define the internal structure of a function. Each function takes a set of variables $\mathcal{I}$ as parameters and returns a set of variables O. The total bit size of these sets of variables are b_i an b_o respectively.

A function also has a number of instructions to perform its computation, which for the convenience of analysis are split between the preamble and postamble, w_{pre} and w_{post} respectively. The preamble represents the instructions that influence the parameters of all the children functions. The postamble are the instructions that do not affect the parameters of the child functions and only affect the output of the current function.

For any two functions $s_i \in S$ and $s_j \in S$ where $i \neq j$ if $s_j \in Ch(s_i)$, there may be associated functions that are also elements of $Ch(s_i)$ representing an

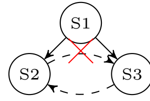

Figure 2.2: Solid arrows represent calls and dashed arrows represent indirect dependencies.

indirect dependence through s_i for the input parameters of s_j such that there is no circular dependencies [Fig. 2.2] among the elements of $Ch(s_i)$. The maximum size of the dependency list is $|Ch(s_i)|-1$. The structure of a program function T, can be represented by an 8-tuple $(b_i, \mathcal{I}, b_o, \mathcal{O}, w_{pre}, w_{post}, \{d_c \subset Ch(.)|c \in Ch(.)\}, Ch(.))$ where b_i is the total bit size of the input variables, $\mathcal{I}$ is the set of input variables, b_o is the total bit size of the output variables, w_{pre} and w_{post} are the number of instructions in the preamble and postamble respectively, d_c is the indirect dependency set for each child function, and $Ch(.)$ is the set of child functions.

2.2.3 Fault Model

The errors we are interested in are not caused by software bugs or persistent hardware faults. Rather the fault model we are assuming is one where the result of a computation can have random errors, such as those caused by manufacturing defects, QM tunneling, process variations, and timing issues.

Let $\mathcal{G} : S \rightarrow Boolean$ be a mapping that takes a function and returns whether there will be any soft errors in the function if is executed in the

current context. For any $s \in S$ we say $P({}^{+}\mathcal{G}(s))$ represents the probability that the function s was computed without any errors given the execution of its children. Furthermore, given $s_i \in Ch(s)$, $P({}^{+}\mathcal{G}(s)|\neg\mathcal{G}(s_i))$ can be > 0, and for all computations $P({}^{+}\mathcal{G}(s)|\neg\mathcal{G}(s_i)) \leq P({}^{+}\mathcal{G}(s)|{}^{+}\mathcal{G}(s_i))$. That is the probability of successfully computing the correct $T_s.\mathcal{O}$ for a function $s \in S$ given such that one of its children has some soft error can be bigger than zero in some circumstances but will always be less than or equal to performing the computation when no soft errors occurred.

For the rest of this paper, we take the pessimistic view. That is we take probability of success of computation of an s given a error in the calculation for an element $Ch(s)$ will be equal to 0. That is when a child computation is wrong we can no longer use the structure of dependencies to eliminate a lot of factors. To make the analysis tractable we will take $P({}^{+}\mathcal{G}(s)|\neg\mathcal{G}(s_i)) = 0$ where $s_i \in Ch(s)$ for the rest of this paper.

2.2.4 Error propagation

The correct execution of any function s can be measured by comparing $T_s.\mathcal{O}$ with the expected output of that function. $T_s.\mathcal{O}$ is dependent on the correctness of $\forall_{s_i \in Ch(s)} T_{s_i}.\mathcal{O}$, $T_s.w_{pre}$, and $T_s.w_{post}$. It is also dependent on other functions that are in the indirect dependency of the parent function. That is if any of the functions it depends on have errors, then $T_s.\mathcal{O}$ will have

an error. The probability of success of the performing the computation defined by s is $P(T_s.\mathcal{O}\, computed\, correctly) = P(^+\mathcal{G}(s)) * \prod_{s_i \in dep(s)} P(^+\mathcal{G}(s_i)) * \prod_{s_k \in Ch(s)} P(^+\mathcal{G}(s_k))$. According to our assumption, there is a zero probability the computation would be successful when the computation any child functions, dependent functions, or the preamble instructions of the containing function contains any errors.

2.2.5 Types and effects of structure

Analysis of the propagation of soft errors through different program structures as laid out in the model reveals that the program structure has a large impact on impact of the errors. This impact can be measured both in terms of the number of variables affected by the errors but also how diffuse the effects are and where in the program the errors will propagate. So depending on the structure, some errors may affect a large percentage of the program in terms of the number of variables that will have erroneous values or even affect large number of functions but a relatively small number of variables.

From this analysis, we expect that shallow programs which have a high interdependencies between functions to have errors propagate such that both the number of variables affected and the number of functions having some error to be high. On the other hand programs that are deep but don't have many dependencies between the functions will likely have any occurrence of

soft errors be isolated to the branch where the soft error occurred.

Now that we have a model that describes the structure and composition of a program, how errors propagate, and the way structure affects error propagation we will describe precisely how slices are generated in that model. We will use these slices to provide resilience by performing an acceptance test on the result of running the slice and the variable in the program that was used as the slicing target. This will allow us to detect any errors that are generated in the program that has coverage in the slice. A small slice can be run at a lower frequency and still complete with or before the program, so the power required to run a slice or even several slices may be attractive compared to the fraction of errors detected.

2.3 Evaluation of Slicing Based Resilience

A thorough evaluation of the benefits of slicing for error detection and resilient execution requires slicing on a large number of real application instances. However, this is impractical not only in terms of effort involved but also in terms of its usefulness and applicability. First our development currently concerns only functional programs without side effects and does not capture the complexities of real programs written in mainstream languages like `C++`. Second, real programs do not allow for control over various parameters of interest (e.g. preamble, dependencies, correctness checks, etc.) Consequently,

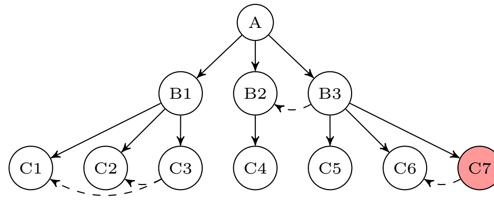

Figure 2.3: A program slice to be performed on the return value of the C7 function.

we evaluate the approach in this paper via simulation only. We purpose built a simulator that allows us to flexibly define and evaluate programs of the type shown in Fig. 3. By varying various parameters, such a simulator allows us to explore a very large space that would be impossible to explore with real programs.

In the following we describe how individual functions are generated. We show how these functions are combined to generate programs. We then give an algorithm that is used to generate the slices and show how a slice on a variable in a program gets turned into a standalone program. The method of fault injection is then detailed along with how the error is propagated throughout the program. We then describe how p-states are used for power management and the minimum frequency that slices need to run at. We explain the potential effect the structure of the program has on both the usefulness of a slice to detect errors and on the p-state required to run the slice. Finally we describe the experimental setup and the parameters that were used to run the experiments.

2.3.1 Generation of Functions

Each time a function s is generated, instance values are given to the tuple T. If $Ch(s) = \emptyset$, $T_s.b_i = 0$. Also both $T_s.w_{pre}$ and $T_s.w_{post}$ are given relative sizes representing the size of instructions in these parts of the function respectively. For the experiments we have ranged these sizes to 10 possible incremental values varying randomly. These functions do not exist in isolation and cannot perform its computation without being part of a larger program.

2.3.2 Generation of Programs

As illustrated in Fig. 3.1, two prominent features of a program structure are the direct calls that the functions need to make to perform their computation and the indirect dependencies as results propagate through variables from one function to another.

To generate a program, we continuously add a generated function to S. That is $S = \{s_i\} \cup S$, where the $Ch(s_i) \subset S$. We must generate s_i such that $T_{s_i}.Ch(.) = \emptyset$ when $S = \emptyset$. When $|S| \geq 0$, s_i will be generated such that $T_s[dep \in d_i \, with \, probability \, P \, if \, dep \in Ch(s)]$. The function generation process will not stop unless there exists $s_j \in S' \subseteq S$ where $Top(s_j) = True$.

2.3.3 Generation of Slices

Fig. 3.2 illustrates a slice for the program in Fig. 3 generated on some variable in the $C7$ function. In this specific case whether the variable is part of the preamble or postamble will not change the overall program structure of the slice. $T_{C7}.w_{pre}$ has an indirect relationship on $T_{C6}.\mathcal{O}$ through $B3$ so it gets pulled into the slice unmodified. $C7$ is called from the context of the $B3$ function, so a modified version $B3$ is generated without $T_{B3}.w_{post}$ and $T_{B3}.\mathcal{O}$ is set to be the variable of the output of $C7$. This allows the call to $C5$ and any variables that gets modified by the results of $C5$ and any calculations that is needed solely for input to $C5$, to be removed from $B3$.

A similar process happens when we generate the code that flows into the modified $B3$. Since $B2$ gets generated unmodified that means $C4$ is also generated unmodified. That leaves the function A, which needs to be modified so that it will be the $Top(.)$ of the generated slice. The modified function A has all instructions in $T_A.w_{post}$ removed and $T_A.\mathcal{O}$ is set to the returned variable of $B3$. Thus far there has been no dependency on the $B1$ function, neither it nor its children are generated in the slice. Since there was a dependency on $B1$ in the original A function all the instructions that feed solely into the input of $B1$ or only used to perform the final program out are removed. As we can see a sizable portion of the program may be removed in the generated slice.

To make a slice out of a program, we introduce a cut operator Γ that for

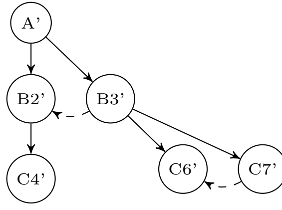

Figure 2.4: The corresponding program slice on the program in figure 3.1

a given program S, takes a function $s \in S$ and a variable v that is either in $T_s.w_{pre}$, $T_s.w_{post}$, or is a result of a child computation and then returns a new function from s, namely s', and any $Ch(s')$. The way Γ constructs s' is if v is an in $T_s.w_{post}$ and $\{v\} = T_s.\mathcal{O}$ then s' is an exact copy of s constructed. If $\{v\}! = T_s.\mathcal{O}$ but is in $T_s.w_{post}$ then s' is an exact copy of s with the modification that $T_{s'}.\mathcal{O} = \{v\}$ and $T_{s'}.w_{post}$ is modified so that all instructions after the last modification to v are removed. On the other hand if v is only in $T_s.w_{pre}$ then s' is constructed as a copy of s with no child, no postamble instructions, a pruned preamble up to the last modification of v with all instructions not contributing to the modifications of v in the preamble removed, and $T_{s'}.\mathcal{O} = \{v\}$. Finally if v is a result of a child computation, then the operation of cut becomes more involved. $s^{'}$ will be constructed as a copy of s with the following modification. The set of child elements is constructed as follows. $T_{s'}.Ch(.)$ initially only includes the function that modifies v. From there all the dependencies of $T_{s'}.Ch(.)$ included in $T_{s'}.Ch(.)$ and the appropriate dependency set is updated. This is done continuously till no more new elements are added to $T_{s'}.Ch(.)$.

The process for generating a slice for a given Program S, some function s, and on some variable v is given in Algorithm 1. By the end of the process we will have an executable slice S'. The generated slice S' will have exactly one $Top(.)$ function. This is the function that will be the entry point into the slice. The slice can then be taken and run on a separate CPU core.

The $Top(.)$ element of the slice S' will return the variable that the program was sliced on. Once we have the computation from the slice, we use that to detect any soft errors that may occur or propagate in the primary program. We do this by first recording the result of the slice and the full program in memory or on disk, and then comparing them for discrepancies. A discrepancy implies an error in some overlapping instructions between the slice and the primary program. For simplicity, we do not consider the actual cost of result matching, whether in terms of time or energy since the variables themselves can range in size from program to program and the time to make a comparison is linear in the size of the variables.

Proposition 2.1. *Any two unique slices of a program S must have instructions, performing the same computation, duplicated between them.*

This proposition tells us that any time we have more than one slice of a particular program, and the set of slices provide coverage of all variable modifications, then the size of the instructions in our slices will be larger than the original program.

Algorithm 1 Slicing algorithm

```
SLICE(S:Program, s∈S, v:variable):
  S′ = {}
  s_new = s
  v_new = v
  While |S′| increases:
    s′,Ch(s′)=Γ(S, s_new, v_new)
    S′ = S′ ∪ s′ ∪ Ch(s′)
    s′_temp = s∈S such that s_new ∈ Ch(s)
    v_new = var in s′_temp result of s_new()
    s_new = s′_temp
  return S′
```

2.3.4 Fault Injection Method

There are two ways a soft error can be introduced into a function, as explained in our model. The first way is that random bit-flips happen due to the mechanisms we describe previously. The chance of this happening to any sections of code will be proportional to the size of that section. The other way is through error propagation. For the simulation described below a soft error is always introduced exactly once. This causes an error in a variable that is either in the preamble or postamble of a single function. The probability of

variable having an error is exactly the size of the section of code computing the variable divided by the total program size.

The introduced error is then propagated as follows. If the faulty variable is in the postamble then a series of $propagate - up$ operations are performed. This operation will invalidate the postamble of the parent function and totally invalidate any sibling functions that indirectly depends on the output of the function in question. This operation will happen recursively on the parents until the Top is reached. If the faulty variable is in the preamble of a given function then in addition to the $propagate-up$ operation and $propagate-down$ operation will be performed. This operation will invalidate all the variables of the child functions and this will happen recursively till there are no more child functions to propagate the errors to.

2.3.5 Power management

What makes slicing attractive for resilience is that it gives us the ability to run an executable on another CPU core using DVFS. Since the dynamic $Power \propto k * V^2 * f$ where V is the voltage, f is the frequency and k is a constant; any decrease in voltage will have a squared effect on the dynamic power used. We will be basing the experiments on the ability to put a core into a certain p-state. Now of course we can't just put that core to run on the highest p-state, thus using the lowest (voltage, frequency) for every slice

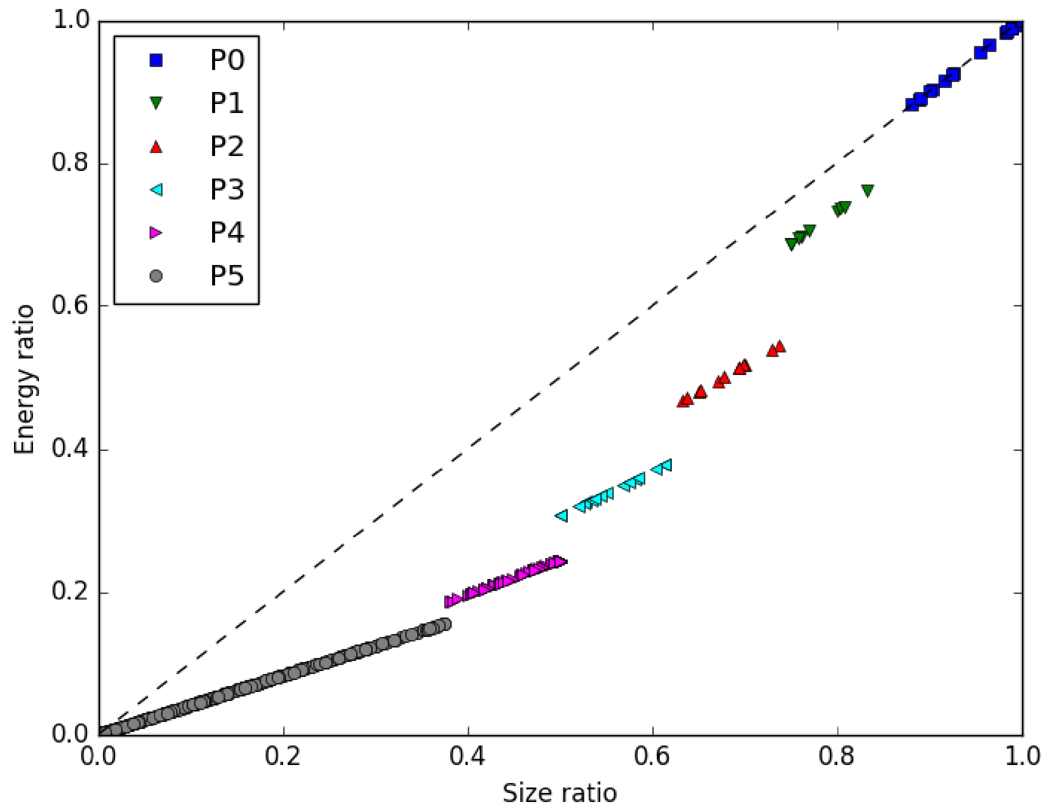

Figure 2.5: The energy needed to evaluate a slice in terms of the size of the slice.

taken of a program, because we need to ensure that slice finishes execution with the primary program or beforehand. This is to ensure that the primary program and slice finish at approximately the same time thereby enabling the acceptance checks without further delays.

2.3.6 Characterization

There is an inherent tension between the usefulness of slicing in the detection of errors, the complexity of the program structure, and the energy

required to run the slice. For example if full program coverage was wanted, should one slice be created representing the full program or should several slices be created with the hope that we can run them at a higher p-state? Due to proposition 1, we know that taking several smaller slices that provide full coverage of the program will by necessity have redundant execution of code shared by the slices. There is no one answer to this question, as it depends on the program structure and the available values for the p-states of a given core.

For programs that require a higher resilience, the set of slices need to cover a larger percentage of the variables in the primary program. This requirement for coverage will either mean that more slices are needed or a larger slice can be used. For programs with a lot of independent functions many slices will give the option of running the slices on several cores many of which can be run at a higher p-state. In the cases where the program has many dependencies then a smaller number of larger slices need to be generated. These slices may need to run at a lower p-state so as to finish in time for the acceptance test when the primary program is ready.

2.3.7 Experimental setup

We built a simulator to test the efficacy of using program slicing for power-aware fault tolerance from two fundamental views. The first is that if we can indeed get an energy saving by slicing and running the slice at a higher

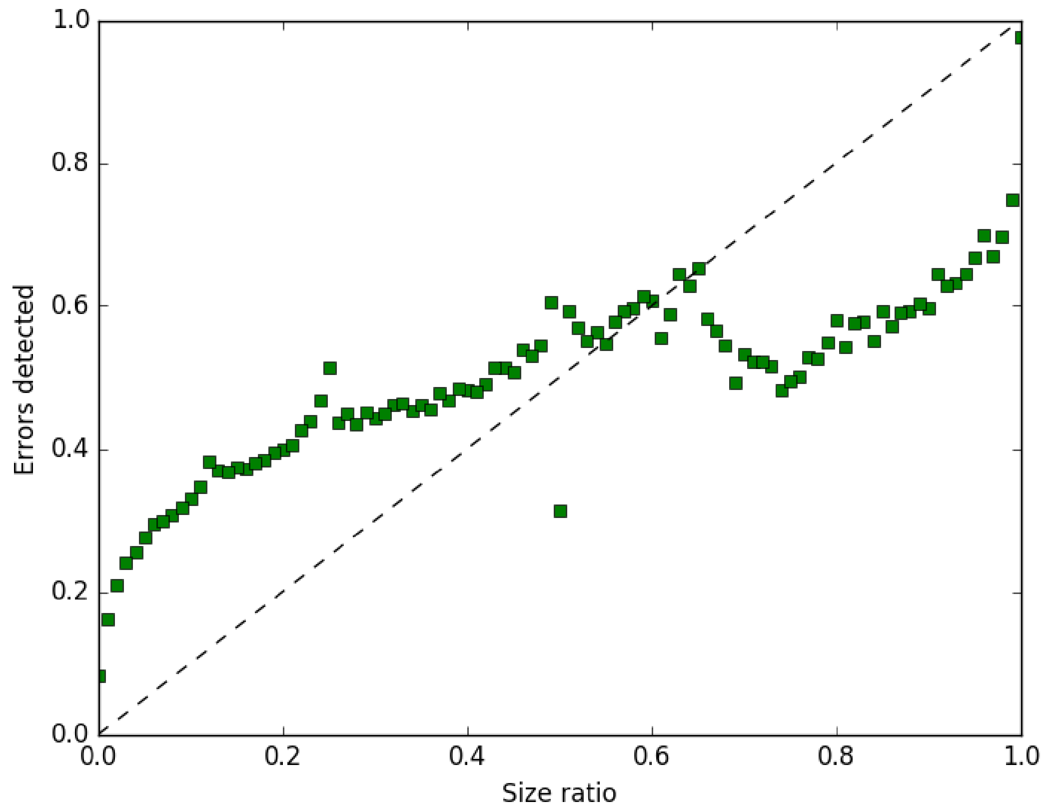

Figure 2.6: The fraction of errors detected as a function of the slice to program size.

p-state without sacrificing the time it takes for an application to performs its computation, yielding significant energy savings. The second is that if a significant percentage of soft errors can be detected and corrected without requiring a significant need for an increase in energy needs. The simulator was built on top of SimPy [3], which is framework for building process-based discrete-event simulations, using the generation methods described earlier in this section.

For the first set of experiments, a set of random ¡function, variable¿ pairs

are chosen for generating slices. The program slice starts as soon as the original program starts because it is not beholden to any data generated by the original program. The simulated core on which the slice is run is set to the maximum p-state value such that the slice finishes executing within the time it takes to finish executing the primary program.

For the second set of experiments, a slice is generated and run at a p-state such that it finishes with or before the primary program. Then errors are injected and propagated in the primary program as we previously specified. If the slice was done on postamble variable and the primary program has an error, whether it was propagated or originated, in any variable of the corresponding function then the error will be detected. On the other hand if the slice was performed on a variable in the preamble variable, then the slice will only be able to detect errors that originated from bad input variables or from errors that were injected into the preamble of the corresponding function.

2.3.8 Parameters

The experiments depend on several parameters. For the program creation we randomly added between 0 and 5 children to every function with a uniform probability and the depth with we any branch can take is also uniformly random, taking a value between 1 and 5 functions deep. The dependencies between a function and its sibling functions that preceded it, were generated

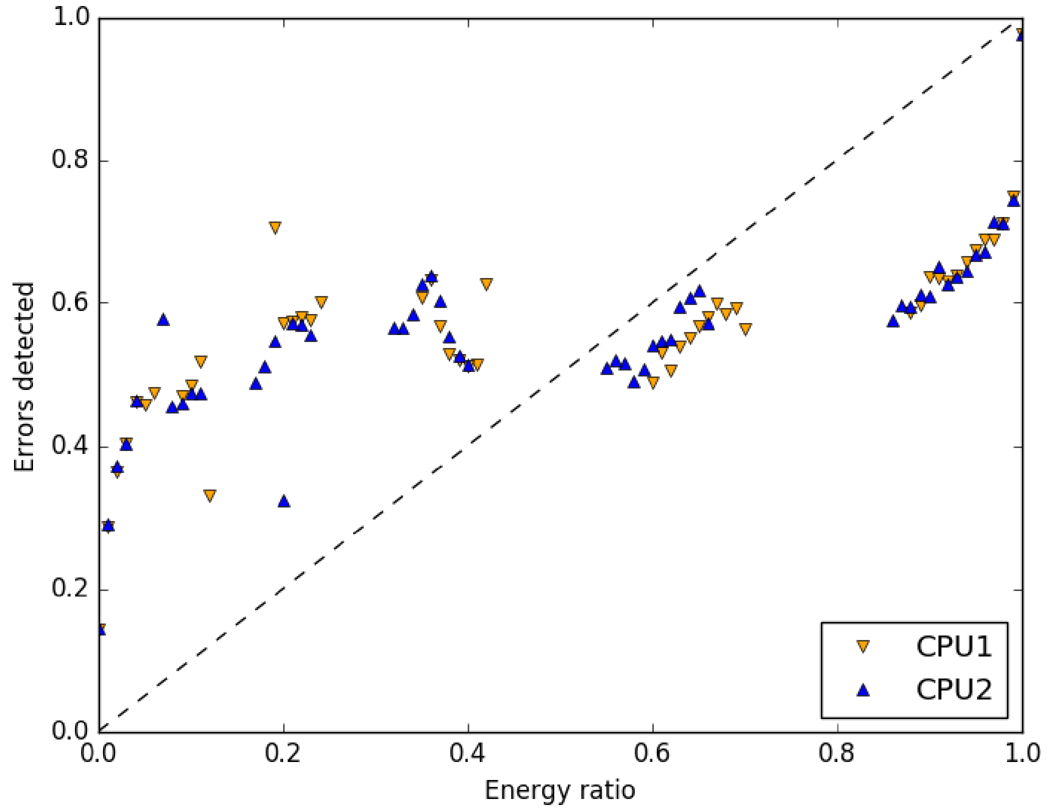

Figure 2.7: The fraction of errors detected as a function of the extra energy needed under two different p-state sets.

with 50% probability.

We run the simulation on two different p-states sets. The first (CPU1) set of p-states is represented by the p-states of an Intel Core i7 Nehalem [1]. The second (CPU2) set of p-states is represented by an Intel Haswell 4770K [2]. Table 1 gives the voltage and frequency for each of the six possible p-states for both CPUs.

P-State	CPU1 (V, GHz)	CPU2 (V, GHz)
P0	(1.484, 1,6)	(1.012, 3.5)
P1	(1.420, 1.4)	(0.958, 3)
P2	(1.276, 1.2)	(0.899, 2.5)
P3	(1.164, 1)	(0.845, 2)
P4	(1.036, 0.8)	(0.791, 1.5)
P5	(0.956, 0.6)	(0.737, 1)

Table 2.1: P-State voltage, frequency pairs for two CPUs.

2.4 Results

The first set of experiments show that there is a real and measurable difference from running a slice at a higher p-state. The experiments show [Fig. 2.5] that slices running at higher p-state values have slopes that puts it well below the energy used if they were running at the lowest p-state. Based on this we can conclude that if a slice is on the border line between between different p-states, then there is a large energy saving potential in taking a slightly smaller slice of the same set of functions. This makes slicing as a resilience method an attractive option where the energy budget requires adjustment such that a tradeoff can be achieved between how much resilience is provided and how much energy is consumed.

The second set of experiments measure the effect of both the size [Fig. 2.6] and energy [Fig. 2.7] requirements of a slice as compared to the program, has on our ability to detect soft errors. Fig. 2.6 shows that when we only consider slices that run at the same p-state as the program we can still detect a large

fraction of errors when using slices that are a small fraction of the program size. For example using slices that are 20% of the program size yield an error detection rate of about 40%.

A full duplication of the program would get a 100% detection rate, though we would need to double the energy used by the application. A benefit of program slicing for resilience is that initially the rate of soft error detection increases faster than the energy required to complete the computation of that slice. As we can see [Fig. 2.7], the percentage of errors detected for the amount of extra energy needed to run the slice is large especially on the lower end of energy scale. With about 20% energy use we can detect about 50% of soft errors that occur in the primary program.

It is important to note that the results do not imply that we can take multiple slices of any program and get full program coverage so that any soft error is detected with low energy use. That would depend on the program structure itself, and in some instances to get full program coverage, duplicating the Resilience by Exploiting Regular Structure in HPC Applicationsprogram would be better. This is because as stated in proposition 1 there must be duplication in any two unique slices of a program, and if most of the code is shared between the slices then the slices are doing so much work that it has to run at a lower p-state.

For purely sequential programs, getting slices with full coverage must ne-

cessitate at least using the same energy as the program if the slices need to be finished with their computation by the time the primary program is ready for the acceptance test. This will be equivalent to running a duplicate of the program. In the cases where most of the functions in the program are independent of each other, then it will be possible to run several slices, which together provide full coverage of all the program variables. It may be possible to stack some of these slices on the same core running at a lower p-state though it will not be possible to run all the slices on one core at the same p-state as the primary program without increasing the overall time needed to execute the program, due to proposition 1. On systems with lots of unused cores, the slices can be spread on several cores and can be run at different p-states depending on the when the acceptance test is expected to be run in the primary program. In this scenario only a fraction of the slices need to run at the same p-state as the primary program. It is clear that this technique can make many programs more resilient to soft errors while only requiring a small increase in the energy needs.

2.5 Related Work

Modular redundancy is an old and well known technique for error detection and masking. Engelmann, Ong, and Scott [25] have shown that in an HPC environment, Dual-modular redundancy (DMR) and Triple-modular redun-

dancy (TMR) increased the availability of the compute nodes. The purpose of DMR setup allowed the detection of errors where TMR allowed a voting mechanism to correct errors. However the duplication of work required for DMR and TMR is undesirable from an energy perspective.

An early software based method to detecting faults in applications was through the use N-version [11] approach to fault tolerance. The research into N-version was based on the idea that replication can occur in three areas: number of repetitions, hardware, number of times software is loaded. From a modern view, fault tolerance based on N-version is viewed as inefficient, and can be viewed as the worst case scenario when there is a need to obtain fault tolerance in an application.

Assertion based techniques have also been used to detect soft errors. These assertion techniques are dynamic in that they take the context of the program at the point of performing the assertion. Assertions have the advantage that they can detect faults during execution and have a high chance of detecting the faults that were designed to detect. Problem specific assertions [44] such as validating the result of an algorithm by checking invariants in the results is expensive in terms of the design, development, and testing cost. Assertion techniques have also been used as a mechanism to detect control flow problems that causes jumps between code blocks which have been caused by soft errors [33].

A modern programming variation of the ideas as put forward by N-version is to automatically duplicate components of the software [50, 48]. The duplication is done through a set of transforms such that the replacement code is functionally identical to the code being replaced. One type of duplication is the duplication of variables and the operations to change those variables. Another the duplication of on control flows so that any soft errors that affect control flows will be detected. The duplication requires a significant increase in the memory and energy requirement of running an application.

Redundant multithreading (RMT) is a redundancy technique that happens at the hardware level. In RMT programs are replicated across independent threads and then a comparison is performed on the outputs of the programs. Mukherjee, et al., [47] evaluate different RMT architectures on both single-processor and dual-processor simultaneous multithreaded single-chip devices. The architectures differ in the way the inputs are fed to the duplicated programs and the how the programs are synchronized. Our scheme provides a generalization of RMT through selective coverage of the set of program as done in this paper. Program slicing can also be viewed as a way to safeguard control flow by protecting the variables that affect control flow.

CHAPTER 3

HPC APPLICATION RESILIENCE USING SLICE SWARMS

3.1 Introduction

The reliable execution of a large scale HPC application requires that none of the large number of computing elements on which it runs fails or generates erroneous outputs during the execution. This is becoming increasingly difficult as the number of computing cores involved in the computation reaches into the millions and beyond. For example the current top performing system in the world is Sunway ThaihuLight [4], which is based at the National

Supercomputing Center in Jinan, China. It is composed 40,960 nodes, each with a 260 core chip, giving the system over 10 million cores. A study [32] was done on a Cray XT5 system called Jaguar, which consists of about 225k cores and 360 terabytes of main memory. This system was logging about 350 ECC errors per minute, and that a non-correctable double bit error was detected about once a day. This is already a problem and continues to get worse as we march towards exascale computing, where due to the number of components, current resilience methods will not scale. If the lack of scalability in the resilience methods in HPC applications is not properly addressed, strategies such as attempting to recover from a failure or error using the last known good checkpoint will not scale to exascale computing [39]. The reason for this is that as number of nodes increase, the number of checkpoints and recovery procedures that need to occur will also increase, which increases the length of time it takes the application to complete. This extra time to completion will thus be a contributing factor to the number of failures and errors. Furthermore, if the verification procedures, the procedures that ensure the application results fall within the allowed tolerance, occur at the end of the application or they are not sensitive enough to catch these errors, then the results need to be recomputed. This will require the application to restart execution from the beginning, rendering the checkpoints useless for catching these kind of errors as it is not known when these errors were introduced in the execution of the

application. Another important aspect of HPC programs is their large energy footprint. This footprint increases as the application is scaled and as the recovery overhead increases. Thus, the resilience methods must be sensitive to energy constraints.

The problem with many general purpose approaches to resilience is that the detection of errors happen either via periodic acceptance tests or when the errors manifest into issues or crashes which forces the resilience mechanism to immediately deal with the situation. In both of these cases a costly recovery process must be initiated which delays the completion of the application and uses extra energy. Current resilience methods that do have the ability to detect errors throughout the application use full redundancy methods which drastically increases the energy needs of the application.

In this paper, we introduce a resilience mechanism that will detect errors closer to the point where they occur and thus able to more efficiently deal the occurrence and propagation of errors. Our approach specifically deals with single program multiple data (SPMD) applications. Our contributions are: (i) show that system resilience for an application can be determined without individual core-level or node-level reliability data (ii) provide a model to reason about resilience and energy usage in HPC applications (iii) provide a mechanism to estimate the reliability of different parts of the application from continuous executions (iv) show how to optimize the distribution of slices to

reduce overall energy usage of the application.

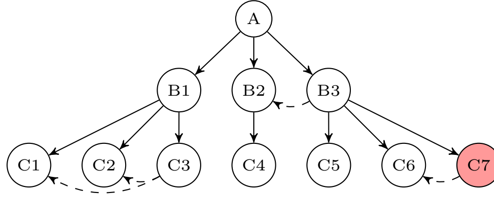

Figure 3.1: A program slice to be performed on the return value of the C7 function.

3.2 Background

3.2.1 Program Slicing Based Resilience

Program slicing [59, 54] can be used as a mechanism to detect errors in an energy efficient manner [6]. We use slicing as a mechanism in which given a variable, another program can be generated that only includes the instructions that directly or indirectly affects the value of that variable. In our model, programs are composed from functions that have various dependencies on one another. A slice on a variable that is computed within a function, would then be the program that contains only the functions that would be needed to compute the variable. The functions themselves would be modified such that the instructions that cannot influence the computation of the variable are removed. As an example, suppose we have a program composed of the functions in Fig. 3.1, where the solid lines are direct dependencies and the dashed lines

are indirect dependencies. An indirect dependency is a dependency where the result of one function is passed as a parameter to another function by a controlling function. Let us also suppose that we want to generate a slice based on the variable that is going to be returned from function $C7$. In this case we will have a program composed of several modified functions, as represented in Fig. 3.2, in which the final computation will be the variable that was sliced on.

We use slicing to generate a secondary program to detect errors that would otherwise either go undetected, pertub the results, or would perhaps cause issues later on in the processing pipeline. The way the error detection is accomplished is by comparing the result of a slice with the point in the program where the corresponding variable is computed. These are synchronization points between various points in the application and the slices. The slices themselves should have finished computing before the application reaches the corresponding variables. If these are identical, then we know that there was no error in either the direct or indirect dependency of the variable in question.

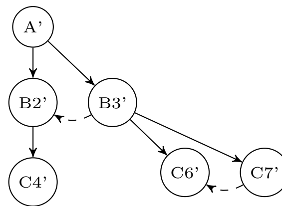

Figure 3.2: The corresponding program slice on the program in Fig. 3.1

In this work, we focus on improving the resilience of HPC applications by scaling the detection and handling of errors. We consider opportunities and constraints that are unique to HPC environments, such as the large number of processing cores and the need to reduce high energy costs of running applications in HPC environments. We are interested in the overall system resilience rather than the specific resilience properties particular to cores, nodes, or other hardware components. Considering the large number of cores and a specific application, some of the available cores in an HPC system can be provisioned to run slices as a way to detect errors and ensuring that the slices are constructed on variables that provide the highest resilience to the program. The principle idea behind the detection of errors is that each of the cores that are executing a process belonging to the application will have several associated cores that will be running program slices, each sliced on a different variable of the application. As soon as each slice finishes its execution and the program has computed the value of the relevant variable, a verification step is performed to ensure that no errors have been introduced from the beginning of the program up to the computation of the corresponding variable.

3.2.2 System level error estimation

When evaluating the occurrence of errors in simple applications, the number of these events can be assumed to arrive according to a Poisson distribu-

tion. The longer the application takes to execute, the more likely an error will be introduced in the computation, though for any fixed time periods, we can expect the same number of occurrences. The time to first arrival of an error, occurring during a run, will then follow an exponential distribution. When the same application is run on identical hardware environments, we expect each run of the application as a whole to follow a similar distribution. In this case we can use each run to provide better estimate on the parameters used to estimate the occurrence of these errors. When the underlying hardware or external environment causes changes to the rate of errors for different processes, then care must be taken in the estimation of the errors. This understanding of how likely an error is introduced during the computation of a variable, can be further extended when dealing with HPC environments. The extension that we consider in this paper is for HPC applications that are composed of many highly parallel processes running off of the same code base. Different parts of the program will use different operations, some that are more susceptible to error than others. The cause and variability across a program could stem from factors such as the way those operations compound issues, the length of time to complete the operation, or which electronic circuits they activate. The precise property these applications need will be more formally described in Sect. 3, though for application having these properties, we can view the processes as many independent experimental trials, and then for each variable

we have data on which we can estimate the parameters for rate of arrival of the errors.

The variability in the manufacturing of the electronic components for HPC systems will increase due to miniaturization, and this will cause more transient faults to occur at a higher rate [49] during the exercising of certain elements of a component. These anomalies may not necessarily occur at the same rate when exercising the same element on another component. Since these irregularities do not occur uniformly during the manufacturing processes, it would be hard to characterize their spread across components through the occurrence of rare events. If we assume that the spread of manufacturing irregularities on the components of a processing cores do indeed induce a Gaussian distribution on the mean of the error rate, we can simplify the modeling of the occurrence of these errors. Rather than view the combination of each component on a core and each usage as independent pairs, we can derive one parameter to model the expected arrival of errors in the system caused by a specific component the processing cores.

Proposition 3.1. *Let $\lambda_1, \ldots, \lambda_{\mathrm{N}}$ be the means of the arrival time of errors caused independently by a particular component on each of N cores. Let the value of the means be sampled from a Gaussian distribution, $\lambda_1, \ldots, \lambda_N \sim Norm(\lambda_*, \sigma)$. If each of these components are run for the same amount of time, then we can replace the parameters with one parameter $\frac{\lambda}{N}$.*

The parameter of the exponential distribution, which represents the rate at which errors arise, will not be Gaussian distributed when accounting for the system components as the source of the faults. This is because the number of errors tend to escalate when the issue in source faults cross some threshold. Since the parameter required for each variable in the system has been reduced to one parameter, a run can use all the cores as independent trials and the parameter for each variable can then be derived. This is assuming no prior understanding of the reliability of the components that the programs are run on. If it is known that some nodes or components have a higher likelihood of introducing errors for the computation of a specific variable, then that information can be taken into account so that additional resilience resources could be allocated to deal with the hardware or environmental factors.

For the purposes of detecting non-stopping errors throughout the execution of the application, we can collapse the errors that have any effect on the computation of variables into result of any function or indeed the computation of individual variables themselves. That is, the calculation of each variable can be viewed as smaller programs, where the input is analogous to receiving data from one part program and output to the variable being used in another part of the program. Any issue in these calculations, irrespective of the cause will freely cause erroneous results in these calculations. In this case, the parameter will give the likelihood that the calculation for a specific variable was the

source of an error, irrespective of the underlying cause (e.g. use of specific features of the core, failing memory modules, or cosmic rays). This is in contrast to computing the value of a variable using corrupted data from other variables resulting in erroneous results. When corrupted data is propagated, and detected further along the processing pipeline, there is no general purpose way to determine where along the pipeline the error occurred. Since the acceptance test happens after the slice finished executing, every variable that is included in the slice could be the source of the error and must be considered suspect. The errors can be evenly attributed among these variables, though the approach we adopt is that attribution is relative to the effort required to compute each variable compared to the total amount of effort required for the slice.

3.2.3 Energy consumption and slices

It may seem that using extra cores as part of a resilience method to detect errors, is as wasteful as duplicating the computations in terms of energy. Through the use of the dynamic voltage and frequency scaling (DVFS), we can run the cores at lower frequency on potentially much lower energy, depending upon how much voltage reduction is feasible. Since a slice will typically execute much fewer instructions than the main program, the overall energy consumption of a slice be substantially lower than that of the entire program.

Since the slices are run on another core, their execution does not lengthen the time to completion for the application. The program may also be written in such a way, where the primary process doesn't compute and use that variable till much later during the execution. The later a variable verification process is delayed the more energy can be saved by putting the cores running the slices into a lower voltage and frequency state. Other works have shown how DVFS and its effect on the occurrence of errors can be used to optimally make redundant components [52] and compilers [51]. In this paper, we will use known p-state value pairs as an approximation of what can be accomplished by lowering voltage and frequency which trades execution speed of the slice for low energy error detection.

3.3 Model and Analysis

In this paper, we use the same program and slicing model introduced in [6]. A notable feature of that model is that computations are not required to occur in a linear fashion when there are no dependencies between the different computations. It also captures the propagation of errors across the computational dependencies. These two features are important when dealing with resilience in HPC applications, as they capture the fact that the symptoms of transient faults cannot be observed at every point after they caused an error in a program.

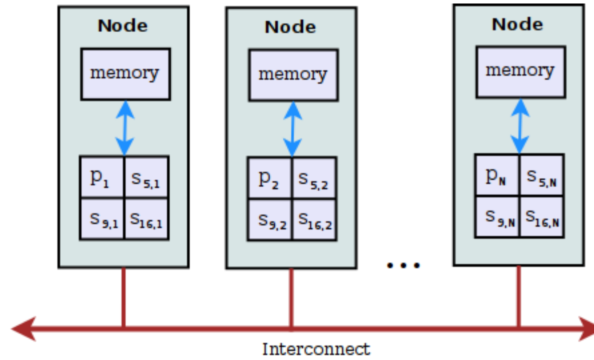

Figure 3.3: Architecture layout of slice placement

3.3.1 HPC processes and slices

An application written for HPC environment can be composed of many tasks that may perform heterogeneous computations, although the majority of applications developed for HPC environment are SPMD. We consider applications where the application is SPMD and processes distributed to a large number of cores. Each of these processes are run from one image and can be viewed as independent. Each HPC application consists of N processes denoted by the set $P = \{p_1, p_2, ..., p_N\}$ [Fig. 3.3] . The processes are not constrained in their ability to send and receive message during the execution of the application. Every process in a task executes the same image of a program S, which is a non-empty set of functions. The lower level properties required of these functions have been defined in the work on slicing-based resilience [6]. For any program S, we define the children of a function to be to all the functions it calls directly.

We define V to be the set of application variables $\{v_1, ..., v_n\}$, where n is the total number of variables. Since each process in P is running the same image, they will be computing the same variables, though the values of those variables are process and data dependent. V_p is the set of application variables for a specific process $p \in P$. We denote a slice s on variable $v_i \in V$, $i = \{1, 2, ..., n\}$, as s_i and when referring to a slice corresponding to a process p as $s_{i,p}$. The slices are constrained such that message sending is a no-op. Also, any generation of random values by the slice use the same seeds as those used to compute the corresponding variable in the main process. For each $v \in V$ there is an probability, $q(v)$, that the computation of variable v will introduce an error into the process and hence the program. We define $w(s)$ weight of the slice which is an estimation of the runtime of the slice.

Ideally, when using slices for resilience purposes we want slices to include variables that generally have a high probability of introducing errors compared to the weight of the slice. That is for some variable v_i, there is a preference for a slice s_i with a relatively high $\frac{Q(s_i)}{w(s_i)}$, where $Q(s_i)$ is the likelihood that there will be an error in the computation of s_i, that is $Q(s_i) = 1 - \prod_{v \in s_i}(1 - q(v))$. When the application starts, we don't have estimates for $q(v)$. The more times an HPC application runs, the better we can estimate $q(v)$, given by $e(v)$. We can reduce the absolute difference between estimates and their true value below any tolerance value ϵ, that is $\sum_{v \in V} abs(q(v) - e(v)) \leq \epsilon$, given

enough application runs.

As stated before, each application will have a corresponding slice for resilience purposes. For $i \in \{1, ..., n\}$ and for each slice corresponding to a process $p \in P$, the estimates $e(.)$ can be used to generate slices $s_{i,p}$'s such that, in the expectation, errors are discovered earlier in the execution of the application process and that process can be restarted. One advantage to early restarts is that energy is not wasted in both finishing the execution of the process and checkpointing. In the case where there is more than one slices on the same variable corresponding to a process, the results may be used for a slice that passed the acceptance test to do forward error correction.

To better understand the interplay between the occurrence rate of errors and the checkpointing in the context of transient errors, we need to get a better view of how early detection and detection delays affect the time wasted in the execution of resilience related functions rather than execution of the core application. Let T_{core} be the time needed to run one process of the core of the application. If no errors are detected during the execution and no errors were introduced in the results then the time needed for a no error executions, denoted by T_{NE}, is the time needed to execute the core application plus the time need to execute all the checkpointing. That is $T_{NE} = T_{core} + C * T_C$, where C is the number of checkpoints and T_C is the time that each checkpoint takes. For any given number of checkpoints, the time it takes to dump the data

for recovery, and time it takes to perform recovery, an optimal placement of checkpoints has been derived in [60, 23]. More recent work [43, 9] has looked at energy constraints and the uneven occurrence of errors in determining the best placement of checkpoints. We can use any method to determine the placement of checkpoints and work from within any framework where there is a known time, T_{opt} for the optimal placement of the checkpoint.

In the case where an error exists and is detected, the time needed to run the process $T_{Err} = T_{NE} + T_{lost}$, where T_{lost} is the lost time from a checkpoint till the error is detected and T_{Err} is time needed to run the process when errors are detected. $T_{lost} = w(s) * f(s)$ where s is the corresponding slice and f is a function that maps a program or slice to the speed ratio such that the it finished its computation by the time it is done in the corresponding process. Two possible ways to reduce this lost time due to the detection of an error, the first is to reduce $w(s)$ and the second is to reduce $f(s)$. When $w(s)$ is reduced, it is less likely an occurrence of an error will be detected and when $f(s)$ is reduced more energy will be used for the execution of the slice.

The goal then becomes to get the whole application to complete faster and do so with less energy than traditional resilience techniques. As we show in the rest of this section, we are able to do this by creating several slices for each process to detect the occurrence of errors that would otherwise not be detected till a later time in the execution of the application or not at all.

The application runs faster since it doesn't have to wait till the end of the work unit to restart when an error arises. The mechanisim also decreases the energy consumption since the work that needs to restart happens earlier. The goal then is to reduce the per process wasted energy, E_p, defined as the energy required to do the following: (a) run the application, (b) recompute computations with errors, (c) to perform the checkpoint, and (d) run all the corresponding slices - that is E_{NE}, E_{lost}, E_c, and E_s. E_s is a function of the size or time used to run the slices and is some fraction of E_r. E_{lost} can be reduced when an error is found sooner.

3.3.2 Characterizing the recomputation time under single checkpoint

To recover from an error, a recovery process needs to be initiated. As part of the recovery process an application can restart the computation from a checkpoint or start from the beginning of the application. The decision on where to start the recovery process should depend on where the error most likely originated rather than where it was detected. This is because the error could have propagated far from the original source before being detected. Thus a restart should avoid doing unnecessary work by starting from the beginning, but at the same time should not restart from a point where the error has already started propagating. We will consider two scenarios, the case when

only one slice per process is created and the case when multiple slices per process is created.

Single slice

Let's say we have exactly one slice of the application for every possible process, thus we have N slices. We say that there is regret when an error is produced in the computation of a variable and that variable is not a part of the corresponding slice, because resources used for that slice provided no benefit. There is also regret when the slice contains variables beyond the variable that was the source of the error, because the slice has used extra resources to detect that error. Another way to formulate that is to say that the regret is the effort needed to compute $w(s)$ when an error happens during the computation of a variable that is also computed in the slice versus the effort to compute $w(s_n)$ when the error happens in another part of the application. That is $r(s) = Q(s) * w(s) + (Q(s_n) - Q(s)) * (w(s_n))$. As an example, for an application that takes 25 hours to execute with exactly one error occurring, the total amount of time the application is expected to complete depends on which part of an application is sliced. In the case where each variable beyond the first depends on exactly one variable then the total expected time will follow the curve in Fig. 3.4.

The first time an application is run, we don't know much about the proba-

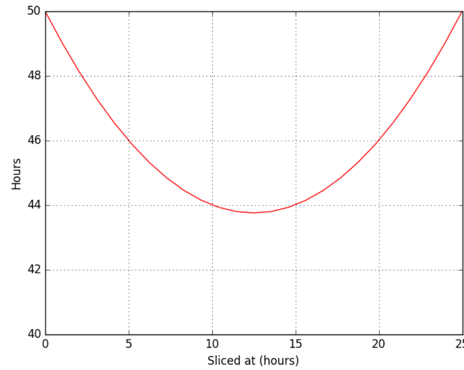

Figure 3.4: Total execution time for simple application with an error at slice point

bility of errors occurring due to the computation of individual variables. As we get better estimates of the resilience of different computations we can better estimate the best variable to slice on. More importantly our goal is not just to reduce the effects of errors being introduced in certain computations but to increase resilience of the system as a whole. By increasing the resilience we are also increasing the time needed for optimal checkpointing. So there are three major pieces that we need to solve. The estimate of the rate of error of each variable. The second is the distribution of slices so as to reduce the overall resilience of the system. Finally, the best distribution of slices that it is within the energy budget for the application and under the max running time.

For each variable $v \in V$, $q(v) = e^{-\lambda_v t}$ where λ_v is a parameter unique to the variable v. In an environment where the application can perform one checkpoint and has enough available cores so that each process can have one slice, we identify the waste to be the extra computation time beyond a purely

no error execution. For the No Error execution we have $Waste_{NE} = T_{core}$, since checkpointing is the only operation delaying the completion of the application. In the case where an error occurs, we have two cases.

1. The application is sliced on a variable v_i which is before the checkpoint. Here the waste becomes:

$$Waste_{Err} = Q(s_i) * (w(s_i) * f(s_i)) \tag{3.1}$$

2. The application is sliced on a variable after the checkpoint. In this case we have no mechanism of knowing if the cause of the error we are detecting occurred before or after the checkpoint took place. All that is known if an error is detected is that it has occurred in the processing path that is covered by the slice.

$$Waste_{Err} = Q(s_i) * (w(s_i) * f(s_i) + T_C) \tag{3.2}$$

This gives us a total waste of $Waste_{total} = Waste_{NE} + Waste_{Err}$. Minimizing the total waste is the same thing as finding the optimal time it takes to run application. This can be written as an optimization problem where we need to choose the variable to slice on given a program and the point at which the checkpoint happens.

It is important to note here that in the single slice case that the application cannot recover from errors by restarting from a checkpoint, unless we are

willing to accept probabilistic correctness in the results computed by the application. The reasoning for this, is that any errors detected after the checkpoint will have an origin sometime before the detection, which means it could have happened before or after the checkpoint. It is also important to note that case (1) is not always smaller than case (2) for a static checkpointing location. An example of this is when a variable right after the checkpoint has a relatively high $q(v)$.

Multiple slices

When we have at least k unused cores for every $p \in P$ application processes, where $k \leq n$, we can run k slices for every process of the application. This case is fundamentally different from the single slice scenario. The reason is that there are combination of slice configurations that can be chosen to make use of the checkpoint during recovery. For example when $k = 2$, two slices can be chosen, one right after the checkpoint, and the other some point later such that if the second slice detects an error the application recovers from the checkpoint. Let coverage be defined as the number of unique variables in a main process that is also being computed in a slice, then one factor that increases resilience is having a large coverage.

Architecturally there are advantages to running slices on the same node

[Fig. 3.3] as the corresponding main program, if there is enough cores and memory on a node to run the slices. One advantage is that acceptance tests can be done quicker, rather than negotiating the data from the execution of a slice to its corresponding process. Another advantage is that any mechanism to sync the messages to the main program and its slices can take advantage of the locality of the execution group to reduce latency and memory requirements.

In this case, determining the time the application wasted when discovering an error with a slice depends on whether the next smaller slice is before or after a checkpoint. If it is before, we will need to start from the beginning. If it is after, then we can restart from the checkpoint.

Let $\mathcal{L}$ be a list of slices that are sliced on different variables. $\mathcal{L}$ is ordered such that for any i, j where i, $j \in \{1, ..., k\}$ if $i < j$ then there exists a $v \in \mathcal{L}_j$ such that $v \notin \mathcal{L}_i$ and $w(\mathcal{L}_j) \geq w(\mathcal{L}_i)$. For each of the k slices we evaluate the waste based on whether the previous slice was before or after the checkpoint was taken. So the waste due only to the kth slice can be determined by knowing if the $k-1$ slice returns a variable that is calculated before the checkpointing takes place or after. In the case where the previous slice returns a variable that is computed before the checkpoint takes place then

$$Waste_k = Q(s_k) * (\prod_{j=1..k-1} (1 - Q(s_j))) * (w(s_k) + T_C). \tag{3.3}$$

In the case where $k-1$ slice takes place after the checkpoint then

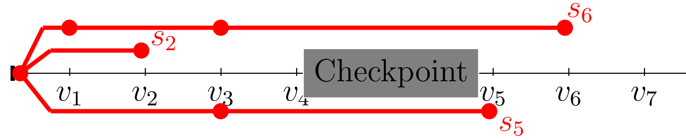

Figure 3.5: A process with three corresponding slices on variables v_2, v_5, and v_6. Solid circles represent the variables that are required by the slice.

$$Waste_k = Q(s_k) * (\prod_{j=1..k-1} (1 - Q(s_j))) * (w(s_k) - \frac{w(s_n)}{2}). \tag{3.4}$$

The example configuration [Fig. 5] shows the order which an application computes its variables along with three slices. If an error is detected on v_6, the application will restart from the checkpoint when no errors where detected at v_5. Otherwise, the application will need to restart from the beginning.

To find the optimal time for the application under this scenario, the selection of the k slices need to be distributed on variables such as to minimize the total time the application wasted, which is equivalent to finding the distribution of k slices so as to minimize the application runtime. That is we want to minimize the total waste, $\sum_{j=1..k} Waste_j$.

The upper bound for finding the best distribution of k slices so as to reduce the total wasted time running the application is $O(n^k)$. For a fixed $k \ll n$ the $\mathcal{L}_k$ slice can be sliced on $n - k$ possible variables, as k must come after every index before it. The same logic goes for the $\mathcal{L}_{k-1}$ slice and since we have k such slices we get the proper upper bound. As k approaches n, the number of configurations approaches 1.

3.3.3 Optimizing energy usage under multiple slices

The energy used by an application running on a socket is given by $\int^{w(s_n)} (\mathrm{V_{dd}}^2 * f(s_n))\, dt$, where $\mathrm{V_{dd}}$ is the voltage that corresponds to the minimum voltage required to run at the corresponding frequency. This energy use is only due to the dynamic power of executing the application as there is a constant underlying static power usage when the cores are powered.. To compute E_s such that it accounts for every slice in $\mathcal{L}$, we need to compute f each slice so as to reduce E_s. The total energy taken by all the slices is $E_s = \sum_{j=1...k} \int^{\frac{w(s_j)}{f(s_j)}} (\mathrm{V_{dd}}^2 * f(s_n))\, dt$. To account for the energy used in the recovery when an error is detected, we need to get the expectation of the waste due to all the of the slices. $E_r = \int^{\mathbb{E}(\sum_{l\in\{1..k\}} Waste_k)} (\mathrm{V_{dd}}^2 * f(s_n))\, dt$.

The energy used for checkpointing can be viewed as the time needed to perform the checkpoint at least once and an additional expectation of the fraction of the checkpoint run during the execution of an application. $E_r = \int^{T_c+T_c^{'}} (\mathrm{V_{dd}}^2 * f(s_n))\, dt$ where $T_c^{'}$ is the extra recomputation of the checkpoint due to the detection of errors. When the $\mathbb{E}(\sum_{l\in\{1..k\}} Waste_k)$ is less than the time when the checkpoint starts on the application then $T_c^{'} = 0$, otherwise $T_c^{'} = T_c$. The reason for this is that if in the expectation, the checkpoint is not included, then we can assume that on average it will not occur during the application run. Reducing the total energy for k slices is a matter of choosing the proper $f(.)$, where $f(.)$ gives the variables that the k slices should be sliced

on. In fact there is only possible $f(.)$ that gives the lowest energy. Reducing the energy usage becomes harder to analyze when the number of slices is variable. The reason is that reducing the number of slices will reduce the resilience due to errors but also decrease the energy requirement, and increasing the number of slices will have the opposite effect. The optimal point can be found using the elbow method [42], where the tradeoff is such that adding extra slices will reduce the waste by decreasing amount.

3.3.4 System level resilience and I/O handling

We have shown how slicing can be used to detect errors earlier and the performance impact this has. We also have shown the energy impact of detecting errors earlier in the processing of the application. For the purposes of this estimation we will assume that every core used to run the application will run along with k slices on k cores associated with that process of the application. Also, to make the analysis feasible we will assume each of the k slices for a process are sliced on the same variable. Each detection of an error gives us a set of variables that may have caused that errors. Some of these variables will overlap with other variables in other slices that have occurred later in the $\mathcal{L}$.

The input and output data consumed and generated by slices requires requires careful handling to make sure that the results computed by the slices and the mains process are the same. As with messages, the output to storage

generated by the slice processes can be converted into a no-op. The larger is issue is the way to handle the way the slice processes consume input data. There are two research issues that need to be resolved for a complicated I/O pattern of an HPC application. The first is how to order the consumption of input data such that the slice processes use the appropriate data segments for their computations. If the segment that needs to be read is dependent on how much data has been read before then needs to be a mechanism to mark the data that is to be used by each read. The second is that there may be scenarios where the data that is generated by the main causes the slice to spend the majority of its time in wait and then the result of the computation is needed soon afterwards. This second scenario is not an optimal use of the core as it is spending time idling rather than doing useful computation.

3.4 Method and Results

In this section we will detail the experimental setup and and discuss the results. We show the overall reliability across a general set of HPC applications and how the selection of variables converge to create a set of slices that provides better system resilience than randoms selection or the lack of any resilience methods.

3.4.1 Experimental parameters

The programs are generated in the same fashion as they were generated in [6]. The parameters that control the injection of errors, are generated according to Gaussian distribution based on the number of instructions needed for that computation. The way values for the Gaussian are generated is based on an assumption that an application on a 100k cores will, on average, encounter an error every 12 hours. Then depending on the size of the generated application, the error parameter for each variable computation is weighted according to their size. For the experiments in section 3.4.3 and 3.4.5 we assume that the $T_c = 10$ minutes and the checkpoint itself will occur about half way through an application run.

For all the simulations, the voltage and frequency pairs of the p-states is set according to Table 3.1. Each slice runs at the highest p-state possible such that it finishes its execution before the main process reaches the corresponding variable. This makes the slice consume the lowest possible energy such that the program is not delayed.

Table 3.1: Voltage/frequency pairs for the P-State

P0	P1	P2	P3	P4	P5
1.012V	0.958V	0.899V	0.845V	0.791V	0.737V
3.5GHz	3GHz	2.5GHz	2GHz	1.5GHz	1GHz

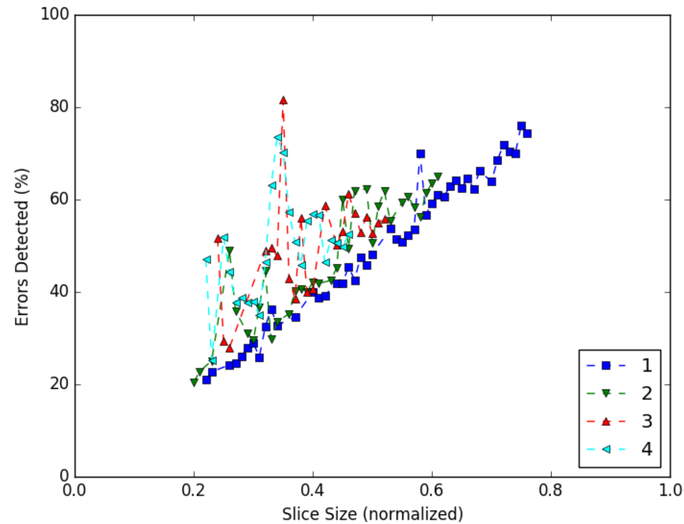

Figure 3.6: The percentage of errors detected when the energy usage of the slices is constrained to 60%

3.4.2 Detection of errors

The reliability of an application is compromised by undetected errors. In this experiment we look at how having multiple slices reduces the risk of having an error going undetected. We generate several slices for the program, ranging from 1 to 4 while constraining the energy usage of the slices to 60% of the primary application. We measure the average size of the slices as compared to the percentage of the program that is covered by at least one slice [Fig. 3.6].

Large overlaps between the slices are wasted computations in terms of providing resilience, though often it is necessary to be able to compute a slice. In the 1 slice case, the detection of errors has a near one-to-one relationship with the size of the slice. This may seem to put the one slice case on similar footing with standard redundancy as the detection rate is proportional to the

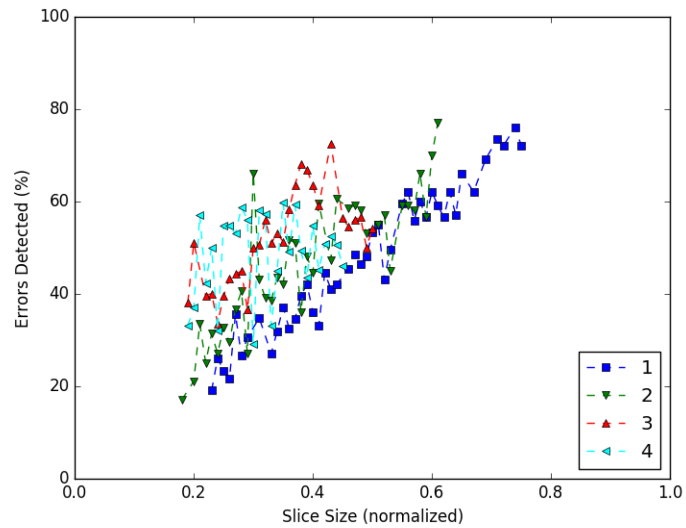

Figure 3.7: The percentage of errors detected when the energy usage of the slices is constrained to 60% and the dependence between functions is low

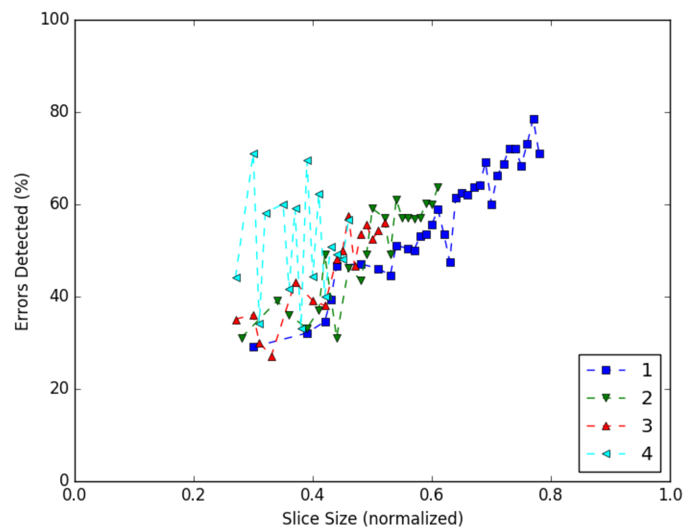

Figure 3.8: The percentage of errors detected when the energy usage of the slices is constrained to 60% and the dependence between functions is high

slice size. The difference in the one slice case is the advantage is reduced energy usage. In the case of the 3 and 4 slices case we see that there is a limit on the average size between 50% and 55% of the application size. This is due

to the generated applications being tightly coupled and thus the generated slices will have a lot of overlap. The evidence that tight coupling is causing this contention can be seen by noticing that for lower average slice sizes, the 3 and 4 slice sizes are detecting more errors than 2. As they increase in average size, the dependencies force overlaps until they cannot be constructed without exceeding the energy constraint, while the 2 slices case improves until about 60%. At about the 30% mark we can see a large rise in the errors detected for both 3 and 4 slice sizes. This is because at these sizes, they can still run at a more efficient energy level, yet are large enough so that they cover unique portions of the application. When the dependence among the functions of the generated program decreases, the opportunity to generate a slice under a given constraint increases. For example in [Fig. 3.7], where the dependency rate is set to 20%, we see the range of possibile errors from increase, as more opportunities arise where slices that can use less with a larger number of instructions is possible. On the other extreme, when it is set to 80%, the band becomes more narrow giving less opportunity for a slice to exist under the energy constraint.

3.4.3 Energy usage under different number of slices

Again, here we vary the number of slices between 1 and 4 and measure the total energy use as compared to the ability to detect errors. The energy

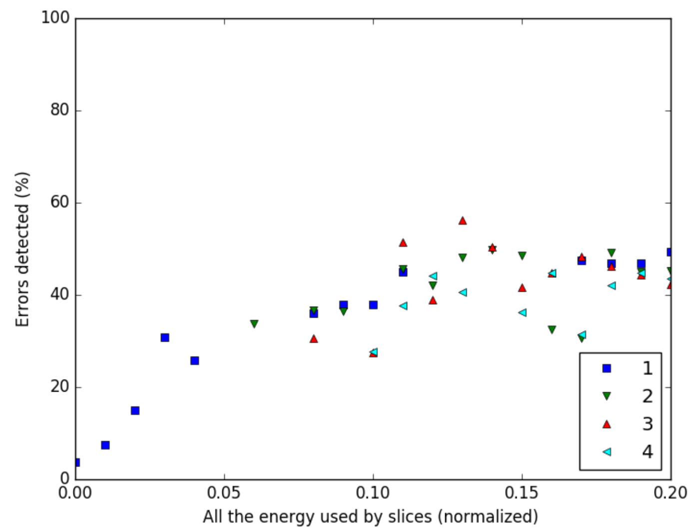

Figure 3.9: Error detection under 20% energy constraints

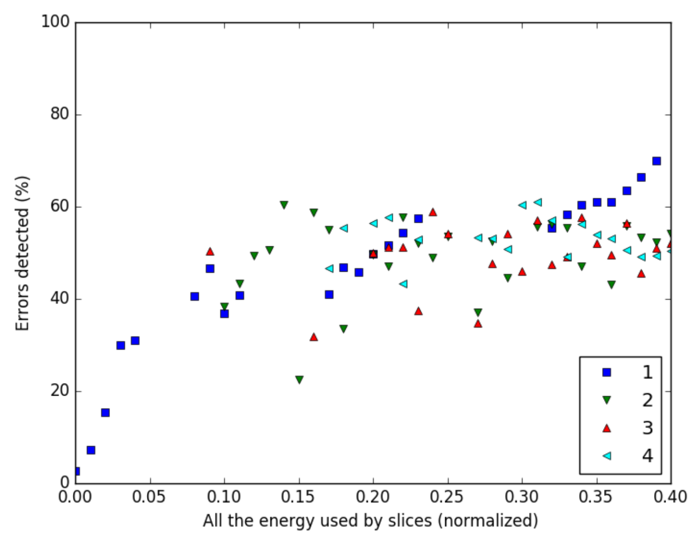

Figure 3.10: Error detection under 40% energy constraints

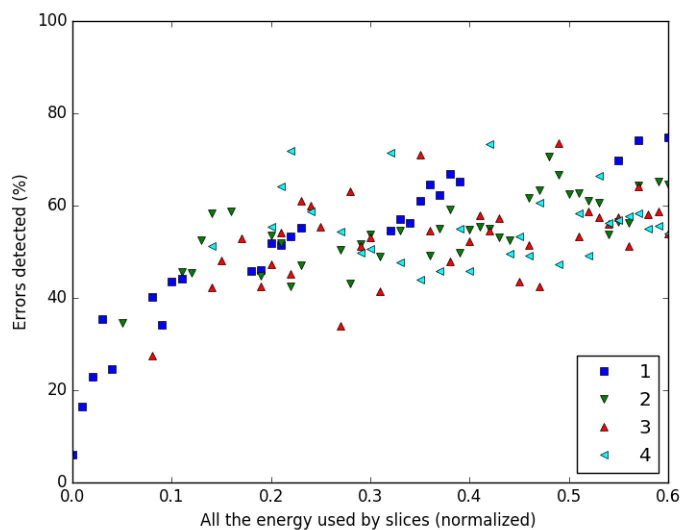

Figure 3.11: Error detection under 60% energy constraints

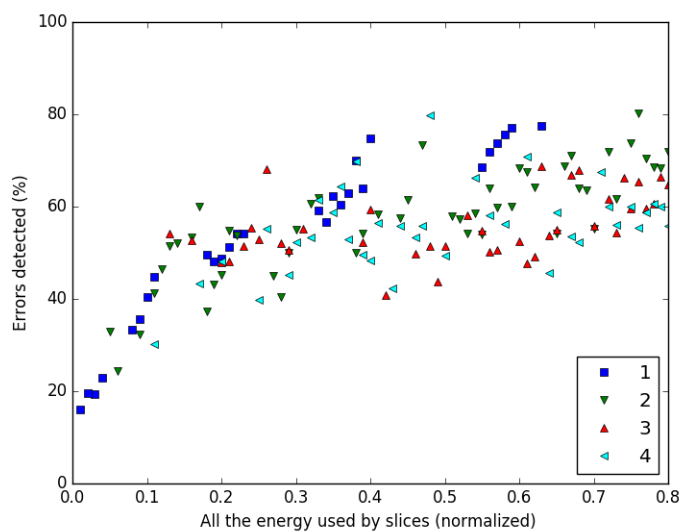

Figure 3.12: Error detection under 80% energy constraints

is constrained to 20%, 40%, 60%, and 80% of the total energy used by the main processes. There are several important things to note in the results [Fig. 3.9, 3.10, 3.11, 3.12]. The most apparent thing is the quick rise in detection of the 1 slice case. Its slope then rapidly changes. This is because as the size increases, the slice is not able to run at a low enough voltage and frequency thus increasing its energy. Also as the size increases, this forces other parts of the program to be included in the slice. For the results under all energy constraints, we see the varying number of slices allow possible resilience solutions under the respective constraints, when otherwise there wouldn't be a configuration that would be appropriate under those constraints. Fig. 3.11 corresponds to the energy graph for Fig. 3.6. Combining the energy and size view, we see that even when the dependencies in the application provide little room to make mostly non-overlapping slices, the higher number of slices provide opportunity to gain enough coverage of the application so that it detects a high number of errors for the amount of energy it uses. For example at 20% using 4 slices we get over 60% detection.

3.4.4 Reliability of coverage

The slices provide error coverage for parts of the program. As the number of cores increases, for both running the main and the slices, the probability of errors increase for both the main processes and slices. Fig. 8 shows that for 1k

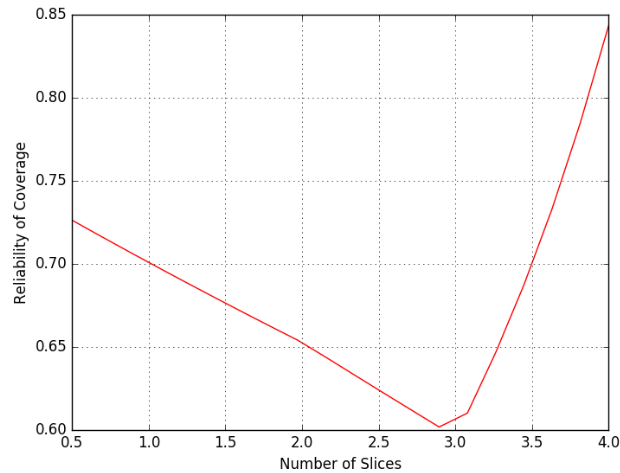

Figure 3.13: 10^3 core dedicated to application execution

cores dedicated to running the main process, the single set of slices provides reasonable reliability where another set isn't needed to be able to pinpoint the occurrence of an error. Fig. 9 shows that, there is a very large possibility that the slices will be a source of errors. For even larger number of cores another set can be considered so as to reduce the effect these errors will have on the completion time of the application.

3.4.5 Time to completion

In this set of experiments, we measure the expected time to completion of applications when slicing is introduced. Like the previous set of experiments, we vary the number of slices from 1 to 4. These are evaluated under three scenarios: 10^3 cores and 10^4 cores.

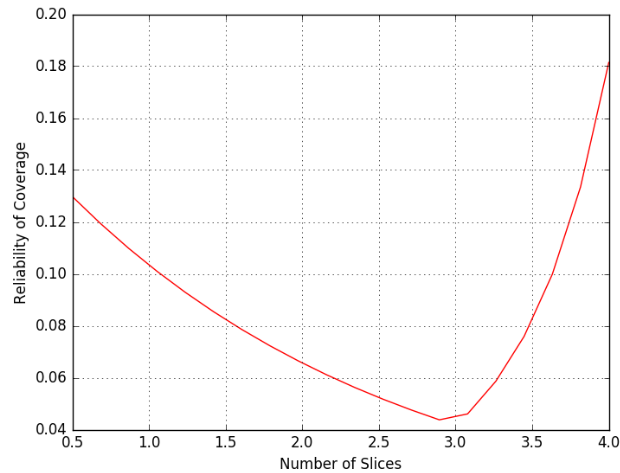

Figure 3.14: 10^4 core dedicated to application execution

The average time to completion [Fig. 3.15], is effected by the rate of the errors, but also by when the errors are detected compared to when they occur. It is also effected by the number of slices and the variables they compute compared to the placement of the checkpoint. For the 1k cores, we see that it adds a negligible amount of time to the total time to completion. For 10k cores, it adds about 10% to the total time to completion. For a larger number of cores, keeping the same constraints of up to 4 slices and the 60% energy threshold, the total time to completion becomes unacceptable. One way to address this is to increase the number of slices and checkpoints. This may require a tradeoff where an increase of the energy threshold and the number of cores used for resilience is necessary. The reason increasing the number of slices would decrease the time to completion is that errors are detected

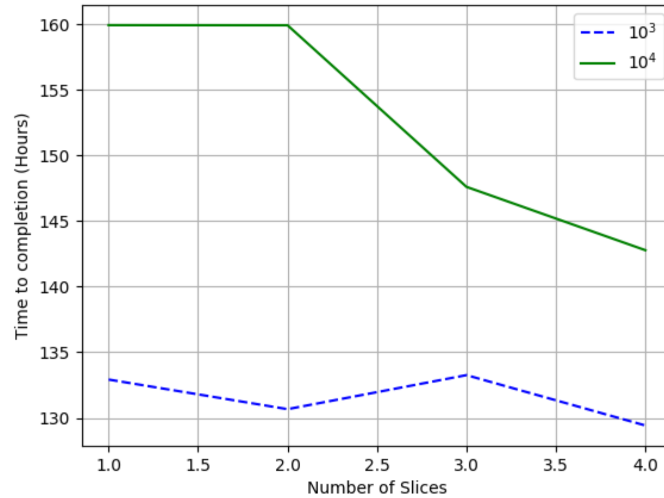

Figure 3.15: Time to completion

closer to the point where they occurred, thus allowing application to attempt recovery sooner. The tradeoff depends on how much dependencies there are among the different variables in the program.

3.5 Related Work

Replication of processes is a feasible [28] approach to resilience against failures in HPC applications when energy usage is not a constraining factor. One approach to process replication is rMPI , which is an extension to MPI [18], that transparently replicates processes and has the ability to recover from node failures and process crashes. Other studies [24] have shown that there is benefit to recovering from failures by adding redundancy to checkpointing

environment. The energy usage of the resilience mechanism is an important consideration when deploying large HPC applications. Three resilience techniques that were designed to recover from crashes have been evaluated [45] for their energy efficiency, and parallel recovery [19], an enhancement to message logging has been shown to be most energy efficient of the mechanisms that were evaluated. DVFS [56] has also been used to reduce the energy requirements of an application at the cost of increasing the time it takes the application to finish. Cui et al. propose a method they call shadow computing [22], that provides resilience against failures while being energy efficient. It does this by running companion processes, or shadows, that lags behind in execution to each main process by using DVFS to reduce the frequency. When a process failure is detected, the shadow can then be sped up and takes over as a main process.

RedMPI [29] is another MPI extension. It is able to detect data corruption errors. It does this through a specialized protocol that detects differences in the messages between paired processes. Using double redundancy errors could be detected and with triple redundancy errors can be corrected. The penalty that is paid for this resilience is the large amount of extra energy consumed.

There has also been recent work in combining resilience techniques to improve the overall reliability of an HPC application. One such technique [16] combines ABFT methods with checkpointing. By alternating between the two

on some applications, Bosilca et al. show that it is possible reduce the number of both periodic and incremental checkpointing while still being able to recover from crashes and detect errors. Benoit et al., develop optimal resilience patterns [15] for combing methods to recover from crashes and handle errors. Their model covers several six situations that differ in the number of checkpoints and acceptance tests, and under each scenario, provide an approximate optimal distribution that provides the best resilience. Regular structure found in classes of HPC application can also be exploited for resilience. In stencil computations, local error recovery [31] can be used to recover from process failures more efficiently. Recently, a method has been developed to recover from latent errors [27] and has been shown that, under certain constraints, to have less latency and use less resources for the recovery process.

CHAPTER 4

MIMIC: FAST RECOVERY FROM DATA CORRUPTION ERRORS IN STENCIL COMPUTATIONS

4.1 Introduction

In today's high performance computing (HPC) computing environments, the occurrence of faults must be considered during system design and the development of large-scale applications. These faults can lead to both failures of components and errors in the application. One class of errors is the cor-

ruption of application data during the execution of the application. A large scale field study [53] of servers in a datacenter environment, found that 8% of DIMMS are affected by errors every year and had higher than expected FIT rates (failures in time per billion hours) of 25,000 to 70,000 per Mbit. The number of components in the largest HPC systems and in the upcoming exascale systems all but ensure that corruptions error will be common. A study on one HPC system [32] found that it was logging 350 ECC errors per minute and non-correctable double bit error about once a day.

When available, error correcting code (ECC) hardware usually are able to correct single bit-flip errors and detect two bit-flip errors that occur in memory, although higher correctability and detectability is certainly possible at a higher cost. Errors that are not detected may start corrupting the working dataset of the application. The presence of silent data corruption (SDC) requires other approaches to ensuring that the results generated by the application are valid. In an effort to detect data corrupting errors, methods have been developed that include checking bounds of the results, performing redundant computations and ensuring the results match, and statistical analysis of the results. These mechanisms can usually only detect errors well after they occur. Data corrupting errors can propagate across process boundaries when messages are passed between processes. Thus by the time the errors are detected, the propagation could be widespread.

For users of HPC applications to have confidence in the generated results the user must be assured that any data corruption does not adversely effect the interpretation of the results. Therefore, where it is possible, effort must be made to detect the presence of corruption and recover to a good state to continue running the application. A latent error that is not detected early may also eventually lead to failures, which would force an application to restart from the last checkpoint. The earlier the latent error is detected and the recovery process is started, the less time is spent doing computations that will be thrown away. However doing constant detection is also wasteful as it slows down the rate at which the application makes progress and even if a data corruption event occurred, the detection mechanism may not be sensitive enough to detect it during its initial appearance. Once a latent error is confirmed a recovery process can be started.

A necessary precondition for building large scale fault tolerant systems is the need to fail as soon as possible when an error cannot be handled at the point of origin [8]. This makes dealing with SDC errors in HPC applications a challenging problem, since it is usually not known when these corruption occurred and how much corruption they caused. The ramification of this is that it is impossible to start the recovery process as soon as an SDC event occurs, since any detection of data corruption is not instantaneous. The corruption of data could occur and propagate without any signal given back to

the application. The problem of how to isolate and recover from these errors is paramount because traditional checkpoint-restart (CR) methods, are general purpose recovery methods and are expected to run into scaling problems as systems approach exascale territory [5, 39]. Current resilience methods that do have the ability to detect errors throughout the application use full redundancy methods which drastically increases the energy needs of the application. The largest component in the cost of running an HPC systems is the energy consumption [13, 38]. Care must be taken when adding resilience mechanism so that the cost of monitoring and executing the recovery of the resilience mechanism does not become prohibitively expensive.

In this paper, we devise an energy-aware resilience computational model for stencil computing we call mimic replication. Stencil computing refers to a class of applications with a specific structure consisting of many kernels with messages sent between these kernels at predetermined points in the processing cycle. This type of application structure is commonly used in many problem domains that HPC environments are employed to solve. Due to the regularity in the structure found in stencil applications, the propagation of data corruption could be well characterized. Due to the restricted message passing, data corruption in a stencil application propagates in a restricted manner, allowing the maximum propagation of the corruption to be predictable. If an acceptance test can be developed for a certain application to determine whether

an SDC occurred, we can determine the set of kernels that may have been the original cause of the SDC and the possible time steps that it may have been generated. For each kernel we perform the same operation at a lower frequency in another process. Dynamic voltage and frequency scaling (DVFS) [41, 58] can be use to control the frequency of mimics, which reduces their dynamic power requirements. Once an SDC is detected, we speed up a select number of companion processes, thus providing a quick recovery process. The overall mechanism attempts to detect errors closer to the point of occurrence thus reduces overhead of recovery.

4.2 Definitions and Model

4.2.1 Preliminaries

The observable behavior, that processes in a stencil application perform, is structured and predictable. The processes execute an iterative kernel that is central to the problem that the application is addressing. Once these processes reach a predefined point they send messages that is derived from their results to neighboring processes. For every process, the set of neighboring processes are known. In this paper we will only consider logical topologies where processes have the same number of neighbors and the geometry is symmetrical. During the message exchange phase, processes also receive messages that con-

tain results from the neighboring processes. Every so often during this process, a checkpoint is usually performed to ensure that failures or errors during the execution of the program does not result in a loss of the progress made up to that point and force a full restart. Traditionally, one issue that needs to be dealt with but that has not achieved a common consensus in the research community, is on a systematic approach to dealing with data corruption in the execution pipeline of these applications.

A central observation that mimic aims to exploit, so as to handle data corruption in stencil applications, is data corruption can only cross process boundaries by message exchanges between the processes. Since these message exchanges happen during specific points in the processing pipeline, the maximum number of processes that could have its results contaminated by some data corruption during any point in time, is known. During a message exchange, the maximum number of results that can be effected is that which is produced by the process itself and the results of its immediate neighbors. After any number of message exchanges, we know the maximum number of processes that may produce erroneous results given we know when the original data corruption has occurred.

4.2.2 Model

In many applications that are designed for execution in an HPC environment, we find that it is possible to run computationally expensive checks to determine if the results produced by the kernel contain errors or are plausibly error free. The problem is that since these checks are computationally expensive, they cannot be run before every message exchange. Doing so will length the execution time of the application, due to scale will allow other errors to be introduced, and will use extra resources in the execution of the application. If these checks are not run before every message exchange we will not know the exact moment that corruption started and the processor responsible for the data in which the corruption occurred. Like the approach Fang et. al. took [27], Mimic adds a step to the processing pipeline to perform these computationally expensive checks. The assumption is that these checks will indicate, with high probability, if there is a data corruption occurred and made its way into the results of a kernel computation that may cause the application to generate result deemed inaccurate or invalid. These types of checks can be in a form similar to that of detection methods found in algorithm-based fault tolerance (ABFT) methods [55, 20, 17]. ABFT are methods that depend on the on exploitation of an algorithms specific behavior and allowable effects that can occur under the algorithm. This is used to model the fundamental characteristics of the output that can be used to differentiate acceptable results

from those that must contain an error or corruption.

A stencil application consists of n stencil kernels, which are iterative kernels that update grid cells in a stationary pattern. Let $X^t_{i\in \mathrm{I}}$ be the value of the ith cell and timestep t, where $\mathrm{I} = \{1...n\}$ and $t \in \{0...m\}$. We define a *neighbor* function that takes the index i and return the set of indexes of the neighboring grid cells. We will only consider the situation where the message passing structure between processes are symmetric such that $\forall_{j \in neighbor(i)} i \in neighbor(j)$, which we will label property (i). Note that the 1-dimensional index i can be mapped to any k-dimensional grid topology. The other condition, property (ii), that we place on the topology is that $\{\cup\, neighbor(j) | j \in neighbor(i)\}$ is a strict superset of $neighbor(i)$ or is $\{1...n\}$. At every timestep a new value X_i is computed using its current value and the value of its neighbors. The result generated by the application will be some function of the values generated by the all the processes at the last timestep, $\mathcal{R} = f(X^m_1, ..., X^m_n)$. The values X^t_i for timestep $t > 0$ is itself a recursive function of its previous value and the set of the previous neighboring values; that is $X^t_i = f(X^{t-1}_i, \{X^{t-1}_j | j \in neighbor(i)\})$. This recursive structure describes the source of the corruption that originated in one cell and induces it to propagate and taint other cells in the application.

The application will consist of two types of processes that perform computations needed by the application. We will label one set P which we will

call the primary processes and the other M processes which we will call the mimic processes. Taken together $P \cup M = \mathcal{P}$ is the set of all processes in the application. A running process can only fall under one type at a time, that is $P \cap M = \emptyset$. The processes in each of P, M can be labelled such that the ith process is responsible for computing the value X_i. Each set will have its own independent copy of the computed value of X_i, the value of which will depend on the timestep that X_i is inspected. On the implementation level, this separation can be captured in the MPI framework [34] using separate communicators.

A program is defined as $Program \stackrel{\text{def}}{=} (P_1|...|P_n)$. The primary process has several responsibilities: it performs some computation which is mostly spent on the stencil kernel, the value of which depends on the input and the current state. It also receives messages from neighboring kernels, it sends messages to all neighboring processes containing the results of the computation. At some periodic interval it performs a relatively compute intensive check to ensure that the current results does not contain any data corruption. Here we introduce two functions, $TP : I \rightarrow \mathcal{P}$ and $TM : I \rightarrow \mathcal{P}$, which act as lookup table for message exchanges. Given the index, TP will give a reference to the primary process responsible for computing that i^{th} grid cell and TM will return a reference to the mimic process responsible for the ith grid cell under the mimic regime.

In this work, we analyze the general effects of the dimensions of the stencil structures and any specific n-point stencil topology will fall within the same framework. The structure determines the number of neighbors and the messaging patterns. This structure controls the specific propagation size and pattern of corrupting errors and is used pinpoint their possible source.

4.2.3 Recovery model

To develop the recovery model, we introduce a capability in the form of a function. Given a value, the function reports whether the value has been corrupted. The nature of this function is that it is computationally expensive to compute, though it always gives the correct answer. This function is called by a process, and it informs the process if it needs to trigger recovery . Once the recovery porocess starts, the application marks the point at which detection happened and begins the recovery procedure. This function could take the form of an ABFT that is specific to the application.

Lemma 1. *If a data corruption occurs during a computation performed by some process of a stencil application where both property (i) and (ii) holds and the number of cells, n, is fixed, and given that the applications runs for a sufficiently large but bounded time t, then all X's of the application will be tainted by the data corruption. Furthermore, given sufficiently large but bounded time t, the result of every tainted value will be used in the computation*

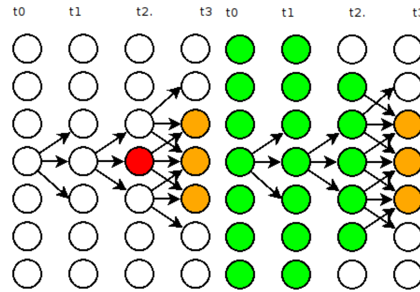

Figure 4.1: Seven processes that can only communicate with its neighbor. Knowing which process detects a corruption can be used to limit the possible time and sources of that error. The red in the left figure represents the origin of the corruption and the orange is the processes that can detect it. The green are those processes that can be originator of the corruption.

of every other value.

Several important observations used in the development of mimic model can be gleaned from lemma 1. Given an ideal detector, one that can detect when the some result has been tainted by the computation using corrupted input, then for an application that executes for a sufficient period of time, a single detector will eventually trigger an occurrence of such an event. Since not all tainted data results in the detectable corruption of that data, such an idealized detector may be only theoretical. A single detector may eventually detect data corruption, but from a recovery efficiency perspective, it would be better to detect the corruption closer to the point of generation. To that end, there are two choices when architecting the application for resilience. The detector can be run more often and the number of detectors can be increased. We will discuss these tradeoffs in the section on architectural considerations. For now, we will fix the number of detectors to n. Under such a scenario, the

choice that needs to be made is how often to run the detectors and how to proceed with recovery.

For modelling the propagation of data corruption, we make a simplifying assumption that any tainted data is also corrupted data, and that all corrupted data is detectable. During the execution of an application, there will be a periodic check for data corruption. This will happen after a process performs computation and obtains its local values though it happens before messages are passed to the neighboring processes.

The beginning of this cycle begins with the process selecting between two mutually exclusive paths. Depending on the timestep it either executes its internal operation and then performs messaging with its neighbors or it performs executes the stencil kernel, performs the error checks and then decides whether to continue with the message exchange or start the recovery process. For any primary process $P_i \in P$, the behavior of the process can specified as:

$$\begin{aligned} P_i \stackrel{\text{def}}{=} & [t\ mod\ check! = 0] \\ & \tau.(sendP_{neighbor(i)}(result)||input).P_i + \\ & [t\ mod\ check == 0] \\ & \tau.check.(sendP_{neighbor(i)}(result)||input).P_i + \\ & \tau.check.Trigger_i(d).0 \end{aligned} \tag{4.1}$$

In process description given in (4.1) we have two possible actions that

are responsible for coordinating the message passing between the current process and its neighboring processes, input and send. The input action can be thought of as a group of parallel actions waiting to receive the results from the neighbors. The abstraction of the send action is similar, with the caveat that its semantics require it to use TP and TM to know which process to send the message to. The $sendP$ action uses the TP function to know what primary process corresponds to the i_{th} cell. The check action is the verification done on the results to detect whether it contained any corruption. If a check detects corruption for the result of a given process during any timestep then a trigger is activated by that process and that process ends. Due to the structure of the stencil application, any time a trigger is activated, the set of processes that may have been the source of the corruption is known. The enumeration of the processes that can be the cause of the corruption is illustrated in Figure 4.1. The red circle illustrated the process and timestep that the corruption originated at. The orange are processes that detected the errors. Since a check is not performed at every timestep, the exact moment the corruption happened cannot be ascertained. The longer the timesteps between successive detection checks the larger the number of processors that must be added to the set of processes that could have led to the detected corruption. Note that, due to the symmetry of the application structure, we can eliminate some processes from being possible causes of the corruption. This can be seen in Figure 4.1

where the number of processes, that must be considered a potential source of the corruption, can be reduced.

$$\begin{aligned} Trigger_i(d) &\stackrel{\text{def}}{=} [d == 0]0+ \\ &[d > 0] \prod_{j \in neighbor(i) \cup i} TM[j \leftarrow 0].TP[j \leftarrow M_j].Trigger_j(d-1) \end{aligned} \tag{4.2}$$

Once the corruption is detected the behavior of the process, as described by equation 4.1, changes. It will activate a *trigger* and end [equation 4.2]. The trigger modifies TP and TM so that the index i of these structures refer to the corresponding process and removing it respectively.

$$\begin{aligned} M_i &\stackrel{\text{def}}{=} \tau.(sendM_{neighbor(i)}(result)||input).M_i+ \\ &triggered.M'_i \end{aligned} \tag{4.3}$$

Once a mimic enters the recovery process, it will run at a maximum frequency, communicating only with the neighbors that also entered the recovery state. As a group they will continue to move their segment of the application forward till they reach the point at which their corresponding main processes reached when the error was detected.

$$M'_i \stackrel{\text{def}}{=} \tau.(send_{neighbor(i)}(result)||input).M'_i \tag{4.4}$$

After a trigger is activated by some P_i, the process continues execution until it reaches a state $\models \Diamond M_i \sim \tau.(sendP_{neighbor(i)}(result)||input).P_i)$. Further-

more, this continues to propagate to the neighbors and from there propagate further till all possibly tainted processes are covered by a mimic process.

4.2.4 The cost of recovery

Resilience methods use extra resources to ensure that application can be recovered. In the traditional checkpoint-recovery method, this includes the cost of checkpointing including the storage and computational resources required for ensuring the application state is checkpointed, the cost of loading a good state from the appropriate checkpoint, and the computational cost it takes for the state of the application to reach the same point at which corruption was detected. The detection process will also use computing resources and will cause latency in the normal application flow.

Mimic also uses extra resources to perform its resilience function. During normal execution the mimics require their own computing resources. Since the frequency at which the mimics are required to run is less than the main processes, they can be stacked on a single core running at a higher frequency if there is no enough cores to run each mimic by itself. The tradeoff is that the recovery process will take longer once the recovery is triggered. This is because state of the application captured by the mimics will lag a little longer during normal execution and must share CPU resources during the recovery process. The resources used in the recovery process is a function

of the detection interval. Having a very long detection interval will allow any corruption errors that occurred to propagate further and thus will require more mimics to restore the application to its current state. On the other hand, with a very short interval, an unnecessary amount of resources is spent ensuring no errors are present. This will make the actual recovery fast as a minimal amount of extra computation between an occurrence of an error and detection.

The cost of the mimic system can be derived by looking at the four components that add to the cost. The notation used in the derivation of cost will be the same notation that is used in [27]. Let D and d be the detection interval and the time needed to perform the detection on one unit. Given a problem size of S in bytes. The first component of the cost function is the cost to move the application state for all the processes to the next detection point. The next component is the cost of performing detection process, which is cost required to run the detector on the application state. During normal execution, the mimics also factor into the cost. This component can be simply expressed as the ratio of the frequency the mimics run at compared to the maximum frequency multiplied by the cost of running the main processes. The last component is cost associated with the recovery process, which depends on the topology and frequency the mimics run at.

$$D * t * S + d * S + freq * (D * t * S) +$$

$$\sum (D * t * S * Recov_{mimic}(D))$$

For different system error rates, the optimal distances between detection points that minimizes the total cost of recovery changes. In a system with a high error rate we would not won't the primary processes and mimics to continue for a long time without a detection taking place. By the same token in a low error rate environment, a shorter detection interval will lead to the primary processes using resources to perform the detection process. This will allow the mimics to catch up in there computations, but getting too close to the primary will not necessarily be an advantage because the time it takes to execute the whole application may take longer.

4.3 Architectural considerations

4.3.1 Lag and Latency

There are a number of architectural tradeoffs that can be made depending on the sensitivity of the detector and the propensity of an data corruption to spread and propagate. If corrupted data tend to jitter and cause a compounding effect further down the processing pipeline, then reducing the sensitivity of the detector from our simplifying assumption of 100% corruption detectability

is possible. This is because under these conditions, given enough timesteps and detection points the error would eventually be caught. The lower the sensitivity towards corruption would require that the *mimics* lag further behind the *main* processes. This is because reducing the lower the detectability sensitivity the less the less likely any detected corruption occurs close to the point of detection. In situations where the detector will not always detect the occurrence of a corruption the developers of the application can require the *mimics* to lag further behind the progress of the *main* processes so that if an error occurs there can be a certain confidence that they will be corrected by the *mimic* recovery process. Given D timesteps between detection between the detector runs, the developer can set a lower bound on the distance of the lag that must be maintained between progress of the processes in P and those in M.

4.3.2 Number of detectors

Another choice that can be made is the number of detectors that need to run on any detection step. Having $|P|$ number of detectors with perfect detection will ensure that at every detection phase any corruption that occurred will be found. If $D > 1$ than we are not fully exploiting the structure of the application by having $|P|$ detectors. Since these are computationally expensive operations it would be beneficial to use less detectors if the it does not

effect the overall resilience of the application. Doing so will not speed up the completion of the application, but rather will reduce the energy requirements of the application.

We introduce the function $overlay(t, R)$ which will tell us how many processors will have coverage by more than one detector after t timesteps if the distance between the detectors, in terms of shortest corruption propagation, is R. The function can be defined as

$$overlay(t, R) = \begin{cases} 0, & t \leq \lfloor \frac{R}{2} \rfloor \\ \text{cause(i)}, & \lfloor \frac{R}{2} \rfloor < i < R \end{cases}$$

The *cause* function is dependent on the stencil topology and gives the number of processes that can be effected by a corruption event. The smaller R is, the more detectors that are run and the smaller D needs to be to make sure that there is no efficiency loss in redundant coverage.

4.3.3 Energy consumption of recovery

There are several ways to reduce the expected energy consumed by mimic. We can increase the lag of the *mimics* compared to the *main* processes. This is done by using DVFS to reduce the frequency of the mimics. Due to the dynamic power relation, which states that the dynamic power used by a circuit is proportional to the square of the voltage multiplied by the frequency. Another important relation is that voltage used is proportional to the frequency.

This gives us a non-linear relationship between the frequency and the dynamic power usage. A simple example is that if we reduce the frequency of program by $\frac{1}{4}$ we can expect to use $\frac{1}{64}$ the energy to complete the program which should take about four times as long to run.

We can also have a target where not all corruptions need to be caught. This may make sense in applications where corruptions make the error larger and larger during every computational timestep. Another is reducing the number of detectors or the number of mimics if we are willing to tolerate the inability to responding to all corruptions. We can also reduce the energy consumption by balancing the number of detectors against the number of mimics that need to run. Let E_{aux}, E_{det}, E_{max} be the energy consumption by a process that is idle, a process that is performing detection, and a process performing the stencil computation at maximum frequency. The least amount of energy is consumed when energy it takes to execute the detection phase is equal to the energy it takes for the mimics to perform the recovery. That is $(E_{aux} + E_{det}) * \frac{n}{R} = E_{max} * 2R(\sum_{t=0}^{\frac{R}{2}} cause(i))$. Solving for R will give us the optimal distance between the detectors.

4.4 Evaluation

In this section we evaluate mimic under several metrics that are useful for understanding its strength versus the traditional checkpoint-recovery scheme.

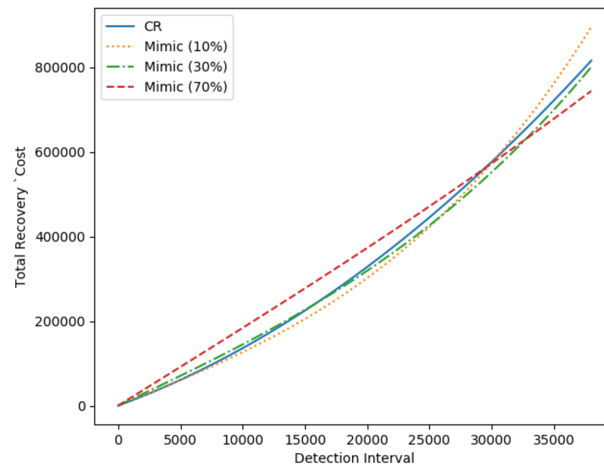

Figure 4.2: The total recovery cost given a system error rate of 0.01 for Mimics running at 10%, 30%, and 70% frequency

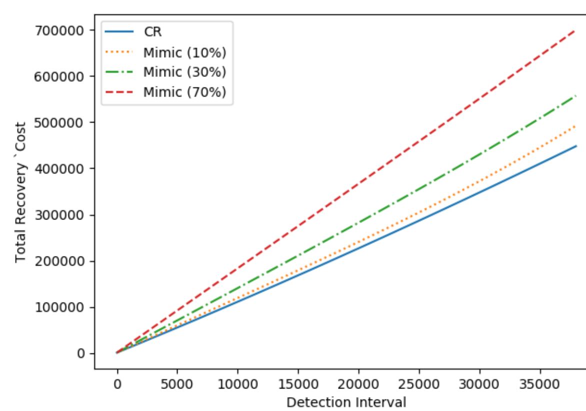

Figure 4.3: The total recovery cost given a system error rate of 0.001

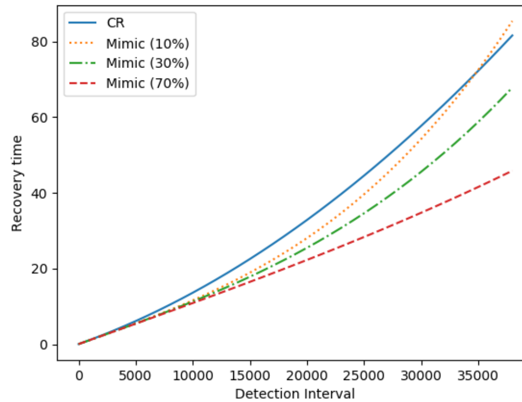

Figure 4.4: The recovery time as a function of the detection interval given 0.01 errors/second

We evaluate the resources used for providing resilience under a range of detection intervals compared to that of checkpoint-recovery. The variation in the detection interval is also explored under different level of system errors. The recovery cost then is evaluated under different number of detectors. The recovery latency is then compared to a checkpoint-recovery scheme. Finally, the extra amount of energy that a mimic based resilience method is expected to consume.

We use the same parameters as used in [27], so that the works can be compared, which we list here for convenience.

4.4.1 Recovery

Several factors affect the cost of recovery under the mimic framework. Two factors that can be controlled are the detection interval and the frequency of

Table 4.1: Configuration

Application size	10^9	bytes
Time to advance an element	10^{-8}	seconds/byte
Time to reload an element	10^{-9}	seconds/byte
Time to store an element	10^{-8}	seconds/byte
Time to run detector on element	100*t	seconds/byte
Detection interval	(variable)	timesteps
System error rate	0.001 to 0.01	errors/second

the mimics. The longer the detection interval the larger the computational cost. This behavior is self-evident since it would take longer to recover to the application state from a good state. In terms of traditional checkpoint-recovery this relationship is linear. The frequency controls the speed at which the mimics run and hence how far they lag behind the primary processes when a detection happens.

We first explore recovery when the system error rate is 0.01 errors/second. For mimics that run at 10% [Fig. 4.2] of the primary processes the slope of the mimic recovery cost starts lower than the checkpoint-recovery cost. As the detection interval increases, the lag of mimics become far enough where the cost of running at max frequency to catch up to the current application state becomes larger than running a checkpoint-recovery scheme.

Changing the frequency to 30% [Fig. 4.2] of the maximum frequency we get an interesting effect where in the window of being evaluated, after the initial crossover in cost between CR and mimic, the total cost stays under CR. This implies total cost is actually less than CR. This may seem counter intuitive,

but that the resources required at any one time is lower, rather that the total used resources will be lower. For example, since Mimic requires processes, more computing resources may be required at any one moment but the total CPU time used is less than the CR scheme.

Under an environment with a reduced error rate (0.001), the total recovery cost stays above the recovery cost of CR. This is understandable by considering that a large number of mimics are running with little recovery usage. One way to alleviate this wasted cost is remedy to this usage is to reduce the number of detectors. This will not reduce the latency, because the processes need to move in lockstep, but it will reduce the total cost.

4.4.2 Latency

In HPC applications, the time to completion for an application is a primary concern when considering any resilience method. So in the previous sections we have shown the resource efficiency of Mimic over CR, but that does not necessarily translate into a quicker application completion. Here we show, that under the detection interval window considered, Mimic recovers faster after a detection than CR. In Figure 4.4, mimic is shown under three different frequency configurations. All of them have some detection intervals in which they outperform CR in the time to recovery. Running at the 10% of the frequency of the main processes, the recovery time is slightly below the recovery

time of CR until it crosses over around the 32k timestep, at which point it takes longer to recover. The reason there is an eventual crossover where the mimics take longer to recover is that after certain point, it is faster to recover from a nearby checkpoint, which gets further ahead as the detection interval gets larger, due to the separation between the main processes which are running at maximum frequency and the Mimics.

4.4.3 Energy Consumption

To explore energy consumption used by the mimic framework, we will first consider a low error rate environment. In addition to being able to recover the application faster, Mimic has a competitive energy profile [see Fig. 4.5]. Below 15k timesteps, running the Mimics at 70% results in a faster recovery time, but it will use more energy than CR. The Mimics running at 30% of main processes use 80% of the energy used by CR at 20K timesteps. In the low detection interval regime, Mimics running at lower frequency use less energy overall. The reason is that for the shorter intervals there is a smaller chance an error is detected and that energy used for the Mimic goes to waste. As the detection interval increases the chance an error occurs during that interval also increases. Since running at lower frequency has a non-linear relationship to power consumption, the energy used till an error is detected is much lower than running at maximum frequency to that point.

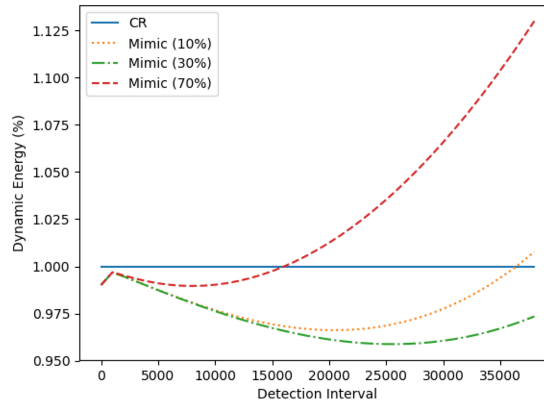

Figure 4.5: Energy usage of the mimics normalized by energy required by CR for varying detection interval for a system error rate of 0.001 errors/second

It is in the higher error rate regime that mimics energy requirements very attractive [4.6]. An interesting observation is that both 30% and 70% requires less energy than the 10%. This is due to two factors that occur simultaneously. The first is the nonlinear effect of dynamic power lower, where much less power is required to run at a somewhat slower frequency. The second is that in the presence of an error corruption, they require less time to make a full recovery an hence require less time running at maximum frequency. Under this high system error rate, setting up mimics to run at 70% will reduce the latency and will use much less energy than CR for a large range of the interval under consideration.

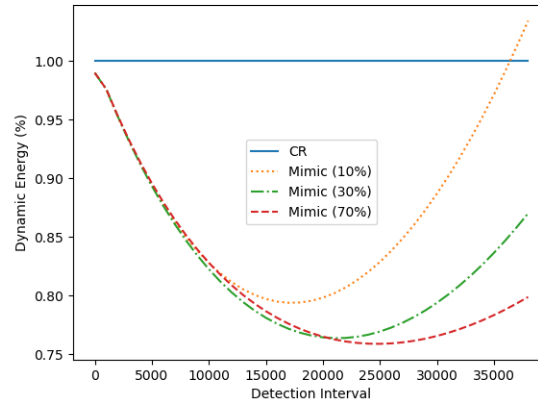

Figure 4.6: For a higher error rate of 0.01 errors/second the system using mimics require less energy

4.5 Related Work

4.5.1 Detecting data corruption

Work into understanding software errors and its relation to the rise of SDC has been well studied [21, 36, 14]. Of particular importance to this work is efforts to understand the frequency of these events and methods to trace corrupted data as it propagates across the application [27]. A recent work that tackles the problem of data corruption directly [27] in stencil application also uses the structure of the application to provide efficient recovery. It does so by using Global View Resilience (GVR) to create versioned checkpoints which allows the checkpointing procedure to capture the application state over time in a low cost manner. Research work has also been done on integrating detection mechanism with traditional recovery methods in order to improve the overall

reliability of an HPC application. One such work [16] uses checkpointing for its traditional role and combine ABFT. It demonstrated that combining these approaches reduce the number of both periodic and incremental checkpointing while still being able to recover from crashes and detect errors. Benoit et al., develop optimal resilience patterns [15] for combing methods to recover from crashes and handle errors.

4.5.2 Energy aware recovery

In the HPC context, DVFS [56] has been used to reduce the energy requirements of an application at the cost of increasing the time it takes the application to finish. A resource efficient form of process replication has been proposed in the shadow computing system [22]. It does so by using DVFS to run the replicated processes at a lower frequency and voltage. When a process failure is detected, a shadow process is sped up until it reaches the point in the computation where the shadow can replace the main process. To address the detection of SDCs in applications and assist in the recovery, program slicing was introduced as method to provide partial resiliency [6, 7].

4.5.3 Process failure and replication

A naive approach to adding resilience to HPC applications is replicating the application processes. This is a straightforward yet resource intensive ap-

proach to enhancing application resilience against failures in HPC applications [28] when energy usage is not a constraining factor. One approach to process replication is rMPI , which is an extension to MPI [18], that transparently replicates processes and has the ability to recover from node failures and process crashes. Other studies [24] have shown that there is benefit to recovering from failures by adding redundancy to checkpointing environment. The energy usage of the resilience mechanism is an important consideration when deploying large HPC applications. Three resilience techniques that were designed to recover from crashes have been evaluated [45] for their energy efficiency, and parallel recovery [19], an enhancement to message logging has been shown to be most energy efficient of the mechanisms that were evaluated.

Comparing duplicated messages is another approach to detecting corruption. One such technique was used in the development of RedMPI [29]. By comparing the messages from replicated processes it is able detect data corruption errors. It does this through a specialized protocol that detects differences in the messages between paired processes. Using double redundancy errors could be detected and with triple redundancy errors can be corrected. The penalty that is paid for this resilience is the large amount of extra energy consumed.

4.6 Conclusion

Data corruption is a problem that effects applications running on the largest HPC systems currently in use and will be a problem that cannot be ignored as the next generation of HPC systems come online. These systems, which will be approaching or surpassing the exascale mark, will have so many components that the occurrence of data corruption during the execution of an application is all but assured and will require efficient resilience methods to ensure that the results of the application can be relied on. We introduced a computational resilience model to handle data corruption in stencil computing applications. Furthermore, we validated the advantages of the approach in that compared to recovery from checkpoint. It has faster time to completion and with respect to process replication it is more efficient.

CHAPTER 5

CONCLUSION

We have introduced a new approach to providing resilience for software applications based on program slicing. Our contributions include a framework under which program slicing can be reasoned about in relation to providing resilience. We also experimentally show that program slicing can provide resilience with attractive energy requirements.

The results demonstrate that otherwise idle cores can be used to provide resilience in an energy efficient manner that will be more compelling as the number CPU cores in modern systems start to increase drastically. Although the techniques proposed in this paper are applicable generally, they are particularly valuable for HPC applications involving long running computations on large number of servers. A lot of the research effort in providing resilience to HPC is being focused on reducing both the power consumption and resources

required. This work shows a method towards these goals while maintaining the same application completion time.

There are three important research challenges that build on this paper. Research needs to be done on how to enhance the slicing scheme to handle side effects in functions. This occurs when a function uses or manipulates state outside its scope. This type of interaction occurs during IO and interprocess communications. Another common example where side effects is introduced into a function is when a function needs a source of randomness to perform its computation. Slicing would need to be done in a way to guarantee that the variables being sliced on in both the program and in the slices would have the same values when no soft errors occur. This would mean that there needs to be some coordination in how the side effect influences the results of computations and adjust behavior accordingly stored in variables of the program and the slices of the program.

Secondly, there may be some advantages to selecting more than one variable to slice on. It is an open question whether slicing on multiple variables provides any advantages in terms of the fraction of soft errors that are detected or the amount of energy saved. In addition to the energy savings and ability to detect soft errors, research on this question will also need to determine the tractability of determining the optimal or near optimal number and location of variables to slice on.

Another promising research direction is how to address the occurrence of failures in long running programs. This problem can be tackled from two different angles. The first assumes the program slice is run at the same or higher frequency than the program and when the slice runs into a failure and addresses what can be done to prevent the program from failing knowing that the slice has failed. The other way is to run several slices at a lower frequency, each capturing different program variables and when a failure occurs in the program addresses how the slices can be set to run at a higher frequency and utilized so that the primary program doesn't have be rerun.

Resilience techniques for HPC applications need to scale and become more energy efficient to allow applications to fully exploit the HPC environments of today and tomorrow. We have shown that slicing can be used as an error detection mechanism and that it can be scaled to large number of cores. We have shown that when compared to full redundancy it is an efficient technique to detect errors for the resilience an HPC receives. This is because the slices can be made to compute a large part of the program while running at a low frequency, which reduces the amount of energy needed to run a slice.

Due to the scale at which future HPC application will work on, future research in this area should address ways to bring resilience against errors and failures into a unified energy efficient implementation framework. It will also address implementation issues such as message passing and randomness.

We will also revisit the simplifying assumption made in this work of having each slice run on its own core and evaluate ways to share cores among the slices. Finally, effort will be put into finding ways to perform quick verification between other variables in the slice and the application using intermediate application dependent sanity checks. This will make use of the computation work the slice is already performing to catch errors earlier to the point of occurrence.

Finally, by utilizing the structure common to many HPC applications, we developed a method to efficiently recover from the detection of SDCs. We do this by recognizing the propagation of SDCs to the message flows between kernels. Using this observation, we generate companion processes to each of the primary processes. We use these companion processes to speed the recovery process when an error is detected during the acceptance test. These companion processes run at a lower frequency than the primary processes which reduces their energy consumption. When an error is detected a predetermined number of these companion processes run at the maximum frequency.

Future research in ways to enhance a Mimic type approach to handle the propagation of data corruption can take several paths, three of which we discuss. First is to enhance ABFT for more problems and develop a theoretical model of what type of applications are amendable to ABFT. Second, better characterize the propagation of corruption in HPC for different types of ap-

plications. Third, though many stencil applications are symmetric, a larger number of applications can benefit from this framework if work is done on extending the framework to handle non-symmetric stencil applications and explore the boundaries on the messaging patterns that must be known for a Mimic type approach to be useful.

REFERENCES

[1] Intel and core i7 (nehalem) dynamic power management. https://impact.asu.edu/cse591sp11/Nahelempm.pdf. Accessed: 2016-04-15.

[2] Measuring the power/energy of modern hardware. http://www.prism.gatech.edu/\char‘\~gtg417r/micro47/downloads/Section4.pdf. Accessed: 2016-04-15.

[3] Simpy. https://simpy.readthedocs.io/en/latest/. Accessed: 2016-04-15.

[4] Sunway taihulight: National supercomputing center in wuxi. https://www.top500.org/resources/top-systems/sunway-taihulight-national-supercomputing-center-i/. Accessed: 2016-01-25.

[5] Erika Abraham, Costas Bekas, Ivona Brandic, Samir Genaim, Einar Broch Johnsen, Ivan Kondov, Sabri Pllana, and Achim Streit.

Preparing hpc applications for exascale: Challenges and recommendations. In *Network-Based Information Systems (NBiS), 2015 18th International Conference on*, pages 401–406. IEEE, 2015.

[6] Anis Alazzawe and Krishna Kant. Power-aware program resilience through slicing. In *2016 18th IEEE International Conference on High Performance Computing and Communications 2016, 14th IEEE International Conference on Smart City, 2016 2nd IEEE International Conference on Data Science and Systems*, 2017.

[7] Anis Alazzawe and Krishna Kant. Slice swarms for hpc application resilience. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 1–10, 2017.

[8] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, Mikroelektronik och informationsteknik, 2003.

[9] Guillaume Aupy, Anne Benoit, Thomas Hérault, Yves Robert, and Jack Dongarra. *Optimal Checkpointing Period: Time vs. Energy*, pages 203–214. Springer International Publishing, Cham, 2014.

[10] Jean-Luc Autran and Daniela Munteanu. *Soft Errors: From Particles to Circuits*, volume 39. CRC Press, 2015.

[11] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, 11(12):1491, 1985.

[12] S. Balasubramanian, N. Pimparkar, and et.al. M. Kushare. Near-threshold circuit variability in 14nm finfets for ultra-low power applications. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*, pages 258–262, March 2016.

[13] Javier Balladini, Remo Suppi, Dolores Rexachs, and Emilio Luque. Impact of parallel programming models and cpus clock frequency on energy consumption of hpc systems. In *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*, pages 16–21. IEEE, 2011.

[14] Leonardo Bautista Gomez and Franck Cappello. Detecting silent data corruption through data dynamic monitoring for scientific applications. In *ACM SIGPLAN Notices*, volume 49, pages 381–382. ACM, 2014.

[15] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Optimal resilience patterns to cope with fail-stop and silent errors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 202–211, May 2016.

[16] George Bosilca, Aurelien Bouteiller, Thomas Herault, Yves Robert, and Jack Dongarra. Composing resilience techniques: Abft, periodic and in-

cremental checkpointing. *International Journal of Networking and Computing*, 5(1):2–25, 2015.

[17] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[18] Ron Brightwell, Kurt Ferreira, and Rolf Riesen. *Transparent Redundant Computing with MPI*, pages 208–218. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[19] Sayantan Chakravorty and Laxmikant V Kalé. A fault tolerance protocol with fast fault recovery. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[20] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[21] Cristian Constantinescu, Ishwar Parulkar, Rick Harper, and Sarah Michalak. Silent data corruption—myth or reality? In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 108–109. IEEE, 2008.

[22] X. Cui, T. Znati, and R. Melhem. Adaptive and power-aware resilience

for extreme-scale computing. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 671–679, July 2016.

[23] John Daly. *A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps*, pages 3–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[24] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 615–626. IEEE, 2012.

[25] Christian Engelmann, Hong Ong, and Stephen L Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the IASTED International Conference*, volume 641, page 046, 2009.

[26] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture,*

2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on, pages 7–18, Dec 2003.

[27] A. Fang, A. Cavelan, Y. Robert, and A. A. Chien. Resilience for stencil computations with latent errors. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 581–590, Aug 2017.

[28] K. Ferreira, J. Stearley, J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2011.

[29] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[30] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester. Bubble razor: An architecture-independent approach to timing-error detection and correction. In *2012 IEEE International Solid-State Circuits Conference*, pages 488–490, Feb 2012.

[31] Marc Gamell, Keita Teranishi, Michael A Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. Exploring failure recovery for stencil-based applications at extreme scales. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 279–282. ACM, 2015.

[32] A. Geist. Supercomputing's monster in the closet. *IEEE Spectrum*, 53(3):30–35, March 2016.

[33] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 581–588. IEEE, 2003.

[34] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced features of the message passing interface.* MIT press, 1999.

[35] D. Hand, M. T. Moreira, H. H. Huang, D. Chen, F. Butzke, Z. Li, M. Gibiluka, M. Breuer, N. L. V. Calazans, and P. A. Beerel. Blade – a timing violation resilient asynchronous template. In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pages 21–28, May 2015.

[36] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/I-*

FIP International Conference on Dependable Systems and Networks (DSN 2012), pages 1–12. IEEE, 2012.

[37] Scott Hemmert. Green hpc: From nice to necessity. *Computing in Science & Engineering*, 12(6):8–10, 2010.

[38] Scott Hemmert. Green hpc: From nice to necessity. *Computing in Science & Engineering*, 12(6):8–10, 2010.

[39] William M. Jones, John T. Daly, and Nathan DeBardeleben. Application monitoring and checkpointing in hpc: Looking towards exascale systems. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, pages 262–267, New York, NY, USA, 2012. ACM.

[40] S. Kim and M. Seok. Variation-tolerant, ultra-low-voltage microprocessor with a low-overhead, within-a-cycle in-situ timing-error detection and correction technique. *IEEE Journal of Solid-State Circuits*, 50(6):1478–1490, June 2015.

[41] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 123–134. IEEE, 2008.

[42] Trupti M Kodinariya and Prashant R Makwana. Review on determining

number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.

[43] Yudan Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. L. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–9, April 2008.

[44] Aamer Mahmood, Dorothy M Andrews, and EJ McCluskey. Executable assertions and flight software. 1984.

[45] Esteban Meneses, Osman Sarood, and Laxmikant V Kalé. Assessing energy efficiency of fault tolerance protocols for hpc systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 35–42. IEEE, 2012.

[46] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: an architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247, Feb 2005.

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

[48] B Nicolescu and Raoul Velazco. Detecting soft errors by a purely software approach: method, tools and experimental results. In *Embedded Software for SoC*, pages 39–51. Springer, 2003.

[49] Konstantin Nikolic, Akram Sadek, and Michael Forshaw. Fault-tolerant techniques for nanocomputers. *Nanotechnology*, 13(3):357, 2002.

[50] Maurizio Rebaudengo, M Sonza Reorda, Marco Torchiano, and Massimo Violante. Soft-error detection through software fault-tolerance techniques. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium on*, pages 210–218. IEEE, 1999.

[51] S. Rehman, A. Toma, F. Kriebel, M. Shafique, J. J. Chen, and J. Henkel. Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 273–282, April 2013.

[52] M. Salehi, M. K. Tavana, S. Rehman, F. Kriebel, M. Shafique, A. Ejlali, and J. Henkel. Drvs: Power-efficient reliability management through dynamic redundancy and voltage scaling under variations. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 225–230, July 2015.

[53] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram

errors in the wild: A large-scale field study. *SIGMETRICS Perform. Eval. Rev.*, 37(1):193–204, June 2009.

[54] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.

[55] VK Stefanidis and KG Margaritis. Algorithm based fault tolerance: Review and experimental study. In *International Conference of Numerical Analysis and Applied Mathematics*, 2004.

[56] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 786–796, May 2015.

[57] Xingsheng Wang, Binjie Cheng, Andrew Robert Brown, Campbell Millar, Jente B Kuang, Sani Nassif, and Asen Asenov. Interplay between process-induced and statistical variability in 14-nm cmos technology double-gate soi finfets. *IEEE Transactions on Electron Devices*, 60(8):2485–2492, 2013.

[58] Chia-Ming Wu, Ruay-Shiung Chang, and Hsin-Yu Chan. A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters. *Future Generation Computer Systems*, 37:141–147, 2014.

[59] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005.

[60] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.