HARDWARE ASSISTED MEMORY CHECKPOINTING AND APPLICATIONS IN DEBUGGING AND RELIABILITY

A Thesis
Presented to
The Academic Faculty

by

Ioannis Doudalis

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2011

HARDWARE ASSISTED MEMORY CHECKPOINTING AND APPLICATIONS IN DEBUGGING AND RELIABILITY

Dedicated to Dimitrios, Elisavet, Stylianos and Thaleia-Dimitra

ACKNOWLEDGEMENTS

My first thanks are due to my advisor Prof. Milos Prvulovic. He gave me the opportunity to pursue my PhD studies in the US, and guided me through this long and not always smooth journey. Always open for discussion, always willing to explain everything, no matter how easy or hard it was, Prof. Prvulovic taught me how to think as a computer architect and a researcher.

I would also like to thank my PhD committee: Prof. Hyesoon Kim, Prof Hsien-Hsin S. Lee, Dr Gabriel Loh and Prof Alessandro Orso. All four have served as a motivation for me, assisting me at different times of my PhD, advising me both on academic and as professional issues.

Fellow students also played an important role in my life as a PhD student. I would like to thank Guru, who helped me from the first time I joined the PhD program and gave me invaluable advice about PhD life and research, Samantika, Kiran, Chenyu, Dong, Nak Hee for the long and lively discussions we had, Jungju and Anshuman for patiently helping me with my papers and presentation slides, the attendees at our architecture seminar for their useful comments about my research. I would also like to thank Alan and Susie for their prompt administrative support, and Chad and Peter for their IT support.

Finally I would like to thank my family, my father Dimitris, my mother Elisavet, my brother Stylianos and my sister Thaleia-Dimitra for their support at every step of my PhD career, from the point I decided to go to the US until now. They have always been by my side and taught me patience and optimism, I will always be grateful to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Thesis statement: Memory and performance overheads of application and system checkpoints can be significantly reduced with the assistance of hardware techniques.

The problems of software debugging and system reliability/availability are among the most challenging problems the computing industry is facing today, with direct impact on the development and operating costs of computing systems. A promising debugging technique that assists programmers identify and fix the causes of software bugs a lot more efficiently is *bidirectional debugging*, which enables the user to execute the program in "reverse", and a typical method used to recover a system after a fault is backwards error recovery, which restores the system to the last error-free state. Both reverse execution and backwards error recovery are enabled by creating memory checkpoints, which are used to restore the program/system to a prior point in time and re-execute until the point of interest. The checkpointing frequency is the primary factor that affects both the latency of reverse execution and the recovery time of the system; more frequent checkpoints reduce the necessary re-execution time.

Frequent creation of checkpoints poses performance challenges, because of the increased number of memory reads and writes necessary for copying the modified system/program memory, and also because of software interventions, additional synchronization and I/O, etc., needed for creating a checkpoint. In this thesis I examine a number of different hardware accelerators, whose role is to create frequent memory checkpoints in the background, at minimal performance overheads. For the purpose of reverse execution, I propose the HARE and Euripus hardware checkpoint accelerators. HARE and Euripus create different types of checkpoints, and employ different methods for keeping track of the modified memory. As a result, HARE and Euripus have different hardware costs and provide different functionality which directly affects the latency of reverse execution. For improving the availability of the system, I propose the Kyma hardware accelerator. Kyma enables simultaneous creation of checkpoints at different frequencies, which allows the system to recover from multiple types of errors and tolerate variable error-detection latencies. The Kyma and Euripus hardware engines have similar architectures, but the functionality of the Kyma engine is optimized for further reducing the performance overheads and improving the reliability of the system. The functionality of the Kyma and Euripus engines can be combined into a unified accelerator that can serve the needs of both bidirectional debugging and system recovery.

CHAPTER I

INTRODUCTION

1.1 The Problems of Debugging and Reliability.

The problems of software debugging and system reliability are among the most challenging problems the computing industry is facing today, with direct impact on the development and operating costs of computing systems. These two problems, even though they seem to be disjoint, share a lot characteristics and sub-problems. Those common traits across areas are the target of current and future research that will result in efficient and retargetable solutions, which will in turn reduce the overall design, manufacturing, programing and operating cost.

Software debugging is the process of identifying and correcting the causes of program errors. It has been estimated [39] that debugging accounts for 60%–70% of the development effort and 80% of project overruns. It has also been estimated that in the early stages of development bugs cost an order of magnitude (50 to 200 times) [10, 11] less to fix than in later stages, and that 20% of bugs cause approximately 80% of program code rework. Software errors are expected to cost to the U.S economy approximately $59.5 billion [78]. Beyond the financial cost, software bugs can result in the loss of human lives [41]. An important problem for software debugging is how to accelerate the process of identifying the causes of incorrect program execution, thus reducing the overall software development and testing time, as well as the associated costs.

System reliability targets the uninterrupted correct operation of computing systems, as well as quick detection and recovery from errors. The types of errors that might affect a system span across the whole hardware/software stack. For example, as processor circuits scale to smaller transistor sizes they become increasingly vulnerable to radiation [13] (e.g. alpha particles) that can cause transient errors, which in turn can result in erroneous computation. Software bugs can also disrupt the correct operation of a system. For example, the power outage that in 2001 affected 50 million people across eight US states and Canada [67,100], was partly caused by a software bug that resulted in the alarm system's failure.

1.2 Techniques for Improving Debugging and System Reliability.

In debugging, in order to isolate the cause of an error, we often have to re-execute the program multiple times using breakpoints and watchpoints, a process that is extremely time-consuming for long-running applications. To accelerate the debugging process, bidirectional debugging [12] has been proposed. Bidirectional debugging creates for the programmer the illusion of executing the program seamlessly in either direction (forward or in reverse). This functionality avoids multiple re-executions and thus allows the programmer to more rapidly isolate the cause of the program error. The illusion of *reverse execution* is created with a combination of application state checkpointing, which can restore the program to a past point in time, and deterministic replay. A checkpoint is a copy of the whole system state (e.g processor registers and program memory) at a given point in time. The key parameter that affects the interactivity of reverse execution, and effectively its usability, is the frequency of checkpointing. More frequent checkpoints result in the application spending less time in deterministic-replay in order to restore the system to the desired point in time.

For reliability, once an error has been detected, the system has to recover and resume execution. A simplistic recovery approach would be to restart the executing program from the beginning, a solution that would have high cost, especially for long-running applications (e.g. weeks-long protein folding, drug interaction, nuclear explosion, etc. simulations). Restarting the program is not enough to guarantee completion of the execution, especially for cases where the error frequency is high. To improve the availability of the system, a common recovery approach is backwards error recovery, which relies on checkpointing. When an error gets detected, the program is restored to the most recent undamaged checkpoint and resumes execution from there, instead of re-executing the program from the beginning. Like reverse execution, in error recovery frequent checkpointing reduces the amount of time the application will have to waste recomputing the lost state, and thus improves the availability and the efficiency of the system.

1.3 Overview

Even though frequent checkpointing could deliver the desired reverse-execution interactivity and system availability results, it is generally associated with high performance and storage overheads. Such prohibitive overheads render bidirectional debugging a rarely used and not widely applied debugging technique. Moreover, because of the expected increasing error rates [83] and the limitations of the current I/O infrastructure, the increased checkpointing requirements of future systems are expected to limit their scalability [61].

This dissertation proposes a number of different hardware accelerators for application and system memory checkpointing, with applications in reverse execution and system reliability. The system/application memory typically represents the biggest fraction of the state which needs to be checkpointed. All hardware accelerators which I propose aim to reduce both the performance and memory overheads, and construct memory checkpoints very often. The rest of this dissertation is organized as follows:

Chapter 2 provides an overview of the existing system and application checkpointing techniques.

Chapter 3 describes HARE and Euripus, two hardware assisted checkpointing mechanisms which aim to accelerate and reduce the performance and memory costs of bidirectional debugging. HARE and Euripus employ different mechanisms for keeping track of the modified memory and create different types of checkpoints. The primary goal of both techniques is to reduce the memory footprint of bidirectional debugging by enabling the consolidation of checkpoints. For this reason HARE creates redo-log checkpoints, which can be created to be ordered by address and thus efficiently consolidated, contrary to previous checkpointing techniques [68, 94] which use unordered undo-logs. In contrast of HARE, Euripus creates undo-log checkpoints, but uses a trie as its meta-data data-structure for the checkpoints, instead of the commonly used list-of-addresses. This trie can be efficiently traversed in address-order which enables Euripus to consolidate the undo-log checkpoints. Euripus's meta-data data-structure also allows it to store and search for other information efficiently, enabling the mechanism to construct both types of checkpoints (undo and redo-log) simultaneously for limited additional cost. To accomplish this, Euripus exploits a number of synergies that develop when both types of checkpoints are being created. Moreover, Euripus's functionality is extended to track additional application information which can further accelerate bidirectional debugging. The different design decisions of the two techniques affect the hardware cost, performance/memory overheads, reverse-execution latency, as well as the additional functionality that each technique can support.

Chapter 4 discusses the Kyma hardware checkpoint accelerator, which approaches the problem of checkpointing from the perspective of reliability. Kyma is structurally similar to the Euripus accelerator, but has been adapted to the characteristics of checkpointing used in system reliability. In reliability, the goal is to improve the availability of the system by reducing its down-time. The down-time of a system consists of the time to restore the system state to the last checkpoint and the time to re-execute to the point of failure, and frequent checkpointing is required to minimize the re-execution time. At the same time, multiple and different types of checkpoints have to be maintained in order to enable the recovery from different types of errors (transient or permanent), as well as allow recovery from errors that escape early detection and corrupt the state of a number of checkpoints. For these reasons, Kyma creates both undo-log checkpoints at high frequencies (e.g. 10msec), for quick recovery from transient errors, and redo-log checkpoints less frequently (e.g. every 1sec), for recovery from permanent errors or errors that undo-log checkpoints cannot be used to recover from. When both types of checkpoints are being created, the potential overhead practically doubles and two memory tracking mechanisms are necessary. To reduce the performance cost, similarly to Euripus, Kyma takes advantage of the synergies of undo and redo-log checkpoints. This allows Kyma to eliminate the modification-tracking mechanism for creating redo-log checkpoints, as well as to reduce the overall memory bandwidth requirements and performance overhead. To represent the checkpoint meta-data, Kyma uses a data-structure similar to the one that Euripus maintains during checkpoint construction. Kyma also exploits checkpoint consolidation in order to create checkpoints at lower frequencies (e.g. 1 minute and 1 hour) at no additional memory copying cost. Those multiple levels of checkpoints allow the system to recover a lot more quickly by exploiting the frequent checkpoints and being able to fall-back to an older checkpoint when the system cannot recover the error using the latest checkpoint. Kyma also leverages fast non-volatile memory (e.g. PCM) to store frequently created checkpoints, thus overcoming the bandwidth limitations of other permanent storage solutions.

Finally, Chapter 5 presents the conclusions of this work.

1.4 Scope of this Dissertation

The main goal of my work is to demonstrate to the reader that specialized hardware accelerators, designed for constructing memory checkpoints, can efficiently reduce the performance overheads of software techniques and retain the benefits of software solutions, e.g. ability to consolidate checkpoints. The proposed solutions are evaluated through simulations that demonstrate the performance benefits, and analytical models have been used to show the impact of the proposed techniques in the reverse-execution latency that the user will experience during bidirectional debugging (HARE, Euripus) and the availability of the system (Kyma). The evaluation results demonstrate that through careful design the proposed accelerators provide the benefits of both hardware (low performance overhead) and software (low memory requirements) techniques.

CHAPTER II

OVERVIEW OF MEMORY CHECKPOINTING TECHNIQUES

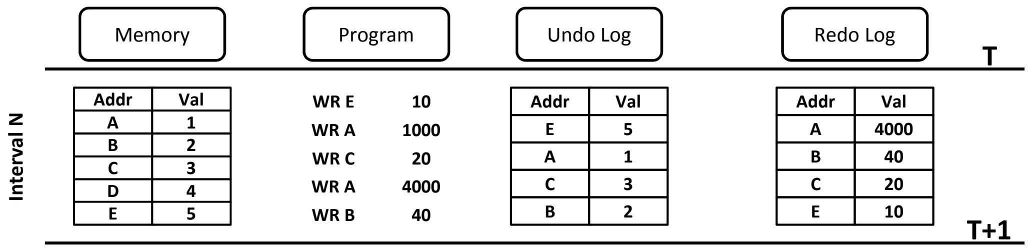
2.1 Introduction to Checkpointing and Logging

This chapter presents an overview of checkpointing and logging techniques and how they are typically implemented. Checkpointing and logging have been studied across multiple fields. Their purpose is to record the modifications of the system over-time, with the intent to restore the system to a past point in time if needed.

A checkpoint is a consistent snapshot of the full state of the system (or an application) at a given point in time t. To create such a snapshot, the system pauses execution (to prevent any further modifications), copies all its state to a secondary location, and then resumes execution. To restore the system to a specific point in time (t), the snapshot taken at time t is copied back to the system's state.

Logging has a similar purpose to checkpointing, and is often used synonymously. However we can make a distinction: logging records the modifications of the state of a system over-time, e.g. as a set of records of the form <state_id,value>. The state_id can be, for example, an address, if the system is tracking modifications to memory, while the value can be either the old or the new value of the modified memory location. Based on the value recorded, old or new, the log can be used to "move" the system's state either to a past or to a future point in time.

Given these descriptions of checkpointing and logging, checkpointing appears to have higher performance overheads than logging, especially when performed frequently. Checkpointing typically needs the execution of the system to be paused for the checkpoint to be constructed, while logging records the modifications as they occur, with a less drastic impact on the system's/application's execution. To improve the performance of checkpointing, several techniques have been proposed that construct checkpoints incrementally, by checkpointing only the state of the system that has been modified during a checkpointing interval, and not the full state. Such an incremental checkpoint is similar to a log that has recorded the latest values of the modified state of the system. As a result,



Addr	Val
A	1
B	2
C	3
D	4
E	5

WR E	10
WR A	1000
WR C	20
WR A	4000
WR B	40

Addr	Val
E	5
A	1
C	3
B	2

Addr	Val
A	4000
B	40
C	20
E	10

Figure 1: Example of undo and redo-log checkpoints

the terms checkpointing and logging have few differences, and in the rest of this thesis I am going to use these two terms interchangeably to denote a mechanism that records the set of changes needed to "move" the system's state to a specific point in time.

Checkpoints/logs can have three typical forms: First, there is the full checkpoint, which corresponds to the original definition of checkpointing: it saves the full state of the system as it was at the time when the checkpoint was created. Such a checkpoint is expensive, so it is typically created in order to establish a common starting point for incremental checkpoints. Second, a redo-log checkpoint (Figure 1) is an incremental checkpoint/log that records the latest values of the modified state of the system at the end of the checkpointing interval at time *T+1*. A redo-log checkpoint cannot by itself restore the system to time *T+1*, because it does not contain the full state of the system. Instead, a redo-log alone can only move the state of the system forward in time, from *T* to *T+1*. To restore the system to time *T+1* from another point in time (not *T*) using redo-log checkpoints, the process has to start with a full checkpoint, and then apply all subsequent redo-log checkpoints in chronological order until time *T+1*. Finally, an undo-log checkpoint (Figure 1) records the oldest values of the modified state of the system during the checkpointing interval. Recording the oldest value during an interval allows the undo-log checkpoints to roll-back the system state from the end of the current interval (time *T+1*) to the beginning (time *T*). Undo log checkpoints can be used to restore the system to a more distant past state, e.g. to time *T-3*, by applying them in reverse chronological order. Like redo-log checkpoints, undo-log checkpoints also do not contain the full state of the system, so they require a valid state of the system as a starting point, but this starting point should be more recent than the "target" point. For this purpose, undo-logs can use the *current* state of the system, if it is consistent and undamaged. This gives them a significant advantage if the "target" state is in the recent past relative to the current state of the system., e.g. when performing a rollback in case of an error or for reverse execution.

There are cases though, that the point in time t we want to restore the system/application to does not correspond to an existing checkpoint. In such a case the system is restored to the closest checkpoint in the past from time t, and the system re-executes until it reaches time t. The ability to restore the application to practically any point in time with the help of checkpoints has multiple applications. This dissertation focuses on two such applications of checkpointing: error recovery (reliability) and bidirectional debugging. In error recovery, after an error is identified, checkpoints are used to recover the system to a past error-free state. In bidirectional debugging, checkpoints are used to restore the system to a past or a future state, allowing the user to move to any point in time in the application's execution, and inspect how the program's values change over time.

2.2 *Checkpoint Implementations*

A checkpointing mechanism can be implemented with the assistance of the application, library, operating system, virtual machine, a dedicated hardware mechanism, or by some combination of these (e.g. hardware-assisted OS-based checkpointing). it should be noted that checkpoints at different levels of the HW/SW stack capture a different part of the system state and, if used alone, can be used only for recovering that specific subset of the system state.

Application Level: Checkpoints can be created at the application level through explicit application code [30, 52], with the assistance of compiler instrumentation [6, 15, 16], or with the help of a checkpointing library [17, 63–65]. In application-driven checkpointing [30, 52], the application periodically pauses computation and stores (e.g. to a file) any intermediate results it has generated. In case of a crash, the application can resume execution using the last saved data. Compiler-assisted checkpointing [6,15,16] relies on compiler instrumentation for creating checkpoints at intervals that minimize the overall checkpointed memory state [6], and enforces the consistency of checkpoints in multi-threaded applications [15,16] with the help of the underlying run-time system. Library checkpointers [17, 63–65] attach to the application's code and keep track of the application's modified memory. A typical implementation [63] uses memory protection to identify the modified memory and create incremental undo-log checkpoints using the "copy-on-write" functionality provided to the library by the underlying OS.

Operating System Level: Creating application checkpoints at the operating system level [95] relies on existing operating system mechanisms for keeping track the application's memory modifications. For example, to support bidirectional debugging, Flashback [95] establishes checkpoints of the executing application by creating "shadow clones" of the application using OS functionality similar to the "fork" system call. The "fork" system call creates a copy of the original application, and uses memory protection to copy, in a lazy fashion, the modified memory locations. Every "fork" creates a new checkpoint of the application that the debugger can use to recover the application to.

Virtual Machine Level: Checkpointing implemented using virtual machine functionality [23, 24, 80, 104] aims to provide reliability or bidirectional debugging functionality for the whole guest operating system. Similarly to how libraries or the kernel monitors the application's modifications, the virtual machine keeps track of the modified memory state and creates memory checkpoints, and also records any communication of the VM with the external environment. For the purpose of reliability, the state of the checkpointed VM can be replicated on another system [104], allowing non-interrupted operation if the original system experiences an error.

Hardware Level: Checkpointing at the hardware level [1, 36, 54, 68, 94] allows recovery of the whole system state, not just of a specific application or VM. Hardware assisted checkpointing generally incurs lower overheads comparing to checkpointing at higher levels of the system, and can create checkpoints more frequently. The performance advantage is due to the buffering and delayed copying, which enable hardware techniques [1, 68, 94] to efficiently create checkpoints in the background, without stopping the application's execution. Moreover, the finer tracking granularity, of blocks instead of pages, that hardware uses to track memory modifications, allows hardware to identify more precisely the modified memory locations, and avoid copying non-modified memory that a coarser tracking granularity might checkpoint. These optimizations dramatically reduce the performance overhead of hardware techniques.

2.3 Checkpoint Storage

Checkpoints and logs need to be stored in a location separate from the system's working data. When the checkpoints are used for recovering the system from errors, the storage medium used should have higher resiliency to errors than the part of the system that suffers the error. Checkpoints are typically stored in non-volatile memory, such as hard-disks [30], to survive power failures. Because checkpointed data are rarely accessed, access latency is not a primary concern. However, checkpointing can require non-negligible throughput. Disks for example, may provide limited bandwidth, increasing the construction time of checkpoints. For this reason massively parallel processing (MPP) systems, such as Blue Gene, use high bandwidth I/O subsystems [114]. In the future I/O bandwidth is expected to increase at a lower rate than the processor's computing power. This asymmetry in the system, if not solved, could cause future systems to spend more than 50% of the execution time in checkpointing [61].

To reduce the performance cost of checkpoint storage, some checkpointing techniques store checkpoints in memory [52, 65, 66, 68, 94]. This reduces the resiliency of the system to memory errors (e.g. a bit flip because of an alpha particle) and or power failures. To overcome this limitation in systems with multiple nodes, checkpoints are stored across multiple nodes using memory redundancy, e.g. *N+1* memory parity or other error correcting codes [52, 65]. If a node of the system fails and there is a spare node, then the original memory state of the failed node can be reconstructed using the redundant memory information distributed across the other nodes of the system. Memory-cached checkpoints have lower performance cost than the ones stored on a hard-disk, and can significantly improve the performance of the system. Another proposed memory-based storage solution is to leverage future non-volatile memory technologies such as Phase Change Memory (PCM). Dong *et al.* [22] have demonstrated that the use of such memory technologies can overcome the limitations of checkpointing in future MPP systems.

Another approach to checkpoint storage is multi-level checkpointing [31,52,92,101,102], which creates different types of checkpoints, one cached on the nodes of the computing system in memory, and another one stored in disks. This approach allowed systems like the one proposed by Moody *et al.* [52] to exploit the benefits of both types of checkpoints. Memory-based checkpoints can be

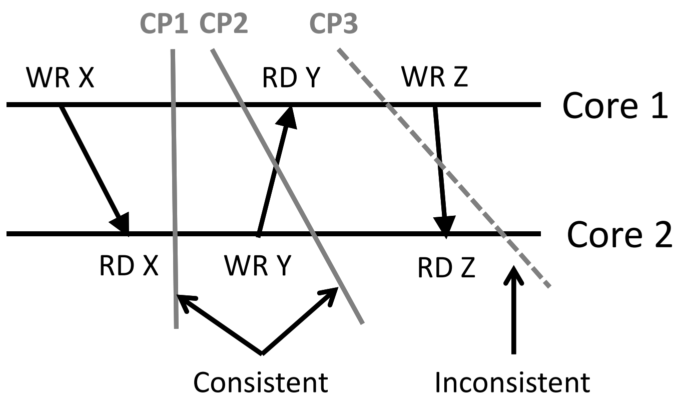


Figure 2: Examples of consistent and inconsistent checkpoints.

created frequently enough, satisfying the checkpointing requirements of future MPP systems, while checkpoints stored on disks can be constructed at a rate that the I/O subsystem can sustain, allowing the system to recover from errors that memory-based checkpoints cannot recover from, e.g. total power failures.

2.4 Checkpointing of Multi-Processor Systems

Checkpointing faces another challenge in a system that has multiple computing elements, such as a distributed system, a multi-processor system, or a multi-threaded application. The challenge is how to create consistent checkpoints, especially when the processing elements of such systems communicate, e.g. through shared memory. Figure 2 shows two cores communicating through shared memory and examples of consistent and inconsistent checkpoints. The first and the second checkpoints (CP1 and CP2 respectively) are consistent checkpoints because, if the system is restored using from CP1 or CP2, the consumers of variables X and Y will not observe the modifications before they actually happen. On the other hand, the third checkpoint will not restore the system to a consistent state, because the second core will have an updated value of variable Z before the first core has actually performed the write.

Consistent checkpoints can be constructed using three methods:

Global: In global checkpointing [36, 47, 54, 55, 65, 66, 68, 94], all processors of the system periodically synchronize to create a single global checkpoint. The synchronization is typically implemented with the help of a two-phase commit protocol [91].

Coordinated: In coordinated checkpoints [1, 2, 4, 5, 66, 109], each processor creates checkpoints locally, while the system keeps track of the communication between the processors. When processor *A* decides to checkpoint, all other processors which have consumed values from *A* have to checkpoint as well. This approach creates a consistent checkpoint similar to checkpoints CP1 and CP2 in Figure 2.

Uncoordinated: In uncoordinated checkpointing [25,26,97,98], individual processors are also creating local checkpoints, but unlike coordinated checkpoints not all processors which have communicated are forced to create a checkpoint together. Interactions between processors are still recorded, similar to coordinated checkpointing, and this information is used only at recovery time. When processor *A* rolls back to a past state, all processors who consumed values from *A* are forced to roll-back as well. The advantage of this approach is that it eliminates the synchronization cost during checkpointing. The disadvantage is that, it is possible under some scenarios that the roll-backs might extend to the beginning of the computation, causing the system to start computation from the beginning, a phenomenon also known as *domino effect* [72].

2.5 Summary

This chapter presented an overview of the different checkpointing techniques and how they can be implemented at different levels of the system. It also highlighted the performance issues that have to be taken into consideration when creating checkpoints, and also the challenges that appear when designing checkpointing solutions for multi-processing systems. Chapters 3 and 4 discuss how these techniques are used for the purposes of bidirectional debugging and reliability respectively.

CHAPTER III

HARDWARE-ASSISTED BIDIRECTIONAL DEBUGGING

Debugging is an important and costly part of software development. It has been estimated [39] that debugging accounts for 60%–70% of the development effort and 80% of project overruns. It has also been estimated that in the early stages of development bugs cost an order of magnitude (50 to 200 times) [11] less to fix than in later stages. Much of the debugging time and effort is spent on back-tracking from the point where an error is manifested (e.g. the program crashes) to the point where the problem originated (e.g. an incorrect value was computed). Typical back-tracking consists of finding the variable directly responsible for error manifestation, finding where that variable was last modified, checking if that modification is a direct result of incorrect computation, and repeating this process if the modification simply propagates another incorrect value.

An example of buggy execution is shown in Figure 3(A), where a zero value in variable X causes the program to crash, e.g. with a divide-by-zero exception. Traditional back-tracking (Figure 3(B)) would involve placing a write watchpoint on X, re-running the program, inspecting the state of the program each time the watchpoint is triggered, and eventually finding the statement that placed the problematic value (zero) into X. In our example, this statement is simply "X=Y" so another back-tracking step (with a write watchpoint on Y) is needed to find where Y became zero. In this example, the buggy code is the one that sets Y to zero, so back-tracking ends there and the programmer can start figuring out how to fix that code.

This traditional approach to back-tracking is both time-consuming and labor-intensive; multiple re-executions of the program may be needed (one for each back-tracking step), often with numerous programmer interactions in each re-execution (to inspect state whenever watchpoints are triggered). bidirectional debugging [12, 27, 80, 112] (also called reverse-debugging) has been proposed as a powerful technique to reduce both time and effort needed for back-tracking. In addition to (traditional) forward execution, bidirectional debugging aids back-tracking by also providing backward

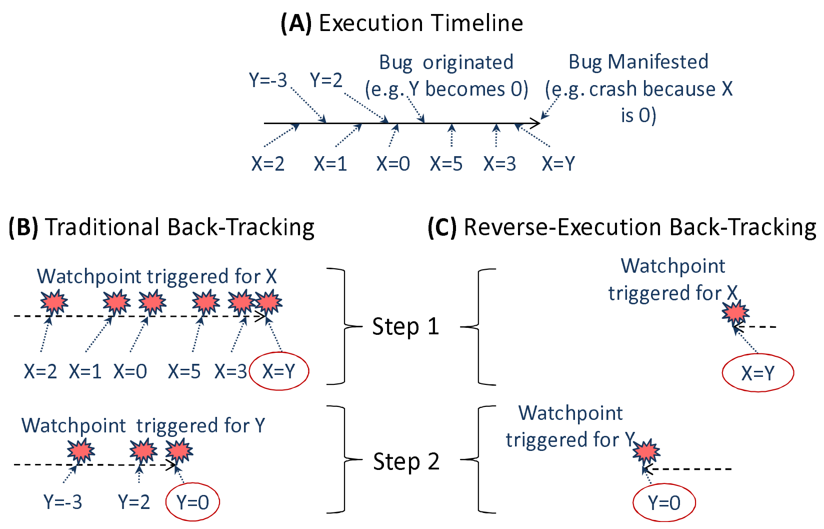

(A) Execution Timeline
Y=-3
Y=2
Bug originated (e.g. Y becomes 0)
Bug Manifested (e.g. crash because X is 0)
X=2
X=1
X=0
X=5
X=3
X=Y
(B) Traditional Back-Tracking
(C) Reverse-Execution Back-Tracking
Watchpoint triggered for X
X=2 X=1 X=0 X=5 X=3 X=Y
Step 1
Watchpoint triggered for X
X=Y
Watchpoint triggered for Y
Y=-3 Y=2 Y=0
Step 2
Watchpoint triggered for Y
Y=0

Figure 3: Debugging example with back-tracking.

(reverse) execution. As shown in Figure 3(C), reverse-execution saves back-tracking time and effort. It saves time by only going through past execution once (not once per back-tracking step). It also saves effort, by involving the programmer only once in each back-tracking step (to inspect code that writes the watched variable).

True reverse-execution is not supported in real processors and systems, so the *appearance* of reverse-execution is typically implemented using checkpointing and deterministic replay, as shown in Figure 4. Checkpoints are created periodically during forward execution (Figure 4(A)). For reverse-execution (Figure 4(B)), a prior checkpoint is restored, re-execution finds the latest occurrence of a watchpoint, the checkpoint is restored again, and in another re-execution we stop where the latest watchpoint was found. As shown in Figure 4(C), the most recent checkpoint interval may have no watchpoint occurrences. In this case, increasingly older checkpoint intervals are replayed until a watchpoint occurrence is found.

From the user's perspective, this checkpoint/replay implementation of reverse-execution is indistinguishable from "real" reverse execution if it has similar "speed" and "reach"[1] as forward execution. For "speed", the time needed to reverse some execution should be similar to the time that was needed to forward-execute it. For "reach", we should be able to reverse-execute from any point all the way to the start of the program, just like we can forward-execute all the way to the end.

[1]Reverse-execution, its "speed", and its "reach" refer to what the user (of the debugger) perceives during bidirectional debugging.

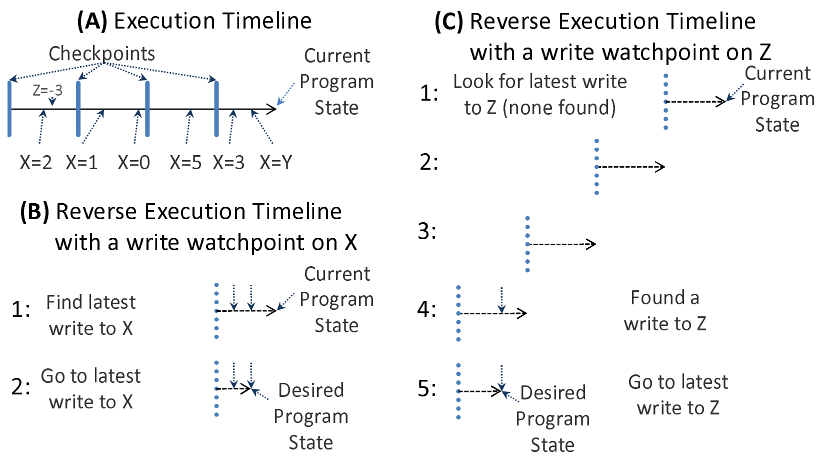

(A) Execution Timeline
Checkpoints
Z=-3
Current Program State
X=2 X=1 X=0 X=5 X=3 X=Y
(B) Reverse Execution Timeline with a write watchpoint on X
1: Find latest write to X
Current Program State
2: Go to latest write to X
Desired Program State
(C) Reverse Execution Timeline with a write watchpoint on Z
1: Look for latest write to Z (none found)
Current Program State
2:
3:
4: Found a write to Z
5: Go to latest write to Z
Desired Program State

Figure 4: Reverse-execution via checkpoint/replay.

Disproportionally long reverse-execution times and/or inability to reverse-execute far enough into the past are likely to frustrate programmers and cause them to revert to "traditional" debugging approaches. Ability to reverse long periods of execution is also needed for analysis of security attacks that may deliberately exploit bugs with long dormancy periods.

Unfortunately, "speed" and "reach" create conflicting demands for checkpointing frequency: "speed" needs frequent checkpointing so short periods of execution can be reversed quickly, but for "reach" checkpoints should be created rarely to retain a lot of checkpoints without running out of space. To achieve both goals, some software-only tools create checkpoints frequently (for short-term "speed" of reverse-execution) and then *consolidate* [12] them as they age (to conserve space over the long term). Unfortunately, software-only checkpointing approaches for reverse-execution [12, 23, 24, 27, 63, 79, 80, 95, 112] have large performance overheads (when checkpointing often enough for good "speed" of reverse-execution), and they cannot efficiently record memory races for deterministic replay of multi-core execution.

Hardware support has been proposed as a way to minimize performance overheads of checkpointing for error recovery [60, 68, 110] and, in combination with deterministic replay, for debugging [60, 69, 110]. Performance overheads are reduced mostly 1) by saving checkpoint data in the background and on-demand, as application execution modifies memory content, and 2) by tracking race outcomes in hardware. In recent years, memory space needed to record race outcomes has been reduced by multiple orders of magnitude [32, 51, 59, 111]. However, the total memory used for checkpoint storage is dominated by saving modified data blocks (Data Log in Figure 5), where hardware schemes could only reduce space consumption (typically by a factor of 2 to 3) using compression of individual checkpoints. In contrast, consolidation would result in orders-of-magnitude space reduction for data logs. Unfortunately, checkpoints produced by existing hardware schemes are not amenable to efficient consolidation because 1) consolidation of two checkpoints relies on finding and eliminating duplicate data blocks (those saved in both original checkpoints), which requires checkpoints to be sorted (or at least searchable) by data address, whereas 2) existing hardware schemes save data blocks on-demand, thus data blocks in each checkpoint are ordered by time of modification (or use), not by data address. As a result, these schemes must sacrifice space (by giving up on consolidation) or performance (by sorting checkpoints after they are created).

This chapter presents two hardware memory-checkpoint accelerators: HARE (Hardware-Assisted Reverse Execution) and Euripus. Both are low-cost hardware solutions which provide efficient checkpointing and enable *consolidation*. The two techniques follow different approaches regarding the type of checkpoints they create. I study the effects that this design decision has on the hardware requirements, the performance and checkpoint storage overheads, and the reverse-execution latency that each technique can achieve. I also propose techniques that leverage the additional functionality provided by the hardware in order to further accelerate reverse-execution. Overall, both techniques incur minimal performance overheads at high checkpointing frequencies, they reduce the memory requirements of long-running applications by more than an order of magnitude by using checkpoint consolidation, and they provide reverse-execution times similar to forward execution times for the same number of instructions.

The rest of this chapter reviews bidirectional debugging (Section 3.1), describes the HARE (Section 3.2) and Euripus (Section 3.4) techniques and their implementation details (Section 3.3 and Section 3.5 respectively), and finally presents a quantitative evaluation (Section 3.6) and conclusions (Section 3.7).

3.1 A Review of Bidirectional Debugging

In addition to commands for executing the application forward (step, continue, etc.), a bidirectional debugger also provides commands for reverse-execution (rstep, rcontinue, etc.). As described, reverse-execution is typically implemented by restoring a checkpoint and then forward-executing to the desired point, using the number of executed instructions to define "position" in the execution time-line. For example, if N instructions have been executed since the last checkpoint, a `rstepi` command (undo one instruction) causes the debugger to restore the last checkpoint and re-execute N-1 instructions. In some cases, the desired point is unknown a priori. A typical example (shown in Figure 4), is when the user (of the debugger) sets a write watchpoint on some variable and then uses a `rcontinue` command (reverse-execute until watchpoint is triggered). The desired point in this case is the most recent write to the watched variable(s). The example in Figure 4(B) shows a situation where the watchpoint is triggered while replaying (phase 1) the most recent checkpoint interval. In this phase, replay continues to find if an even more recent write exists in that checkpoint interval. Once the instruction count for the last write is ascertained, another replay of the same checkpoint interval (phase 2) is used to reach that point. Alternatively, the debugger creates temporary checkpoints as it finds each write in phase 1, then phase 2 consists of simply restoring the most recent temporary checkpoint. If the watchpoint is not triggered while replaying the most recent checkpoint interval (Figure 4(C)), the debugger replays progressively older intervals until it finds one where the watchpoint is triggered.

Assuming that checkpoints can be restored as quickly as they can be created, and assuming that replay is as quick as original forward execution, the time needed for reverse-execution will be equal to the time that was needed for the same execution in the forward direction, plus the time to restore and replay the rest of the target checkpoint interval (once if no temporary checkpoints are created, twice otherwise). Therefore, the "speed" of reverse and forward execution will be similar if the target checkpoint interval is relatively short compared to the amount of execution being reversed. For small amounts of reverse-execution, e.g. reverse-executing only a few instructions, recent checkpoint intervals must be very short (so their replay appears instantaneous), thus leading to very frequent checkpointing (at least several times per second). On the other hand, long-range reverse-execution achieves good "speed" even with longer checkpoint intervals.

Frequent checkpointing that can provide good "speed" for small amounts of reverse-execution leads, over long periods of execution, to huge space overheads. To illustrate this, Figure 5 shows, for 8-threaded execution of PARSEC 2.0 benchmarks, the rate (in GBytes/minute) at which space is consumed by checkpoints when they are created whenever 1 billion instructions are executed

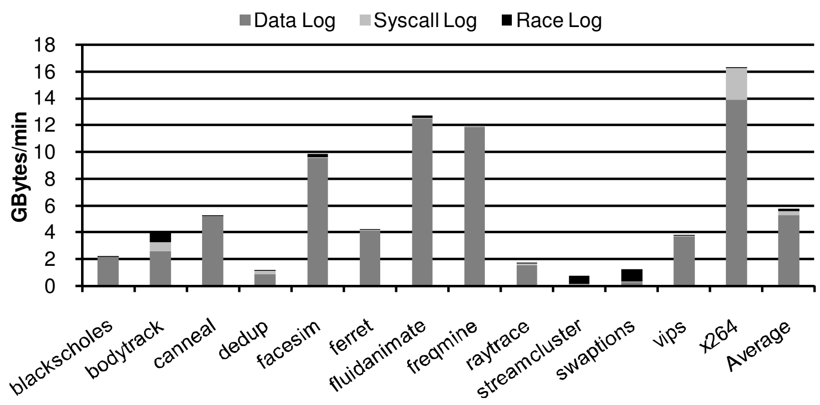


Figure 5: Space used for frequent checkpointing.

(by all threads, resulting in 10-30 checkpoints per second) without using consolidation. We see that facesim, fluidanimate, freqmine, and x264 consume more than 10 GBytes per minute, so good "reach" of reverse-execution would be infeasible for long-running applications (e.g. hours of execution time) with similar memory access patterns.

The checkpointing space requirements in Figure 5 are broken down into checkpointed data (Data Log), system event logs for deterministic replay (Syscall Log), and race logs for deterministic replay of multi-threaded execution (Race Log). Data Log and Race Log sizes are obtained by modeling the approach used in FDR [110], but without using compression (which typically can only reduce these space requirements by a factor of 2 to 3). We observe that Data Log (checkpoints themselves) is the dominant component (by far). This may seem to contradict recent research in deterministic replay, which focuses almost exclusively on reducing Race Log sizes [32, 51, 59, 111], especially knowing that PARSEC is not as I/O intensive as commercial workloads used to evaluate work that focused on recording races and *non-deterministic system events* (such as I/O) [79, 95, 110]. However, results presented for commercial workloads by Xu et al. [110] lead to similar conclusions (data logs are the dominant component).

As described previously, frequently taken checkpoints are only needed to achieve good "speed" for reversing small amounts of execution, and long-range reverse-execution can achieve good "speed" using checkpoints that are further apart from each other. Some software schemes for bidirectional debugging [12] build on this insight by using *checkpoint consolidation* to dramatically reduce the total space consumed by checkpoints, while retaining good "speed" for short-range reverse-execution.

With consolidation, incremental checkpoints are created often, for “speed” of short-term reverse-execution, but are then merged as they age into progressively coarser-grained checkpoints to reduce space requirements while still supporting good “speed” of long-term reverse-execution.

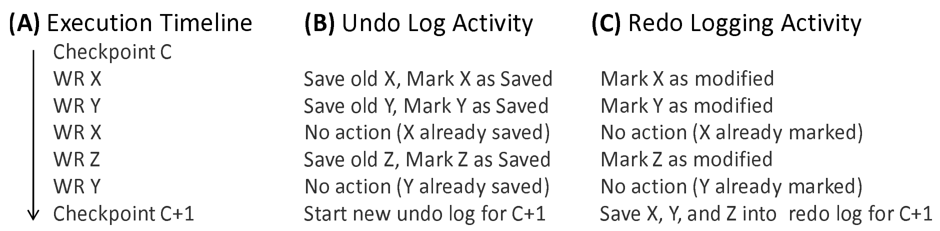
A consolidated checkpoint is the union of its input checkpoints, but without duplicates (data blocks present in both input checkpoints). To efficiently find duplicates during consolidation, checkpoints must be searchable by data address or, preferably, arranged by address so a single merging pass can eliminate all duplicates. This is relatively easy to achieve in software schemes, which can use sophisticated data structures (e.g. search trees) to facilitate checks for duplicates. Additionally, many such schemes identify changes at page granularity [12,27,63,80,95], so there are fewer saved records to search or sort than in schemes with finer (e.g. cache block or even word) granularity.

In contrast, hardware checkpointing schemes [60,68,94,110] dramatically reduce performance overheads of checkpointing by saving data on-demand (without stopping the application) and at finer (typically cache-block) granularity. Sophisticated data structures would result in increased costs, so these schemes simply write out data blocks to a contiguous log in memory as each block is modified (in BugNet [60], when modified data is used for the first time). This results in checkpoint logs with many fine-grain entries that are not sorted by address. Unfortunately, this prevents efficient consolidation: these logs cannot be searched for duplicates efficiently, and sorting them is either time-consuming (if sorting in software) or requires expensive sorting hardware. It should be noted that some hardware solutions [60,94] compress checkpoints, which typically reduces space requirements by a factor of 2 to 3. In contrast, as will be shown in Section 3.6.6, if efficient consolidation could be supported it would provide orders-of-magnitude reduction is total space requirements.

3.2 HARE: Hardware Assisted Reverse Execution

As explained in Section 3.1, to support efficient consolidation we must either 1) sort checkpoints by address after they are created, 2) create checkpoints that are already sorted by address, or 3) use meta-data structures that can be efficiently traversed in order of addresses. Sorting is time-consuming if done in software, and expensive if done in hardware. In HARE, I investigate the second option, i.e. a mechanism that can create already-sorted checkpoints by address.

To efficiently create such a checkpoint, one must 1) know which blocks will be saved before we



(A) Execution Timeline	(B) Undo Log Activity	(C) Redo Logging Activity
Checkpoint C		
WR X	Save old X, Mark X as Saved	Mark X as modified
WR Y	Save old Y, Mark Y as Saved	Mark Y as modified
WR X	No action (X already saved)	No action (X already marked)
WR Z	Save old Z, Mark Z as Saved	Mark Z as modified
WR Y	No action (Y already saved)	No action (Y already marked)
Checkpoint C+1	Start new undo log for C+1	Save X, Y, and Z into redo log for C+1

Figure 6: Checkpointing with undo and with redo logs.

actually save any of them, and at that time 2) efficiently discover these modified blocks in order of address and save them without stopping the execution of the application. The undo-log approach used in prior hardware schemes [68, 94] is incompatible with the first requirement. Figure 6(A) shows an example execution and Figure 6(B) shows the corresponding undo log activity, which saves each old data value just before it is overwritten for the first time in this checkpoint interval. Because data is actually saved during the checkpoint interval, the next checkpoint (Checkpoint C+1) is "created" by simply marking the position in the undo log (or by starting a new log). To restore a prior checkpoint, one would simply restore to memory all values saved since that checkpoint, starting with most recent log entries. However, the entire set of modified blocks is known only at the end of the checkpoint interval – after data from all these blocks is already read out from memory and saved to the log. To produce a sorted checkpoint, considerable on-chip resources would be needed to buffer these blocks (or at least records that contain pointers to saved blocks), as well as expensive hardware to sort these records by address before they can be written out as a sorted undo log. Note that BugNet [60] does not use undo logs, but its logs are sorted by time of first read and thus suffer from similar limitations when it comes to consolidation.

Due to these considerations, for the HARE scheme I forgo undo-log and loaded-value-log approaches, and use redo logs instead. Redo-logging for the execution example in Figure 6(A) is shown in Figure 6(C). A redo-log contains *new* values for modified data, recorded at the end of the checkpoint interval when the set of blocks that must be saved is known and can be written to the log in order of original address. However, redo-logs present two main challenges that should be addressed. First, it is needed to track blocks that are modified during the checkpoint interval and efficiently save these modified blocks (in order of data address) when actually creating the checkpoint. Second, redo-logs can directly support a *roll-forward* from older to newer checkpoints, not

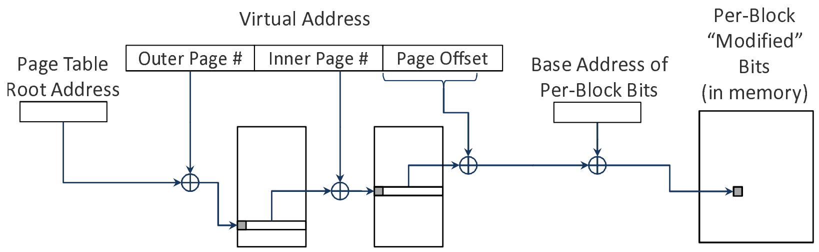


Figure 7: Modification tracking bits (shown in gray).

roll-back from newer to older checkpoints that is needed to provide reverse-execution.

The rest of this section describes HARE's modification-tracking approach, our checkpoint creation approach, how to roll back to past checkpoints created by HARE, how redo logs can be efficiently consolidated, and how logs are organized to reduce memory bandwidth needed for consolidation.

3.2.1 Memory Modification Tracking

Creation of checkpoint C+1 involves saving new values of all blocks that were modified between checkpoint C and checkpoint C+1. To track which blocks were modified, HARE uses a packed bit-array with a "modified" flag (bit) for each memory block in the application's address space, and keeps, looks-up, and updates these per-block bits using a MemTracker-like approach [103].

When creating a checkpoint, a linear search of this bit-array can discover modified blocks in order of their (virtual) address. However, checkpoints will be created often, so only a small fraction of blocks in the entire address space is modified, and blocks that are modified tend to be clustered (because of spatial locality of writes in the application). This means that a linear search of the bit-array would be time-consuming, with most of that time spent scanning through bits that correspond to unmodified blocks.

To improve efficiency of the search for modified blocks, HARE also marks modifications in page tables and TLBs by using an additional dirty bit in each page table (and TLB) entry. Figure 7 shows an example with hierarchical page tables (a page table with only two levels is shown for clarity). When creating a checkpoint, modified pages can be found by identifying modified entries in outer levels, descending towards inner levels for such entries, and eventually searching only the parts of the bit-array that correspond to pages marked as modified. HARE clears "modified" bits in page table entries and the bit-array as it discovers them and saves the blocks they indicate, so when a checkpoint is created the same bits can be used to track modified blocks for the next one. Overall, for each modified page at most one page table node at each level is examined (e.g. 4 nodes for 4-level page tables) and tens of bits in the bit-array (e.g. 64 bits for 4kB page size and 64-byte blocks). Since each modified page contains at least one modified data block that must be copied to the checkpoint, the scan for modified blocks represents only a small fraction (<2%) of the total time and memory bandwidth used for checkpointing.

To simplify HARE's hardware support, it uses a software handler to discover modified pages and create a list of such pages. The hardware checkpointing support then reads this list, scans corresponding parts of the bit-array, and saves discovered blocks to the checkpoint's data log. Note that, as an alternative to using additional page table bits and the bit-vector array, the modification tracking could also be implemented with a single hierarchical meta-data structure, e.g. HARE can use a trie structure from Mondrian Memory Protection [108] with "modified" bits at each level to quickly zero in on modified blocks. The choice of the specific mechanism to track modified blocks is largely orthogonal to the design of our checkpointing and consolidation support, the use of such a data-structure is demonstrated in Euripus (Section 3.4)

3.2.2 Checkpoint Creation

To create a checkpoint, HARE scans the modification-tracking state, finds modified blocks, and saves them to the redo log. In a naive implementation of this approach, the execution cannot continue until the checkpoint completed, as modification tracking for the next checkpoint can interfere with the creation of the current one, and new execution could modify data blocks again before they are saved.

HARE distinguishes between modification marks from the previous and the current checkpoint interval by using two separate modification bits for each block and page table entry. During a particular checkpoint interval, one set of bits is used to track modifications for the next checkpoint, while the other is being scanned and cleared as blocks modified in the previous checkpoint interval are saved. The roles of the two sets of bits are reversed at each checkpoint. In the unlikely case that a checkpoint interval ends while the previous is still being saved, the processor stalls. Such stalls are not observed in any of my experiments – even with very short checkpoint intervals, checkpoint creation completes well before it is time to create the next checkpoint.

To prevent overwrites in the new checkpoint interval from destroying yet-to-be-saved values from the previous one, for each write generated by the processor HARE check's the modification bit from the previous checkpoint interval. If that bit is still set, the block must be saved before it is overwritten. A naive solution would be to stall the processor at this point, but such stalls would occur often[2]. Instead, when a yet-to-be-saved block is about to be modified again, HARE saves this block to another log (called *collision list*), clear its modification bit, and allow the processor's write to proceed. This leaves the main data log for the checkpoint completely sorted by address, but produces an unsorted collision list. Fortunately, our experiments indicate that the collision list is much smaller than the checkpoint list, so it can be quickly sorted in software when checkpoint creation is complete.

3.2.3 Checkpoint Consolidation

During consolidation, the checkpoints are organized in levels based on the number of consolidations that have been performed on a given checkpoint. A new checkpoint is inserted in the lowest level, level-zero. Two level-N checkpoints are consolidated into a new level-(N+1) checkpoint as soon as two level-N checkpoints exist and a third is created by establishing a new checkpoint (for level-zero checkpoints) or through consolidation. This consolidation policy is similar to the one used in software-only schemes [12], and it provides two key benefits: 1) the total number of checkpoints and the total size of all checkpoints at any given time is only logarithmically proportional to the total execution since the beginning of the program, 2) checkpoints at each level of consolidation need to be consolidated only half as often as the previous level, so consolidation work grows slowly and can be bounded by saving very old (and heavily consolidated) checkpoints to disk and not consolidating them further, and 3) the "speed" of forward and of reverse-execution is similar, because replay of each checkpoint interval takes about half as much time as was needed to forward-execute from the start of that interval to the "current" point in the execution.

[2]Blocks with high addresses (e.g. stack) are written often (causing a stall) and checkpointed last, so overlap between execution and checkpointing activity would be minimal

To consolidate two (already sorted) checkpoints into a new (also sorted) one, HARE performs a merging pass that reads the next record from each source checkpoints and compares the addresses from which the two blocks were copied. If the addresses are the same, only the record from the oldest source checkpoint is copied to the consolidated checkpoint and the next record from each source checkpoints is read for the next comparison. If the addresses are different, the record with the lower address is copied to the consolidated checkpoint, while the one with the higher address is retained to be compared with the next record from the other source checkpoint. The result of this merging pass is a consolidated checkpoint that is also sorted by address.

3.2.4 Restoring Checkpointed State

Bidirectional debugging restores past checkpoints in order to provide the effect of reverse-execution. To recover the application to a specific point in time both types of checkpoints undo or redo-log can be used. The debugger can apply the undo-log checkpoints in reverse creation order to restore the application, or redo-logs by applying them in order (starting from a full system checkpoint). The type of checkpoints to use depends on the expected recovery cost which is analyzed in Section 3.6.7.

HARE creates only redo-log checkpoints, which has to convert to undo-log during reverse-execution time. Redo-log entries contain the *new version* of each block that was modified during a given checkpointing interval, while undo-log need the *oldest* one. To convert a redo-log checkpoint C into an undo-log one, for every checkpointed address in C we have to find the value of that address at the beginning of the checkpointing interval. This value can be found by searching all previous redo-log checkpoints in reverse order of creation, starting from checkpoint $C - 1$. The first occurrence of a checkpointed address will hold the necessary value. This search is far more efficient than it seems because 1) most blocks are found after looking in only a few (e.g. one or two) checkpoints because writes tend to exhibit temporal locality, 2) our checkpoints are sorted by address, so binary search can be used to find if a checkpoint contains a given block, and 3) even the worst-case search (all the way to the start of execution) is fairly efficient: consolidation dramatically reduces the total number of checkpoints that are kept at any given time.

3.2.5 Checkpoint Log Organization

The most straightforward checkpoint organization would be an (address-ordered) sequence of records, each containing a block of data and its original address. However, such organization would require saved data to be copied during consolidation. Instead, we separate saved data from its meta-data. Meta-data for a particular checkpoint is kept as a contiguous array of records, each with the block's original address and the pointer to the block's saved data. The system also uses a special "free list" meta-data array that points to free blocks in the saved-data-block space. Consolidation now leaves saved data blocks in place, and only performs a merging pass on the meta-data arrays to create a consolidated one, while records for duplicates go to the free list to be reused when new checkpoints are created.

3.3 HARE: Implementation Details

This section describes how HARE can be implemented and how it can be integrated with existing system and race logging approaches and with the operating system.

3.3.1 Caching Memory Modification Bits

The proposed memory modification tracking allows efficient discovery of modified blocks during checkpoint creation. However, on each store instruction the processor must look up and possibly update these structures, so they should be cached for efficiency. The "modified" bits in page table entries are equivalent to existing "dirty" bits that are used to support virtual memory, so we can extend existing TLB look-up and update mechanisms [33]. The per-block modification bits are kept in a packed bit-array format, but can be cached in primary caches by extending the block's tag array to also keep its "modified" bits (as shown in Figure 8). This approach was called "Interleaved"caching by Venkataramani et al. [103] and was rejected as not flexible enough for their MemTracker scheme (which uses a variable number of bits per word). Since my HARE technique uses only two "modified" bits per block, it can use this simpler caching approach in L1 caches. For L2 caches, the same approach used in MemTracker is adopted – each L1 miss fetches the data and its state separately, so the L2 cache is unmodified and memory blocks that belong to the bit-array are cached normally and simply compete for L2 cache space. This is not a concern because each block of "modified" bits

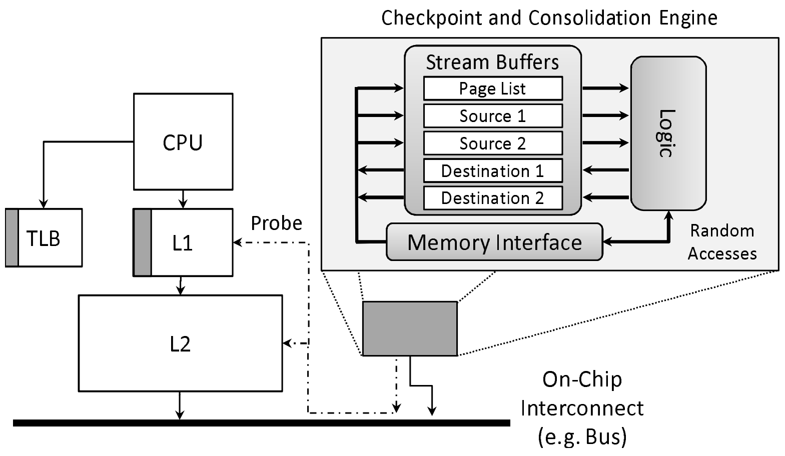


Figure 8: Hardware support for HARE, with added hardware shown as gray.

corresponds to many (e.g. 256 with 64-byte blocks) data blocks. Note, though, that for consistency issues writes have to be appear to happen atomically to both the data and the meta-data to the rest of the system. So a write-back from L1 to L2 proceeds only if both the data and the meta-data are in L2, otherwise it is stalled.

3.3.2 Checkpoint/Consolidation Engine

The bulk of checkpoint creation and consolidation work in my HARE scheme is implemented in a separate engine, as shown Figure 8. It has a memory interface that is used both to 1) read data blocks from and save them to memory and 2) to provide memory access for several stream buffers. Several input and output stream buffers are used to read and write log records, and an additional stream buffer is used to read the list of modified pages (see Section 3.2.1).

When creating a checkpoint, the engine reads the addresses of modified pages, fetches the corresponding per-block “modified”bits from the bit-array, and finds which blocks are marked as “modified”. For each such block, it gets the next free-list record, fetches the block from memory and saves it to the space indicated by the free-list, and adds a new record (with the block’s original address and the address it was written to) to the checkpoint’s log. To get up-to-date values, e.g. when reading modified data blocks that may still be in the writer’s cache, the HARE engine must participate in coherence, generating read requests to obtain data blocks (from either memory or caches) and generating invalidations for memory blocks that are written (logs and clearing of “modified” bits).

When consolidating two checkpoints (as described in Section 3.2.3), the two input stream buffers are used to read records from their logs, and output stream buffers are used to write out the consolidated checkpoint log and to add entries to the free list. However, after a checkpoint is established by our scheme, it consists of two sorted logs (already-sorted data log and software-sorted sorted collision list). To consolidate two such "bifurcated" checkpoints, we can either 1) "consolidate" each checkpoint's data log with its own collision list, then consolidate the two actual checkpoints in another merging pass, or 2) extend the HARE engine with two more input stream buffers and additional logic to handle a four-way merge. In my experiments I use the second approach (four-way merge) to simplify our software handlers and to achieve more efficient consolidation.

3.3.3 System and Race Logging

The HARE mechanism focuses on creating data checkpoints and efficiently consolidating them. It can be used together with existing hardware data race recording mechanisms [32, 51, 59, 110, 111] to allow accurate deterministic forward and backward execution for debugging in multi-core machines. Similarly, non-deterministic system events must be recorded so they can be accurately replayed. The type of event and its timing can be captured by the system at the point when control is transferred (via interrupt, I/O system call, etc.) from the application to the system code. However, changes made by the system in the application's address space must also be captured, and it would be impractical to instrument the operating system to track such changes. Previous hardware implementations [60] capture these modifications by terminating the current checkpoint interval and establishing a checkpoint just before the system event is executed, processing the event normally, then establishing a new checkpoint after the event completes.

The HARE mechanism already has two sets of "modified" bits, which allows it to capture system-event modifications without terminating the current checkpoint in the application. When the system event (e.g. system call) occurs while one set of bits is clear (the previous checkpoint is complete), HARE uses that "free" set of bits to track memory modifications while in system code. When returning to the application, HARE can "checkpoint" only those changes to the system event log (which also resets these "spare" modification-tracking bits), include them to the overall modified state of the current checkpointing interval, and then continue tracking changes in the application's current checkpoint interval using the original set of "modified" bits.

If a system event occurs while both sets of “modified” bits are in use (the application’s previous checkpoint is still being created), the application is simply stalled until that checkpoint is complete, at which time one set of “modified” bits is free. Note that this stall would have to happen even if a checkpoint interval was to be ended before the system event – a third set of “modified” bits would be needed to continue executing while *two* prior checkpoints are still being created.

Finally, note that, because race and system logs cannot be consolidated, their size grows linearly as execution proceeds. Data logs, which without consolidation would also grow linearly and represent most of the overall memory consumption, now grow only logarithmically. Over long periods of time (e.g. minutes or hours) consolidation-reduced data logs eventually become smaller than the ever-growing race and system logs, thus putting priority again on improvements in race and system log recording. In effect, consolidation is *necessary* to keep data log sizes at bay and allow race and system log optimizations to have an impact.

3.3.4 OS Interaction

The HARE approach uses Operating System (OS) software handlers to reduce hardware complexity and ensure correct operation. HARE relies on the OS to determine (typically via timer interrupts) when to create a new checkpoint. At that time, the OS walks the page tables and builds a list (actually, a contiguous array) of modified pages. It then modifies control registers to switch to the other set of modification tracking bits, configures the checkpoint/consolidation engine to start checkpoint creation, and allows the application to continue. If the engine is still creating a previous checkpoint, for that application, the application is suspended until that checkpoint is complete. If the engine is busy doing consolidation, the OS saves the internal state of the engine and starts it on checkpoint creation (consolidation can be delayed without stalling the application). The engine generates an interrupt when its current task is complete, at which point the OS checks if another task (consolidation or checkpointing) should be initiated.

Another OS-related issue arises when swapping out a page that is marked as modified in the current checkpoint - HARE’s engine will not be able to read the block’s data. If the OS keeps a sufficiently large free-page pool, we can delay page eviction until the end of the checkpoint interval (with our frequent checkpointing, this is only a minor delay). For urgent page evictions, modified blocks can simply be saved to the conflict list prior to eviction.

3.4 Euripus: Hardware Assisted Reverse Execution Using a Trie

The previous two sections described HARE, a memory checkpointing accelerator which constructs redo-log checkpoints. HARE's apparent advantage is the minimal hardware cost, which will be demonstrated in Section 3.6, but it has two primary shortcomings: first, it requires the conversion of redo-logs to undo-log checkpoints before reverse execution (which adds to its latency see Section 3.6.7); second, it requires frequent software intervention to generate the list of modified pages and to sort the collision list.

Euripus[3] aims to eliminate these shortcomings of HARE, and also further improve reverse-execution functionality. The goals of Euripus are: 1) reduce the reverse-execution latency, 2) propose a hardware technique that is self-contained and requires fewer modifications to existing hardware structures, e.g caches, and 3) does not require frequent software intervention. To reduce the reverse-execution latency, Euripus creates undo-log checkpoints directly, while retaining the original memory benefits of checkpoint consolidation. At the same time, Euripus aims to provide similar functionality to software techniques [80] by constructing both undo and redo-log checkpoints, but limits the additional performance and memory costs (relative to when undo or redo-logs are constructed alone). Finally, Euripus uses more sophisticated data-structures to keep the checkpoint meta-data, and also stores additional information that will allow to accelerate reverse-execution.

3.4.1 Undo-Log Construction

The primary reason that HARE (Section 3.2) opted for creating redo-log checkpoints over undo-logs was because redo-log checkpoints can be constructed to be sorted by address (Figure 6), while the entries of typical undo-log checkpoints are inserted by the order of modification (resulting in unsorted checkpoints). Consolidation of such undo-log checkpoints would require them to be sorted first, resulting in high performance overheads.

Euripus takes a different approach: instead of a contiguous log used in HARE, it uses a data structure that can still be updated at minimal cost but can be traversed in order of addresses. For this

[3]Euripus is named after the Euripus strait in Chalkis, Greece where the direction of the flow of the water changes periodically because of the tide (http://en.wikipedia.org/wiki/Euripus_Strait).

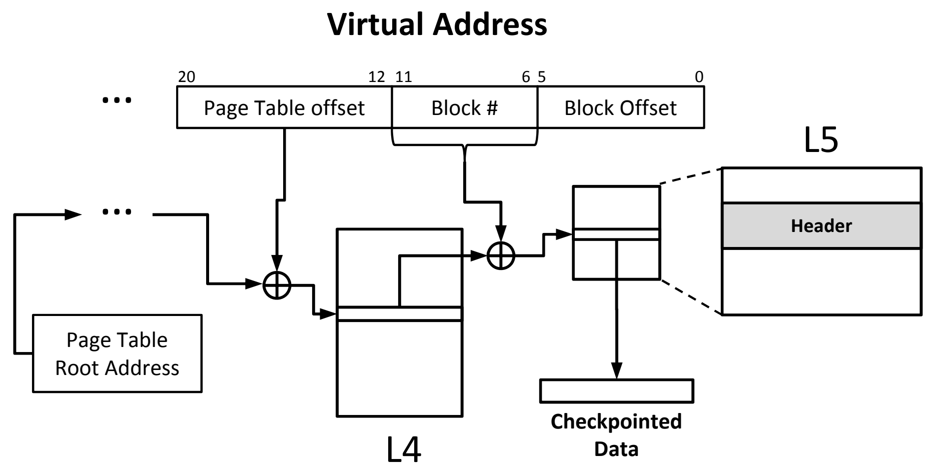


Figure 9: Trie data-structure used by Euripus for representing the undo-log checkpoints.

purpose Euripus uses a trie data-structure (Figure 9). This data structure is a simple extension of the existing page-table structures used in today's processors [33] for virtual address translation. I extend this structure with an additional fifth level[4], which will store pointers to the checkpointed data for every memory block (64 bytes) within a page. The advantages of using this data structure are: it can easily be indexed by hardware to add a block to the checkpoint (similar to existing page-tables), and it can be traversed in address-order during consolidation. In contrast a simple list of blocks is easier to add an entry, but needs sorting prior to consolidation.

3.4.2 Redo-Log Construction

The type of checkpoint necessary for reverse-executing through the programs execution is the undo-log, because by applying it we can "move" to an older program state. The further the user needs to move to the past (and closer to the beginning of the applications execution) the more undo-log checkpoints the debugger will have to apply. In such a case, the program state could be reconstructed using redo-log checkpoints, because fewer of them would have to be applied to recover the memory state. Moreover, if the user wants to move to a future state, after rolling back to a past one, he need not re-execute the whole program: the debugger could directly apply redo-log checkpoints and recover the future program state, as implemented in existing software solutions [80].

The concurrent construction of redo-log alongside with undo-log checkpoints can serve as a

[4]The existing page table in x86 64 bit architectures has 4 levels.

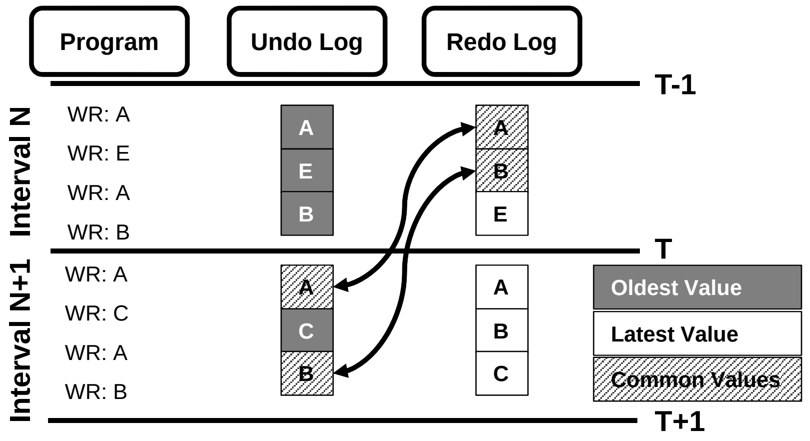


Figure 10: Synergies that develop when both undo and redo-log checkpoints are created.

performance optimization for the purposes of reverse-execution, and should be provided at a minimal performance, memory and implementation cost. To construct redo-log checkpoints, Euripus could use the redo-log mechanism proposed from HARE, but that would increase the overall implementation and performance costs, and also nearly double memory requirements for keeping the checkpoints. Instead, I have selected to extend the undo-log construction mechanism in Euripus to also support construction of redo-log checkpoints.

When both undo and redo-log checkpoints are being created, the following two properties apply (Figure 10):

- For the same checkpoint interval, both the undo and the redo logs checkpoint data from the same addresses, but they copy different data values: the undo-log copies the oldest data value whereas the redo-log copies the newest value seen during the checkpoint interval.
- For addresses that appear in both the redo-log checkpoint of interval *N* (newest values) and the undo-log checkpoint of interval *N+1* (oldest values) their data values are the same.

These two properties allow Euripus to leverage the existing undo-log mechanism, and to reduce the redo-log construction cost using the following methods: First, Euripus can use the undo-log meta-data of interval *N* to identify the blocks to be checkpointed for the redo-log. This approach allows me to remove the redo-log memory modification tracking mechanism used in HARE: the undo-log meta-data become the redo-log's memory tracking mechanism. This eliminates both the performance cost of memory tracking and the memory cost of storing redo-log memory tracking

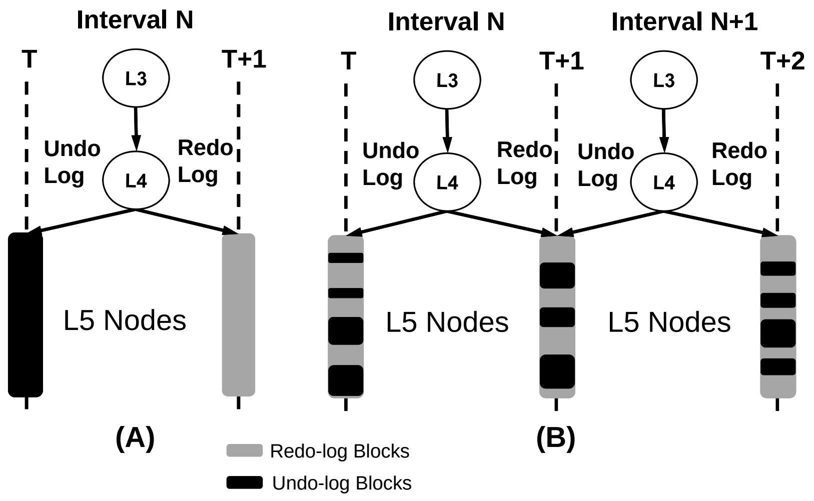


Figure 11: Extended trie data-structure used in Euripus when constructing both undo and redo-log checkpoints.

information, and it also eliminates the additional cache design complexity necessary for atomic reads and write-backs of both data and meta-data from the L1 to the L2 cache. Second, Euripus can avoid copying the same data value twice, once for the undo and once for the redo-log checkpoint. Instead, Euripus can identify the common blocks and copy them only once when constructing either the undo or the redo-log checkpoint.

To support redo-logs, the undo-log meta-data are extended (Figure 9) as shown in Figure 11. First, for every given interval, a single trie meta-data structure holds both the undo-log and the redo-log meta-data (Figure 11(A)). This is implemented by extending the L4 nodes of the trie to have pointers to L5 nodes of both undo and the redo-log checkpoints[5]. The undo-log construction process will update the L5 nodes of the undo-log only, and at redo-log creation time we will walk through the L5 undo-log nodes to find the blocks to be checkpointed by the redo-log.

To identify blocks that are common between consecutive redo and undo logs, the L5 trie nodes are shared between the redo-log of interval *N* and the undo-log of interval *N+1* (Figure 11(B)). This solution allows Euripus to delay the construction of redo-log *N* and not begin it immediately after the end of checkpoint interval *N*. Instead, Euripus can allow the construction of undo-log *N+1* to begin, which results in updating the common L5 meta-data nodes with pointers to copied blocks. When the construction of redo-log *N* eventually begins, e.g. in the middle of interval *N+1*, it will not

[5]The L5 nodes store the pointers to the checkpointed data.

have to copy again blocks already copied by undo-log *N+1*. This approach also naturally eliminates block *collisions*, which HARE defined as the redo-log blocks that had not been checkpointed yet and were modified by the application[6]. HARE inserted the collision blocks in an unsorted list, and at consolidation time the software was responsible for sorting the collision list. In Euripus, blocks are always inserted in the save trie data-structure, and writes from the application are beneficial to redo-logs: they reduce the number of blocks to be copied.

3.4.3 Checkpoint Consolidation

Consolidating the checkpoints from two intervals *N* and *N+1* requires the consolidation of both the undo and the redo-logs from these two intervals. For undo-logs, Euripus has to retain the oldest data across the two checkpoints: if both intervals have checkpointed the same address, Euripus keeps the copy from interval *N*, along with unique addresses from both checkpoints. Conversely, when consolidating redo-log checkpoints, Euripus has to keep the newest value of a given address: for addresses that are common across intervals, the block to keep is the one checkpointed for interval *N+1*.

The advantage of the meta-data organization from HARE was that consolidation just required a linear pass through the two lists of addresses that represented the checkpoints. For Euripus, the same process would require (e.g. for the case of redo-logs), for every block of checkpoint *N*, to look-up the trie of interval *N+1* and search if there is an entry, which is more expensive than a simple linear list traversal. Two characteristics of Euripus can accelerate the consolidation process: 1) the L5 meta-data nodes are being shared between consecutive checkpoints, and 2) for a given interval a block marked as redo-log checkpointed implies that there is also an undo-log block and vice-versa. Thus, by traversing only the undo-log L5 nodes of interval *N+1*[7] we can consolidate the checkpoints as follows:

- If a block is marked as both a redo-log block of interval *N* and an undo-log block of *N+1*, it can be removed, because there already exists an older undo-log block in interval N and a newer redo-log block in interval *N+1*.

[6]The common blocks across checkpoints are the collision blocks.

[7]The same nodes belong to the redo-log of interval *N*.

- If a block is marked only as undo-logged in interval *N+1* (similarly if it was redo-logged in of interval *N*) then we have to search the undo-log *N* (redo-log *N+1*) for an older (newer) block, and if such a block is found then recycle the block. If no such block is found, insert it in the respective tree.

Note that this changes the way that consolidation is done, whereas the policy for selecting the rate of consolidations, and the checkpoints to consolidate, remains the same as HARE (Section 3.2.3).

3.4.4 Improving Reverse Execution Latency

As described in Section 3.1, operations like "reverse-continue" search for the last time a given address (watchpoint) was read or written. This search involves a replay of all the past checkpoint intervals in reverse chronological order until a matching read/write is found. A way to accelerate this search would be to replay only the intervals where this given address has been actually read or written. For the case of write-watchpoints, we can benefit from the existing checkpoint meta-data information and replay only the intervals in which a given address is checkpointed (which implies that it has been written at least once during that interval).

nFor read watchpoints the same approach can be used: when the application reads a block for the first time within an interval, the address of the accessed memory location is sent to the checkpointing engine. Unlike writes, no data needs to be sent for reads, because the engine is just going to record that the block was read. For recording the read information, we extend the trie with additional information bits in the L5 meta-data nodes to mark the read blocks. During consolidation, the read information is consolidated as well: when consolidating two undo-log checkpoints *N* and *N+1*, the read information of the consolidated checkpoint is going to be the `OR` of the read information of the individual checkpoints.

The exact same approach can be used for the case of breakpoints as well, where Euripus is going to mark the code addresses that have executed. For the purpose of implementing and evaluating this technique it is going to be applied only for the read and write watchpoints. This approach can also be applied to HARE, and Section 3.5.5 discusses the implementation cost for both HARE and Euripus.

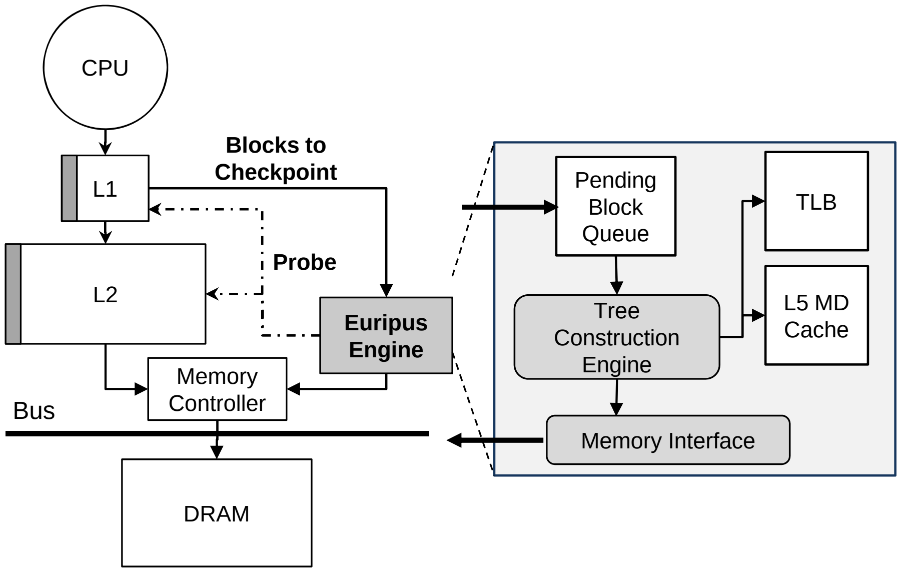


Figure 12: The Euripus hardware engine.

3.5 Euripus Implementation

For constructing the two types of checkpoints and for updating the trie meta-data described in Section 3.4, Euripus uses a hardware engine (Figure 12). The rest of this Section describes the functionality of the engine and how it interacts with the rest of the system.

3.5.1 Undo Log Creation

For identifying the memory locations to be inserted into the undo-log, a *checkpoint* bit is introduced in the L1 and L2 caches. The checkpoint bit behaves similarly to the existing *dirty* bit. When a block is written in L1 we check if the checkpoint bit is set; if not, the block has not been checkpointed in the current undo-log interval, so it is sent to the Euripus engine and the checkpoint bit is set. The checkpoint bit information functions as a first level filter of blocks to be checkpointed: without checkpoints bits every written block would have to go to the Euripus engine. When the block is written back to L2 the checkpoint bit goes with the data. Unlike HARE, where the memory tracking information is stored separately from the data, in Euripus it is stored together with data in caches. As a result, Euripus does not need to address the consistency issues of data and memory tracking meta-data as HARE does. When a checkpointed block is replaced from L2, we lose the checkpoint bit information, and if this block is written again in the future it is going to be sent again to the

Euripus engine. This mechanism is similar to existing hardware checkpointing techniques such as ReVive [68] and SafetyNet [94]. Duplicate blocks in undo-logs are only a performance and not a correctness concern, as undo-logs are restored in reverse construction order and restored duplicates get overwritten with the correct oldest values. In Euripus, we perform a secondary filtering of the blocks to be checkpointed by checking if the block is already copied using the trie meta-data. This eliminates the possibility of duplicate entries from the final undo-log.

Once a block reaches the Euripus engine, it is inserted in the *Pending Block Queue* (PBQ). If the PBQ is full, the core sending the block to be checkpointed will have to delay the original write until PBQ has available entries. Once the block reaches the front of the queue, it is given to the *Tree Construction Engine*, which updates the undo-log part of the trie meta-data for the current checkpoint and checks if the block has already been checkpointed for the current interval *N+1*. Then the engine checks if the L5 meta-data node of the current undo-log has already been inserted as a L5 redo-log node of the previous checkpointing interval *N*, and if not it updates the trie meta-data of the previous interval. This operation indirectly inserts the blocks already checkpointed by the current undo-log in the previous redo-log, which is not going to copy them again. The block is then sent to the *Memory Interface* of the engine and written to memory. Once the write acknowledgment is received, the block is marked as *checkpointed* and as an *undo-log* block.

To efficiently access and update the current and previous trie meta-data structures, the Euripus engine has a TLB to store the translations for the last level L5 Nodes, and a cache, named *L5MD cache*, to store the contents for the L5 Nodes which are frequently accessed.

3.5.2 Redo Log Creation

The sets of modified addresses of consecutive checkpointing intervals do not overlap entirely, especially at high checkpointing frequencies where the application does not have sufficient time to modify a substantial part of its working set. For this reason, we have to eventually start actively copying blocks for the redo-log of the previous interval *N*. At the latest, this active copying should complete at the end of the current interval *N+1*. Euripus starts the construction of redo-log *N* in the second half of the current checkpointing interval, e.g. for a checkpointing interval of 0.1sec, redo-log construction starts after 0.05sec.

To construct the redo-log checkpoint, the *Tree Construction Engine* of Euripus starts walking the trie meta-data in order, and checks if a block has been copied by the undo-log checkpoint in the previous interval *N*. If yes, then this address has also to be checkpointed by the redo-log checkpoint of interval *N*, and the block is inserted into the redo-log part of the trie. Then the engine checks if the *L5* node is being shared by the undo-log of the current interval *N+1*. If the node is not shared, which happens rarely, it is inserted in the trie of the current undo-log *N+1*. This step is necessary for ensuring that all overlapping *L5* nodes are shared between consecutive tries, and improves performance by eliminating block copies by the undo-log that have already been checkpointed by the redo-log[8]. Finally, the block address to be copied is forwarded to the memory interface that reads and writes the data to memory. The data to be checkpointed can reside in any level of the memory hierarchy and for this reason the Euripus engine, similar to HARE, has to probe the caches. Once the write acknowledgment arrives, the trie meta-data is updated with the address of the saved block and the block is marked as *redo-log* and *checkpointed*.

3.5.3 L5 Node Meta-Data Organization

The L5 meta-data nodes store, for every memory block, the pointer to the address where it is being checkpointed, and 3 meta-data bits describing whether it is *checkpointed* and if it is part of an *undo* and or a *redo-log*. Since these bits are accessed frequently during undo/redo-log checkpoint creation and consolidation, we improve the data locality of the *L5MD* cache by packing the meta-data bits of all blocks of the L5 node inside a header (Figure 9). The location of the header within the L5 node is decided based on bits 12-18 of the virtual address, in order to avoid possible address aliasing to the same cache sets and poor L5MD cache performance.

3.5.4 Interaction with the OS/VM

The Euripus engine is managed by the OS/VM and is equipped with the necessary registers to store, for every currently running process on the processor, the roots of the trie meta-data structures, as well as the core-id of the core where a given process is running. An undo-log block to be checkpointed is accompanied by the id of the core where it was modified and gets inserted in the

[8]In such a case the undo-log construction process will mark the block is *undo-log* but will not checkpoint it.

appropriate tree. The OS/VM is also responsible for providing and managing a list of free memory blocks, similar to HARE.

Euripus has to address the system call problem as well. For the case of redo-logs Euripus has to include the modified program state by the system call in the state to be checkpointed and has to assist the generation of the system log. For undo-log checkpoints, the old values of the memory locations modified by the system call have to be recorded. Note that the purpose of the undo-logs is not to "undo" the effect of the system-call, e.g. undo a disk write of a network sent packet, but only to recover the program to a past state[9].

In modern operating systems the kernel and the application share the address-space, typically half of it is reserved for the kernel and the rest belongs to the application. Since Euripus creates checkpoints at the application level for the purpose of reverse-execution, it does not need, and should not be, checkpointing data modified in the kernel-space. To implement this, Euripus marks kernel-space addresses as non-checkpointable by marking the parts of the trie that correspond to the kernel address-space as such. This can be implemented by adding an additional *non-checkpoint* bit in every pointer to a node of a trie. By marking a pointer at a higher level of the trie, e.g. level zero, then all the addresses that belong to the range represented in the sub-tree pointed by the pointer are not going to be checkpointed. When a block is to be checkpointed and inserted in the trie, while it walks the trie, if it finds the non-checkpoint bit set, at any level of the trie, then it does not get checkpointed. When the system-call finishes it copies its output back to the user-space[10], at this point Euripus will copy the old values of the user-space data in the undo-log. Once the blocks modified by the system call are checkpointed by the undo-log they become automatically part of the modified state for the redo-log of the same interval and are going to be checkpointed.

Euripus can assist the system log creation in a similar fashion as HARE does. An additional *system-call* bit is added in every pointer of the trie, thus marking the addresses of the user-space that have been modified by the kernel – the core will have to send to the engine whether it is operating in kernel mode or not for a block to be marked as system-call modified. At the end of

[9]For the purposes of reliability the system wants to be able to prevent permanent effects for erroneous execution and thus treatment of system events when checkpoints are created is different. More details will be provided in Section4.4.4.

[10]None of the user-space addresses would be normally checkpoint protected.

the system call the system log creation mechanism can find the addresses modified by the system-call and record their latest values in the system-log (the system-call bits are cleared at the end of this process). Unlike HARE, Euripus does not have to wait for the construction of the previous redo-log to finish before proceeding in a system call. Euripus's trie meta-data is amenable to any modification, because any changes will affect the design of the Euripus engine alone and leave the rest of the processor design unaffected. For the case of system calls a third memory tracking bit could have been added in HARE to track the system-call modified memory locations, but such a modification would require to change the L1 cache design as well – memory-tracking meta-data are cached along side with the block in the same cache-line.

3.5.5 Improved Reverse Execution

For keeping track of the blocks read within a checkpointing interval an additional *read-bit* is introduced in the L1 and L2 caches, whose behavior is identical to the existing checkpoint-bit that Euripus uses. The first time a block is read the core checks the read-bit, and if it is not set then it sends the address of the block to the checkpointing engine, and sets the read-bit. Similar to checkpointing a block, if the engine does not have available entries in the pending-block queue, then the execution of the processor has to stall until the hardware engine becomes available. The same functionality could be implemented using HARE, which would require to add the extra read-bit in the L1 cache and extend the memory tracking meta-data with the additional information.

3.6 Performance Evaluation of HARE and Euripus

I quantitatively evaluate the HARE and Euripus techniques in terms of estimated hardware cost, performance overheads in forward execution, memory requirements for long-term checkpointing, and reverse-execution time.

3.6.1 Evaluation Setup

In the evaluation, I use SESC [76], an open source execution driven simulator, to model a four-core CMP system with Core2-like parameters: 4-issue, out-of-order cores running at 2.93GHz. Each core has a private dual-ported 32KB 8-way associative L1 data cache. All cores share a

4MB, 16-way associative, single-ported L2 cache. The block size is 64 bytes. A detailed DRAM-model is used to simulate a DDR3-800-like memory system. For the HARE mechanism, an on-chip checkpoint/consolidation engine is modeled that participates in coherence and has (fully snooped) 32-entry read and write queues. The simulated Euripus engine has a 64 entry pending block queue, a 256 entry fully associative trie TLB, a 64KB 16-way associative single-ported L5MD cache, and the memory interface has a 32 entry read queue and a 32 entry write queue. In the evaluation, I am using the SPEC2006 [96] benchmark suite, and I simulate 27 out of 29 benchmarks (Figures 14(a) and 14(b)) using the reference inputs. I omit perl and tonto because of technical incompatibilities with the simulator infrastructure. In the simulations I fast forward though 5% of the simulation, with a maximum of 20 billion instructions, to skip initialization, then simulate 10 billion instructions. For evaluating multi-threaded applications I use the PARSEC [8] (Figure 14(c)) benchmark suite with the *native* input, except in the case of dedup where I use the *simlarge* input, because the application allocates more than 2GB of memory, and exceeds the 32-bit address-space simulated by SESC. For PARSEC benchmarks, I skip to the beginning of the parallel section, then skip an additional 21 billion instructions, and finally I simulate 20 billion instructions. During the fast-forwarding processes of each simulation, the mechanism of each scheme simulated (Table 1) warms-up the memory-tracking meta-data, constructs checkpoints at the same checkpointing frequency (assuming an IPC of 1 for the applications), and consolidates the meta-data of the constructed checkpoints. For estimating the final checkpoint memory requirements of each technique, I use PIN [45], where I profiled the previously listed SPEC2006 and PARSEC applications to completion, using the reference and native inputs respectively. Table 1 summarizes the different techniques used in the evaluation, their type (software or hardware), the type of checkpoints constructed (undo or redo-log) and the type meta-data data-structure used for representing the checkpoint. Following paragraphs describe in detail how different techniques construct their checkpoints.

3.6.2 Hardware Cost

To estimate the hardware requirements of HARE and Euripus I used CACTI 5.3 [99]. For the case of HARE the overall internal state of HARE's checkpoint/consolidation engine (read and write queues, stream buffers, internal latches, etc.) uses less than 5% of the area occupied by one L1 cache, while

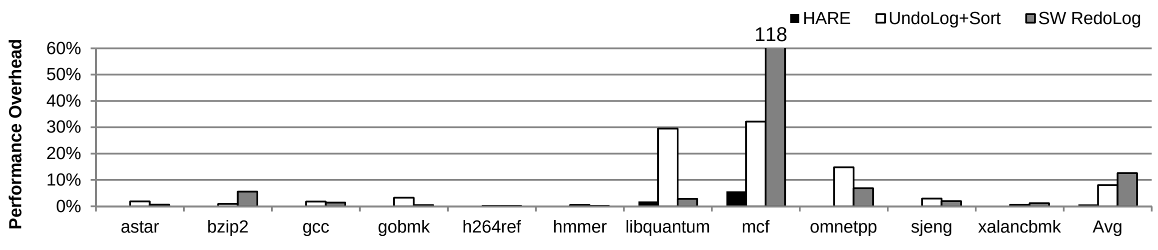
Table 1: Type of checkpoint constructed and meta-data data-structures used for checkpoint representation by the evaluated techniques.

Technique	Type	Undo Log	Meta-data	Redo Log	Meta-data
HARE	HW			✓	List
UndoLog+Sort [68]	HW	✓	List		
SW RedoLog	SW			✓	Page Table
Euripus	HW	✓	Trie		
EuripusUR	HW	✓	Trie	✓	Trie
Hare+UndoLog	HW	✓	List	✓	List
TTVM [80]	SW	✓	Page Table	✓	Page Table

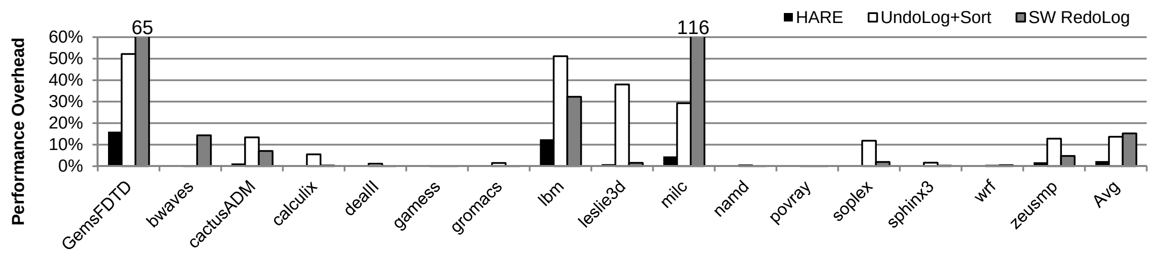
caching of our "modified" bits adds a <1% area overhead to each L1 cache. Finally, the bit-array for out modification tracking is stored in memory and uses only 0.4% of the memory used by the application being debugged. Overall, hardware support for the HARE mechanism amounts to about 6KB of fast on-chip SRAM and represents a negligible fraction of the overall chip area. Its cost is considerably lower than 48KB reported for BugNet [60] or 1416KB for FDR [94], so I expect HARE to add little cost when it is combined with existing race logging mechanisms. Euripus's functionality has higher hardware requirements than HARE. Euripus requires approximately 79KB of on-chip memory state, whereas the HARE uses 6KB, but is still lower than 256KB needed by SafetyNet [94]. The area estimates of Euripus using Cacti 5.3 [99] is 20% smaller than the size of the L1 Data cache, because Euripus's hardware structures do not need to be optimized for speed as the L1 Data cache does (e.g. Euripus's cache has only one port compared to the two ports that the L1 cache has to have for performance reasons). I also used CACTI to estimate the increase of access latency that the additional meta-data bits that HARE and Euripus introduce in the L1 and L2 caches, and observed no difference compared to the baseline.

3.6.3 HARE: Performance Overhead Evaluation

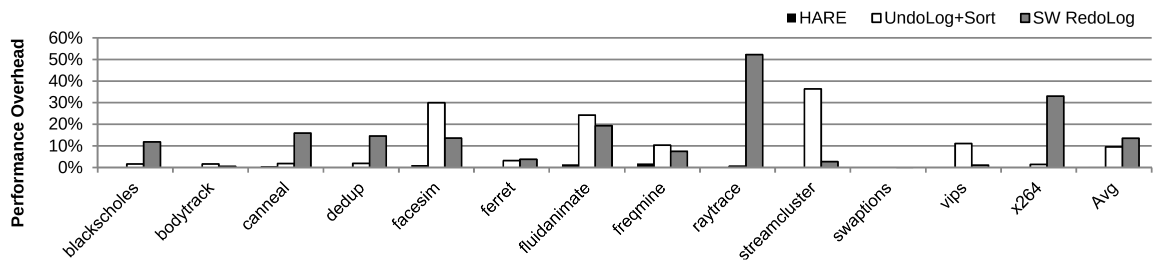
Figure 13 presents the performance overhead of HARE when checkpoints are established every 0.5 sec. I am also comparing the performance overhead of HARE with other approaches for creating consolidatable checkpoints. First, HARE is compared with a hardware technique that constructs undo-log checkpoints, similar to prior hardware techniques like FDR [94] and ReVive [68], but has to sort the resulting undo-log checkpoints in order to consolidate them (called UndoLog+Sort in Figure 13). For this technique the overhead of establishing the checkpoint is simulated, by copying



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 13: Performance overhead comparison of HARE with with UndoLog+Sort and SW RedoLog for checkpointing interval of 0.5 sec.

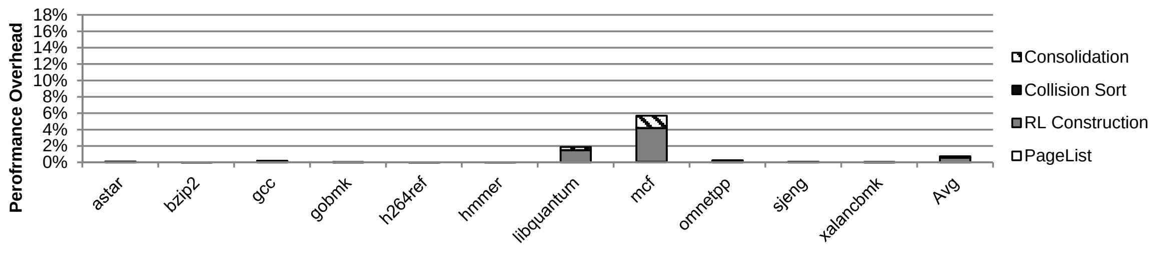
the blocks that are detected not to be checkpointed during a given interval[11], and sorting the resulting undo-log checkpoint. For estimating the cost of sorting the meta-data of the undo-log I profiled in the simulator the latency of sorting arrays of different sizes. The second technique HARE is compared against is a software technique that creates redo-log checkpoints in parallel with the application's execution (named SW RedoLog in Figure 13). This technique approximates the behavior of a shadow software thread which is responsible at the end of every redo-log interval to copy in parallel with the application's execution the pages that were modified in the previous interval. The shadow thread is created by the OS or a checkpoint management library and shares the address-space with the application. To prevent the application from modifying a memory location that has not been checkpointed yet, the shadow-thread write-protects the memory pages to be copied. If

[11]The blocks to be checkpointed are detected using *checkpoint* bit information similar to Euripus

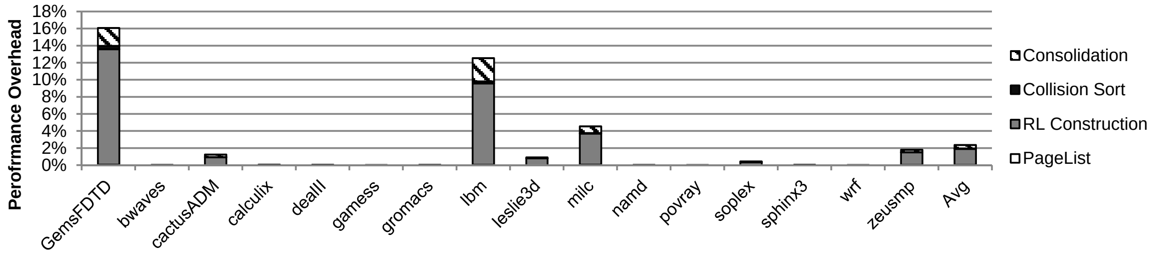
the application tries to modify a write-protected page then a signal is generated, the OS application pauses the application's execution, the shadow thread checkpoints the page, and then the application resumes execution. In the simulator infrastructure this behavior is approximated by running a second program, along side with the one simulated, which behaves similar to the shadow-thread. For the case of the shadow checkpointing thread there are two primary sources of overhead: 1) the time the applications is suspended either because it tried to modify a page that was not checkpointed yet, or the redo-log construction did not finish before the next interval start, and 2) the competition between the application and the shadow-thread for shared on chip resources such as the shared L2 cache, or off-chip bandwidth.

Overall, HARE incurs minimal overheads, with a maximum overhead of 16% across all applications, while the averages are under 6% for SPEC INT, under 3% for SPEC FP, and under 1% for PARSEC (Figure 16). The performance overhead of sorting the undo-log for consolidation is 10% on average across all applications, while it can exceed 50% of individual cases (e.g. GemsFDTD or lbm). The applications with the highest overheads for UndoLog+Sort are the ones which have the biggest checkpoints and sorting them becomes really expensive. The software alternative I investigated has even higher overheads that the two hardware ones, and they can exceed 100% in some cases (mcf or milc). For the high overhead applications, the time the application is suspended because of a page fault, or because the shadow thread did not finish the checkpoint creation in time, can be 70% of the execution overhead.

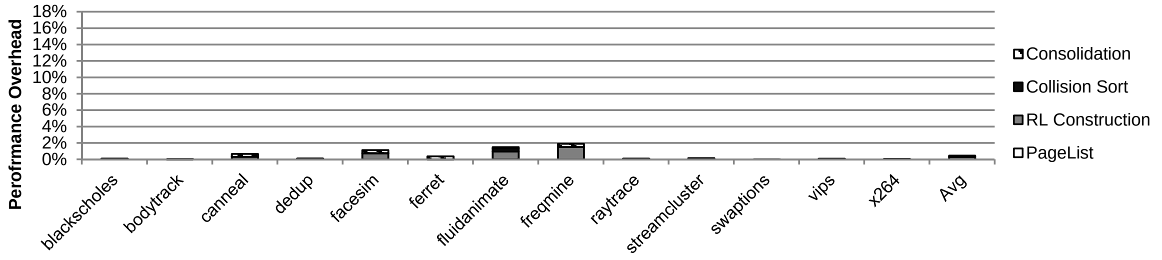
To gain insight into the sources of HARE's performance overhead, Figure 14 breaks this overhead down into its four components (bottom to top): *PageList* is the time needed for HARE's software handlers to scan the page table and find modified pages for each checkpoint; *RL Construction* is the increase in execution time due to using HARE's engine to create a checkpoint (most of the overhead is due to competition for memory bandwidth between the processor and the engine); *Collision Sort* is the overhead of sorting (in software) the collision list after a checkpoint is created; *Consolidation* is the increase in execution time due to using HARE's engine to consolidate checkpoints (again, most of this overhead comes from memory bandwidth contention). The primary source of overhead is the construction of the checkpoint which can result in high overheads for memory intensive applications that modify a lot of memory, e.g. GemsFDTD, lbm and mcf.



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 14: Performance overhead breakdown of HARE for checkpointing interval of 0.5 sec.

The second cause of overhead is checkpoint consolidation. Consolidation causes lower overheads that checkpoint creation (less that 25% of the checkpoint creation cost) becasue it operates only on the meta-data (the checkpointed blocks are not moved). Finally, the cost of sorting the collision list is mininal, $<1\%$. The HARE engine proves to be fast enough so that not a lot of collisions are generated. Unfortunately any slow-down of the HARE engine, e.g. because of not enough memory bandwidth, will result in increase of the number of collisions, because the application is bound to modify memory locations that the HARE engine has not checkpointed yet.

To examine how checkpointing frequency affects performance, I did a checkpointing frequency sensitivity analysis. Figure 15 shows the performance overhead of HARE for the worst performing applications along with the averages for checkpointing intervals of 0.1, 0.5 and 1 sec, and Figure 16 shows the maximum and average overheads of HARE, UndoLog+Sort and SW RedoLog for each

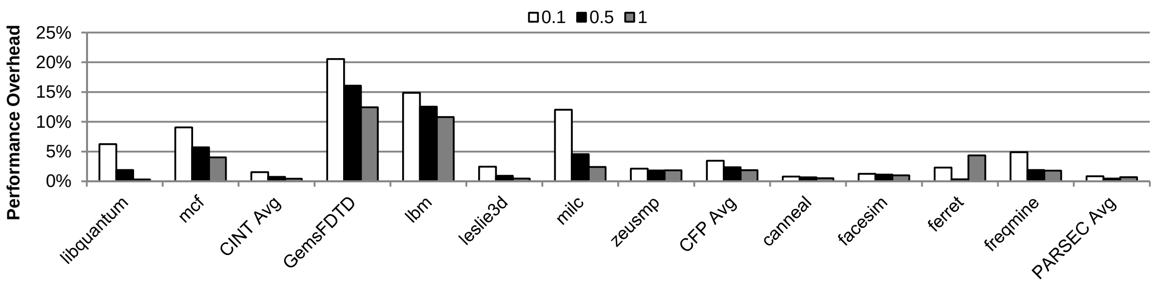


Figure 15: Performance overhead of HARE for checkpointing intervals of 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

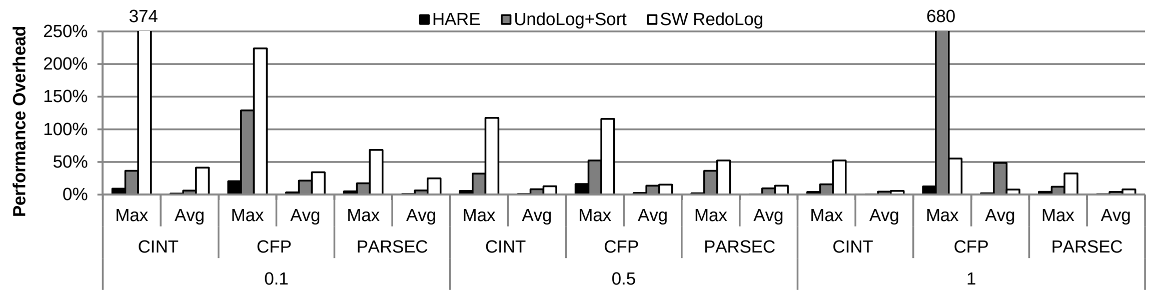


Figure 16: Maximum and average performance overhead of HARE, UndoLog+Sort and SW RedoLog for checkponting intervals of 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

benchmark suite for the same frequencies.

The overheads of HARE do increase when checkpointing is performed more frequently (every 0.1 sec), but they are still tolerable, with <5% additional overhead for the high overhead benchmarks such as GemsFDTD. The overheads of HARE are expected to further increase at higher checkpointing frequencies, e.g. every 0.01 sec, but checkpointing at such frequencies is not expected to be necessary for the case of bidirectional debugging, because the goal of improved interactivity can be met with the range of checkpointing frequencies examined, as will be demonstrated in Section 3.6.7.

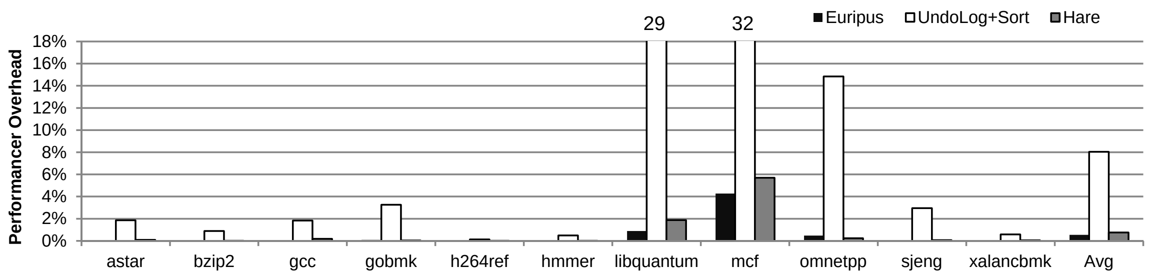
For the other two studied techniques, the performance overheads become prohibitively high when checkpointing every 0.1 sec (Figure 16). SW RedoLog has maximum overheads that would render the application more than two and three times slower compared to normal execution, and as expected the overheads drop at lower checkpointing frequencies. UndoLog+Sort has a similar behavior with higher overheads at increased checkpointing frequencies, but at lower frequencies there are some cases where the performance overhead does not drop at the same rate as does for the other techniques, if not, it increases (e.g 678% for the case of lbm when checkpointing every 1 sec). This behavior can be attributed solely to the increased cost of sorting the undo-logs, whose total size does not decrease as expected, but actually undo-logs become longer. The increased undo-log sizes can be explained by the fact that existing hardware undo-log construction techniques, such as ReVive [68] and SafetyNet [94], can copy the same block multiple times to the undo-log during a given interval, as explained in Section 3.5.1. The number of unnecessary checkpointed blocks increases as the duration of the checkpointing interval does, which renders such techniques unsuitable for low checkpointing frequencies.

Overall, the overheads of HARE are significantly lower than when consolidation is added to previous approaches. HARE's overheads are also low enough to permit debugging of long-running applications with realistic (large) data sets.

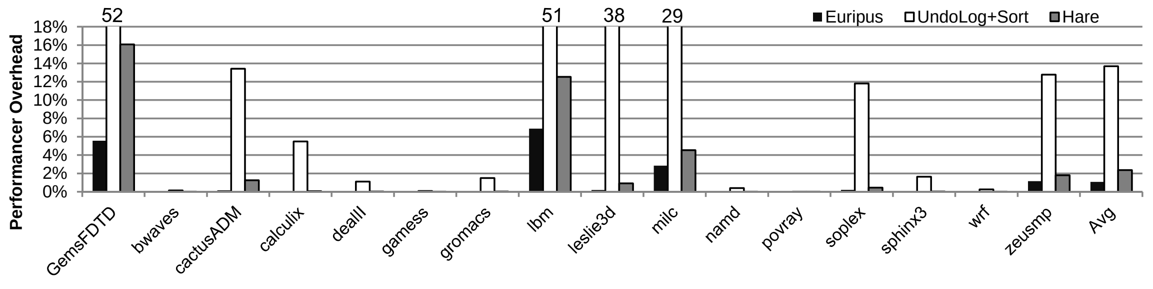
3.6.4 Euripus: Performance Overhead Evaluation

Figure 17 presents the performance overhead of the Euripus technique when only undo-log construction is enabled and the checkpointing interval is 0.5 sec. The engine also consolidates the undo-log checkpoints using the same selection policy as HARE. I am comparing Euripus with HARE and the UndoLog+Sort hardware checkpointing technique described in Section 3.6.3. In this set of experiments I have disabled to redo-log construction fuctionality of Euripus, because it can serve to speed-up debugging and is not essential for restoring a past program memory state, for which only undo-log checkpoints are necessary. As a result, Euripus and HARE provide exactly the same functionality and their overheads are directly comparable.

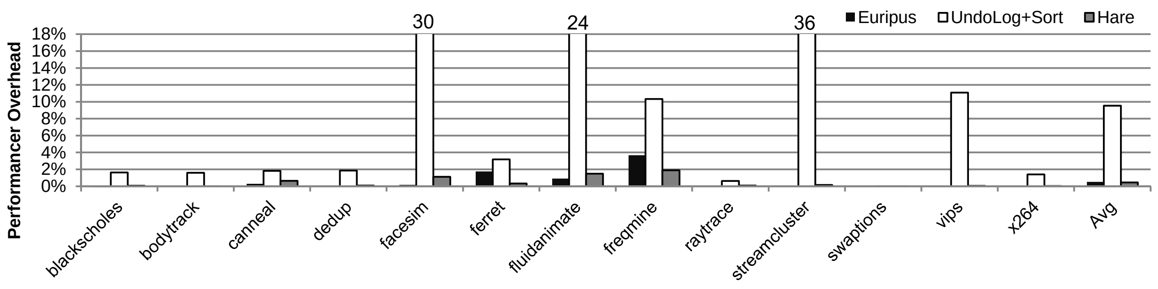
Euripus has low performance overheads, less than $<2\%$ on average across all applications, similar to HARE, and actually lower for the high overhead applications such as GemsFDTD, lbm, milc and mcf. The only two applications that Euripus's overheads are higher than HARE are ferret and freqmine, but only by an additional cost of 2%. There are multiple factors that explain the performance benefit of Euripus compared to HARE: First, the checkpoint creation process of HARE has a burstier behavior than Euripus. HARE walks through the checkpoint meta-data and copies memory immediately, avoiding any possible delays, because of fear of increased number of collisions. As a result, HARE competes aggressively with the application for off-chip bandwidth, resulting in less



(a) SPEC INT



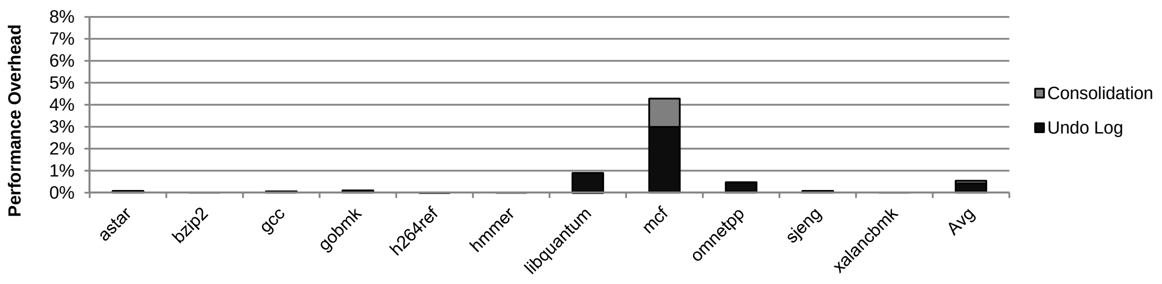
(b) SPEC FP



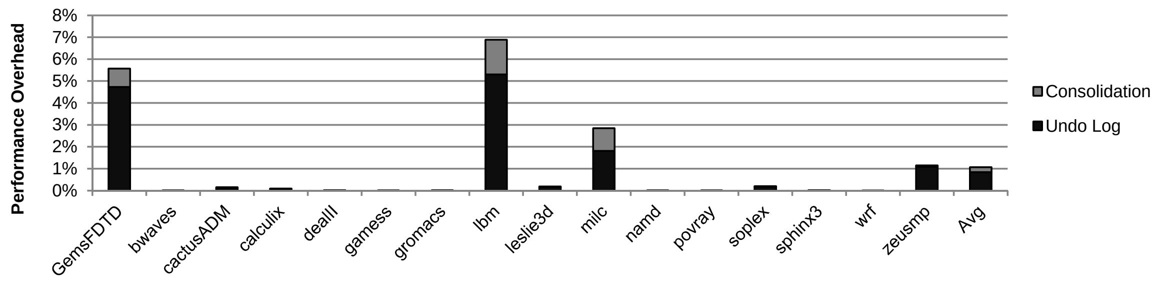
(c) PARSEC

Figure 17: Performance overhead of Euripus, UndoLog+Sort and Hare for a checkpointing interval of 0.5 sec.

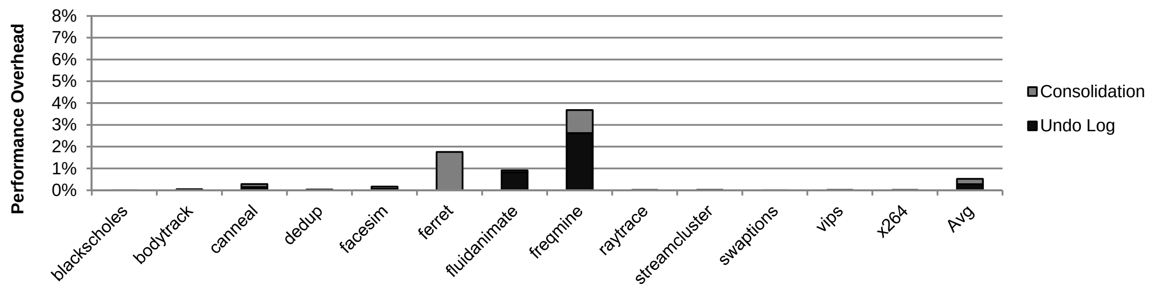
memory bandwidth for the application, which directly affects the memory-intensive ones. Euripus, on the other hand, checkpoints memory blocks the first they get modified, which results in a checkpointing rate that is slower compared to HARE, and matches the rate the application access memory locations for the first time in a checkpointing interval. Second, HARE's consolidation process is more memory intensive that Euripus's. HARE's checkpoint meta-data is 25% of the overall checkpointed memory, which have to be read and written during consolidation. Euripus's trie meta-data is a better suited data-structure for representing a set of modified blocks, compared to a simple list of addresses, requiring less than 14% additional storage. Finally, the data-structure optimizations described in Section 3.5.3, improve the locality in Euripus's meta-data cache, reducing the overall



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 18: Performance overhead break down of Euripus for checkpointing interval of 0.5 sec.

bandwidth consumed for reading and updating the checkpoint meta-data during consolidation.

Figure 18 shows the breakdown of the performance overhead of Euripus for constructing the undo-log (Undo Log) and checkpoint consolidation (Consolidation). Similar to HARE, Euripus's consolidation overhead is minimal, because only the meta-data are being updated and the checkpointed data are not moved in memory. The higher overhead of ferret and freqmine compared to when using HARE can be attributed to the high miss-rate of Euripus's L5MD cache both during checkpoint construction and consolidation, which results in more memory bandwidth to be used by the engine.

Figure 19 shows the worst performing benchmarks and the averages for checkpointing intervals

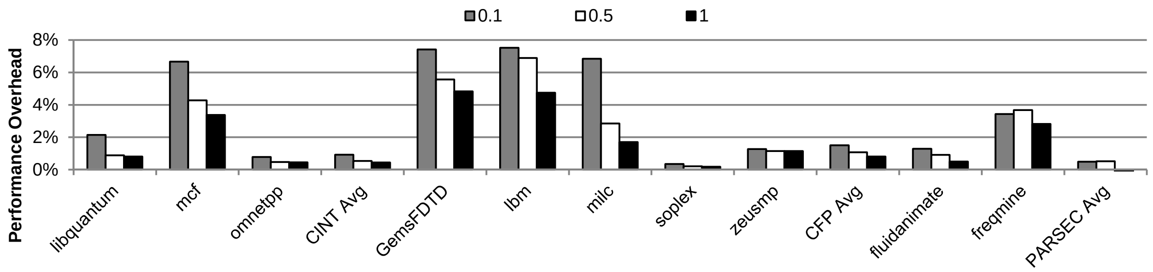


Figure 19: Performance overhead of Euripus for checkpointing intervals 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

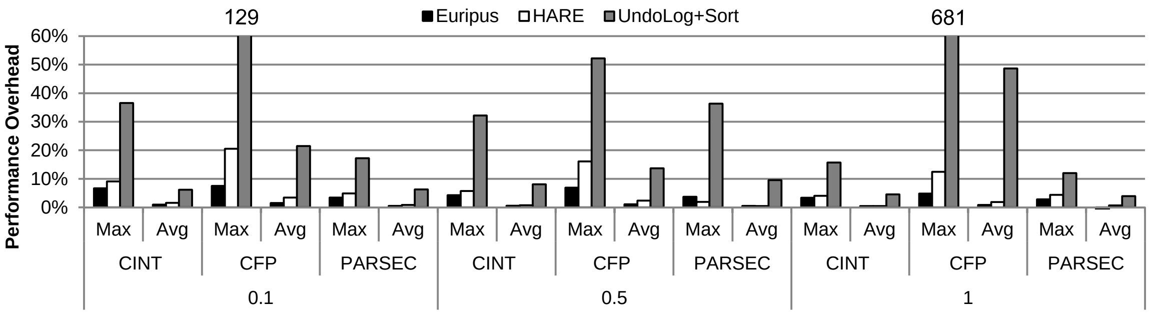
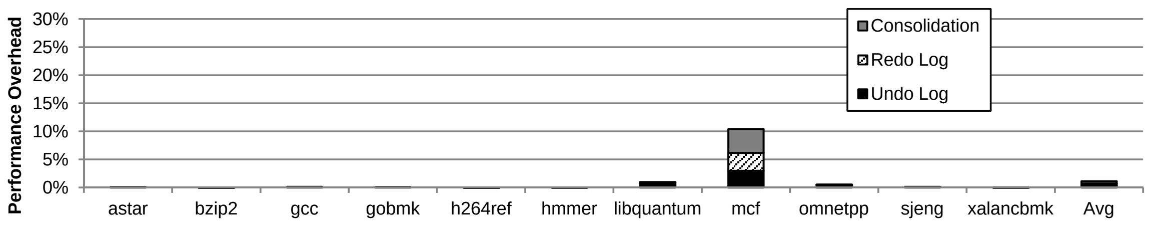


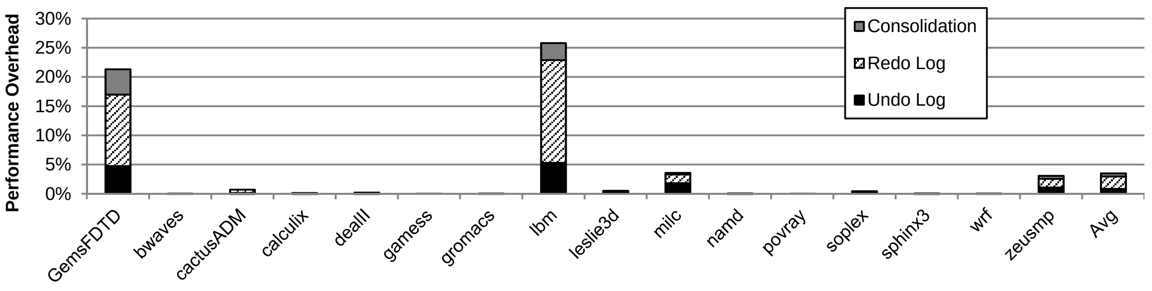
Figure 20: Performance overhead of Euripus, HARE and UndoLog+Sort for checkpointing intervals of 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

of 0.1, 0.5 and 1 sec. Similar to HARE, Euripus suffers low additional performance cost when creating checkpoints frequently, $<8\%$ for the worst performing benchmarks (GemsFDTD, lbm mcf, milc). Figure 20 compares Euripus with the two other techniques. Euripus has lower performance overhead than HARE even at a checkpointing interval of 0.1 sec, while the comparison with UndoLog+Sort demonstrates the inefficiency of a solution that would rely on software-based sorting for consolidating undo-log checkpoints. Euripus exploits its searchable checkpoint meta-data data-structure, and checkpoints a given memory location only once during a checkpoint interval, eliminating the additional memory bandwidth that prior techniques consume.

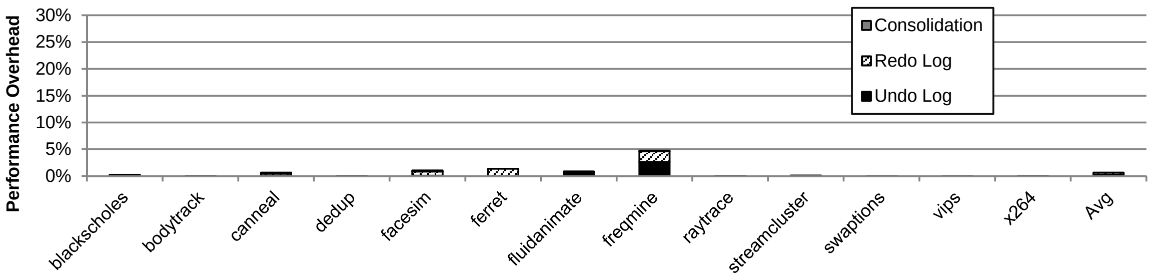
Overall, Euripus incurs minimal performance overheads, lower than HARE, but for an increased hardware cost. Euripus efficiently constructs undo-log checkpoints and enables their efficient consolidation, unlike prior hardware checkpointing techniques which cannot provide such functionality, and would incur high overheads if applied to the reverse-execution problem.



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 21: Performance overhead breakdown of Euripus when both undo and redo-log checkpoints are create for a checkpointing interval of 0.5 sec.

3.6.5 Concurrent Undo and Redo Log Construction

Figure 21 presents the performance overhead breakdown of Euripus when both undo and redo-log checkpoints are created for a checkpointing interval of 0.5 sec. The performance overhead is broken down to the cost of creating the undo-log (Undo Log), the redo-log (Redo Log) and consolidating the checkpoints (Consolidation). Euripus has average performance overhead $< 3\%$ across all benchmarks, with the highest overheads being 25% for lbm, ~16% for GemsFDTD and ~10% for mcf. In all three cases the primary cause of the overhead is the redo-log creation, whose cost can be explained for the following 3 reasons: 1) all three benchmarks are memory intensive and they create big checkpoints, 2) the Euripus engine competes with the application for off-chip bandwidth, and 3) only a limited number of redo-log blocks is synergistically checkpointed by the undo-log creation

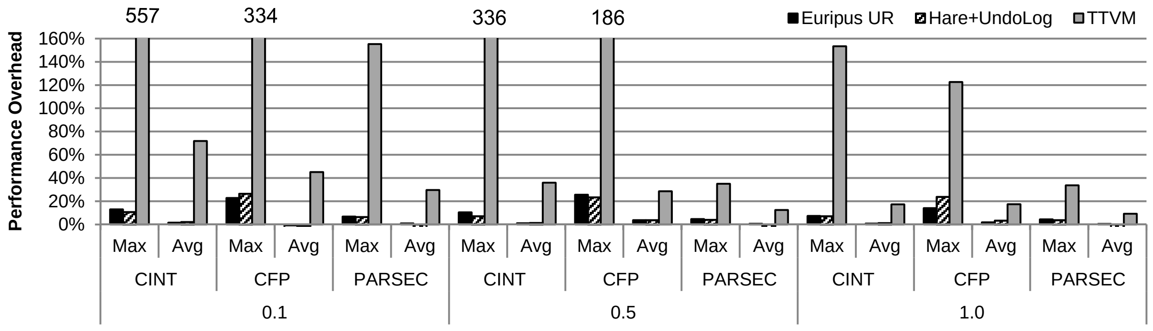


Figure 22: Maximum and average performance overheads of EuripusUR, Hare+UndoLog and TTVM for checkpointing intervals of 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

process ($< 10\%$). Finally, the cost of checkpoint consolidation is increased compared to when only undo-log checkpoints are created, but still remains low, $< 1\%$ on averate across all benchmarks, while the highest values were observed for GemsFDTD (8%) and mcf ($\sim$4%).

I am also comparing Euripus, when performing concurrent undo and redo-log construction (EuripusUR), with a hardware and a software scheme. The hardware scheme, labeled *HARE+UndoLog*, creates undo-log checkpoints using a hardware technique similar to ReVive [68] and redo-log checkpoints using HARE. HARE consolidates the redo-log checkpoints created, but for the case of the undo-logs, they are not consolidated, meaning HARE+UndoLog provides less functionality and theoretically has a small performance advantage over Euripus, since it does not provide consolidatable both undo and redo-log checkpoints. The software technique, named *TTVM*, is similar to the virtual-machine based checkpointing solution proposed by King *et al.* [80]. In TTVM, a checkpointing thread executes in parallel with the application and creates both undo and redo-log checkpoints. TTVM uses memory protection and copy-on-write to identify the pages to be copied and checkpoints them. When simulating multi-threaded applications, the checkpointing thread has higher priority and de-schedules the application's threads if no free core is available.

Figure 22 shows the maximum and average performance overhead for the simulated benchmark suites for checkpointing intervals of 0.1, 0.5 and 1 second. Both hardware techniques outperform TTVM, which suffers high performance overheads especially at high checkpointing frequencies, e.g. every 0.1sec. The primary cause of performance overhead is the pollution of the shared caches with checkpointed data, and the competition between the application and the checkpointing thread

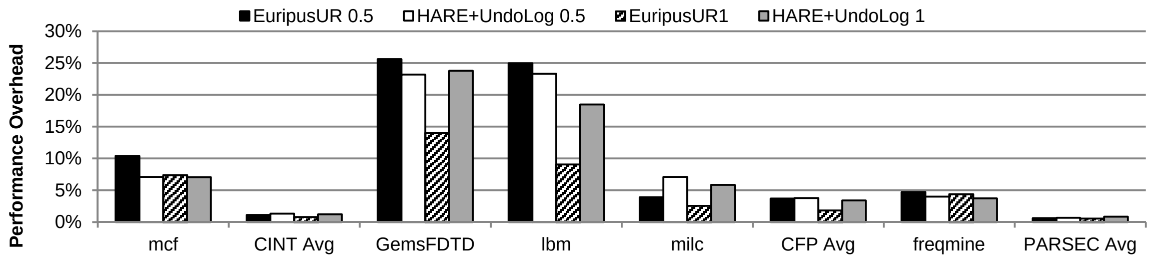


Figure 23: Performance overhead of worst performing applications and averages for Euripus (EuripusUR) and Hare+UndoLog for checkpointing intervals of 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

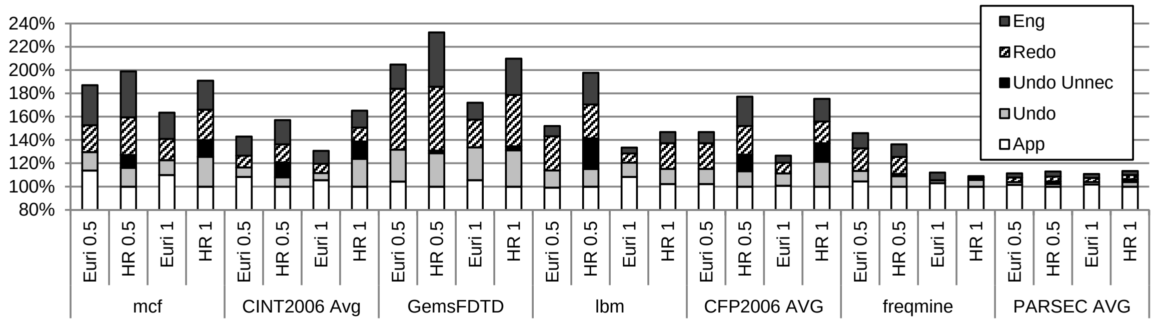


Figure 24: Break down of the memory accesses of Euripus (Euri) and HARE+UndoLog(HR) for the highest overhead applications and the averages for checkpointing intervals of 0.5 and 1 sec.

for cache space. Another source of overhead for TTVM is the time the application is suspended for servicing the page faults which comprises more than 20% on average (maximum 60%) of the overhead. Euripus and HARE+UndoLog have similar average overheads, but Euripus has lower performance overheads at lower checkpointing frequencies (every 1 sec), while HARE+UndoLog's overheads do not reduce at the same rate (Figure 23). The higher overhead of HARE+UndoLog compared to Euripus can be attributed to the following reasons: 1) the lack of synergistic copying between undo and redo-logs, which results in the same data being copied twice, 2) higher consolidation cost, especially for applications which create big checkpoints (e.g. lbm, GemsFDTD), and 3) undo-log memory blocks checkpointed multiple times withing the same checkpointing interval.

To gain better insight into the performance benefits of Euripus compared to HARE+UndoLog, Figure 24 presents the breakdown of the average memory accesses per checkpoint interval which consist of: 1) the application memory accesses (App), 2) the necessary undo-log writes (Undo), 3) the unnecessary checkpointed undo-log blocks (Undo Unnec), 4) the redo-log read and writes (Redo) and 5) rest of the memory access generated by the checkpointing engines (Eng) (e.g. caches misses from Euripus's L5MD cache, or checkpoint meta-data reads/writes during checkpoint consolidation). The accesses are normalized to the application accesses of HARE+UndoLog for each checkpointing frequency. The first observation that can be made is that Euripus's engine generates less memory access than HARE+UndoLog. Euripus's L5MD cache proves sufficient for caching the trie meta-data structures during checkpoint creation. The merge optimizations also assist, resulting in only 60-70% of trie block pointers being updated on average and reducing the L5MD cache pressure and associated memory accesses. On the other hand, HARE during consolidation has to read the meta-data of both checkpoints, which are lists of addresses/pointers the size of which is 25% of the checkpointed data, and write the resulting consolidated list, resulting in more memory accesses than Euripus. The second improvement is that Euripus inserts a given address in the undo-log only once, while the UndoLog may generate duplicate checkpoint entries. This phenomenon is especially pronounced in longer checkpointing intervals, e.g. 1sec, where the probability of a block being replaced from the L2 cache, and the associated *checkpoint* bit information to be lost, increases. Finally, synergistic copying can reduce, if not almost eliminate (e.g. for lbm), the number of blocks copied for constructing the redo-log at low checkpointing frequencies (every 1 sec). In such cases redo-log checkpoints are created practically for free and Euripus just needs to check the meta-data and verify that the block has been copied.

3.6.6 Memory Requirements

To estimate long-term memory requirements, I used PIN [45] to model[12] HARE's and Euripus's functionality. I used the SPEC 2006 [96] benchmarks, with ref input sets, and all PARSEC 2.1 [8] benchmarks, with native inputs and 8 threads. Each application is simulated to completion, using regular checkpoint consolidation in HARE and Euripus when both undo and redo-log checkpoint creation are enabled (*EuripusUR*). The two techniques are compared against a hardware technique that creates undo-log checkpoints (*UndoLog*) similar to ReVive [68] and SafetyNet [94] and a software technique (*Page*) that keeps track of the modified memory locations at a page granularity. Both alternative techniques do not consolidate checkpoints, since the purpose is to demonstrate the

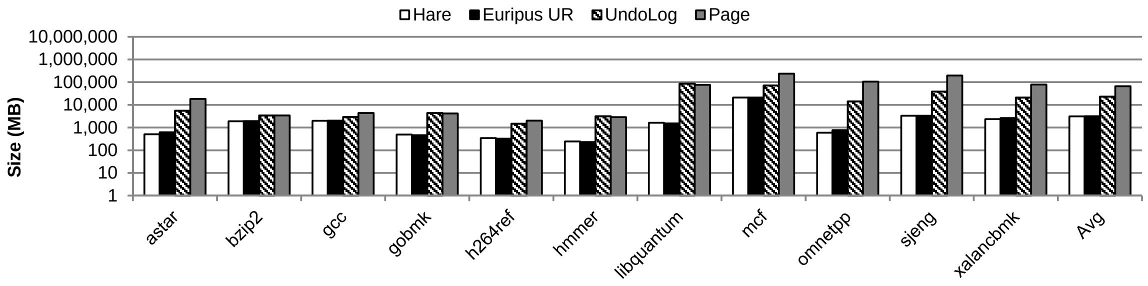
[12]Detailed simulation is infeasible for this, as it goes through only a few checkpoint periods per hour of simulation time.

benefit of checkpoint consolidation. Note that the UndoLog shown here is not the same as in my performance overhead experiments – without consolidation, overheads would be significantly lower, and with consolidation their space overheads would be similar to HARE's and Euripus's. These different kinds of experiments in different parts of the evaluation are used to show that HARE and Euripus achieve both efficient space use and low forward-execution overheads, while prior undo-log hardware schemes can only achieve one or the other – either 1) low performance but high space overheads if consolidation is not used, or 2) low space but high performance overheads if consolidation is used.

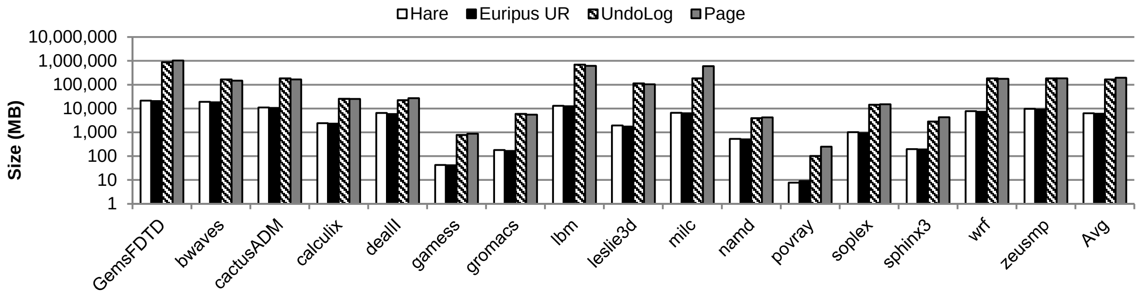
Figure 25 shows the memory requirements of HARE, EuripusUR, UndoLog and Page at the end of the execution of each benchmark when checkpoints are created every 0.5 sec. The resuls include both the total memory checkpointed, as well as the necessary meta-data stored along with the checkpoint. We observe that total space used for checkpoints in HARE and EuripusUR is an order of magnitude (36 times on average) lower than in prior hardware schemes (note the logarithmic scale in these charts). In the worst case, HARE and EuripusUR require up to 20 gigabytes of memory (GemsFDTD, bwaves), while without consolidation that would require more than 1 terabyte of storage. In case that such memory overhead cannot be supported, more aggressive checkpoint consolidation can be performed, limiting the number of older checkpoints maintained, but still maintaining a few to enable reverse-execution further in the past if necessary. Further, note that lower memory requirements of UndoLog compared to Page in applications like mcf, omnetpp and xalancbmk[13]. Such applications have non-continuous memory access patterns, and as a result they tend to modify only a fraction of a memory page they touch within a checkpointing interval. As a result, a finer memory tracking granularity, block for the case of UndoLog, can capture the set of modified memory locations more accurately and avoid copying blocks which have not been modified, as page-based memory tracking schemes do.

The similar memory requirements of HARE and EuripusUR seem counter-intuitive at first, because EuripusUR creates both undo and redo-log checkpoints while HARE creates only redo-log checkpoints. As described in Section 3.4.2, the checkpointed data are shared between consecutive

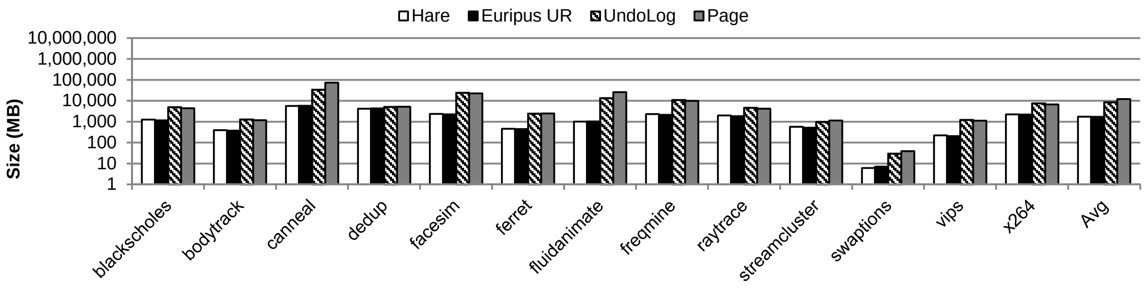
[13]There are some applications where the storage requirements of UndoLog are higher than the Page's, this can be attributed to the higher cost of storing the meta-data, which is a list of address for the UndoLog.



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 25: Memory requirements of HARE, EuripusUR, UndoLog and Page for checkpointing interval of 0.5 sec.

redo and undo-log intervals. As a result, common data are checkpointed only once and the metadata just points to the correct locations. Consecutive checkpoint intervals, though, are not expected to always completely overlap as my results seem to indicate. The similar memory requirements of HARE and EuripusUR can be explained by two factors: First, checkpoint consolidation creates denser and denser checkpoints overtime, with old consolidated checkpoints practically representing the application's whole address-space. Such old consecutive checkpoints almost completely overlap, resulting in no extra blocks to be maintained. Also the younger checkpoints, which are expected to have lower overlap, are created frequently, and their size is relatively small compared to the older checkpoints, while they are consolidated together soon after their creation, eliminating

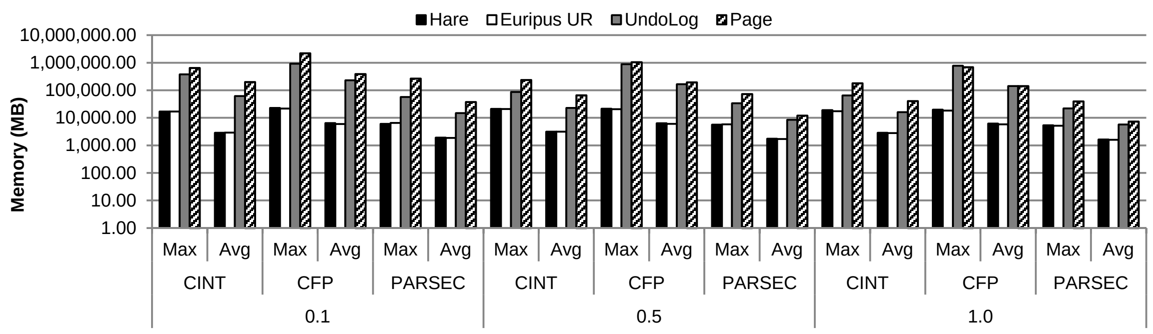
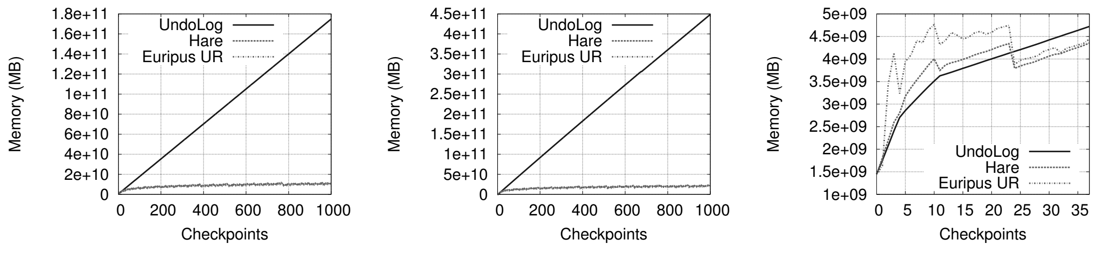


Figure 26: Maximum and average memory requirements of HARE, EuripusUR, UndoLog and Page for checkpointing intervals of 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

any additional blocks. Second, the trie meta-data data-structure used by Euripus is more memory efficient than the list[14] used by HARE. HARE's meta-data has a fixed cost of 25% relative to the data checkpointed, assuming the system supports 64-bit addresses. Euripus's trie requires 18% on average extra storage relative to the checkpointed data for the SPEC INT applications and 14% for the SPEC FP and PARSEC applications. Also the advantage of the trie is that it has lower additional cost when representing denser checkpoints, e.g. the consolidated ones. Note that the memory requirement of Euripus when only undo-log checkpoints are created is going to be similar to HARE's because they will not include the additional redo-log checkpointed blocks.

To reduce space requirements, other schemes could try reducing the checkpointing frequency. Figure 26 shows the memory requirements of the different schemes for checkpointing intervals of 0.1, 0.5 and 1 sec. As shown in the chart, reducing the checkpointing frequency from once per 0.1sec to once per 1 second does significantly reduce total space used in UndoLog schemes (FDR or ReVive) and Page. Even then, however, the space use in these schemes can reach hundreds of gigabytes to capture what amounts to tens of minutes of execution. In contrast, HARE's and Euripus's consolidation effectively changes the checkpoint frequency for older checkpoints, allowing them to use a limited amount of space regardless of checkpointing frequency. In most applications HARE and Euripus when checkpointing every 0.1 or 1 sec consume similar amounts of space. More importantly, checkpointing every 0.1 sec with consolidation enabled often uses an order of magnitude

[14]With every element in the list storing two addresses the virtual address of the block checkpointed and they physical address where it is checkpointed.



(a) lbm (b) GemsFDTD (c) dedup

Figure 27: Memory requirements of UndoLog, Hare and EuripusUR after a number of checkpoints has been created for a checkpointing interval of 0.5 sec.

less space than when consolidation-less UndoLog or Page use when checkpointing every 1 sec.

It should be noted that HARE's and Euripus's advantage over consolidation-less schemes grows as execution proceeds: Figure 27 shows the total memory requirements after a number of checkpoints is constructed (checkpointing interval of 0.5sec) for HARE, Euripus with both undo and redo-log creation enabled (*EuripusUR*), and for UndoLog for several applications. The trend is towards linear growth in UndoLog and logarithmic growth in HARE and Euripus for long-running applications like lbm, GemsFDTD. Dedup is an example of an application that creates only a few checkpoints so the benefit of consolidation is limited. In dedup we can also observe the memory requirements of EuripusUR compared to HARE. In Euripus, young checkpoints share few blocks, which explains the higher memory requirements of Euripus compared to HARE during the first 20 checkpoint intervals. But once a sufficient number of consolidations happen, e.g. after interval 25, the extra blocks are eliminated and HARE and Euripus have similar memory overheads.

Overall, HARE and Euripus have one to two orders of magnitude less memory requirements than consolidation-less techniques, and have the advantage of providing consolidatable checkpoints at low performance overheads, similar to prior techniques. Euripus additionally has managed to created both types of checkpoints, undo and redo-log at the same time, delivering the same memory requirements for limited additional performance cost.

3.6.7 Reverse Execution Latency

The previous sections demonstrated the performance and memory requirements of the HARE and Euripus checkpointing techniques. Both techniques incur minimal performance overheads and have

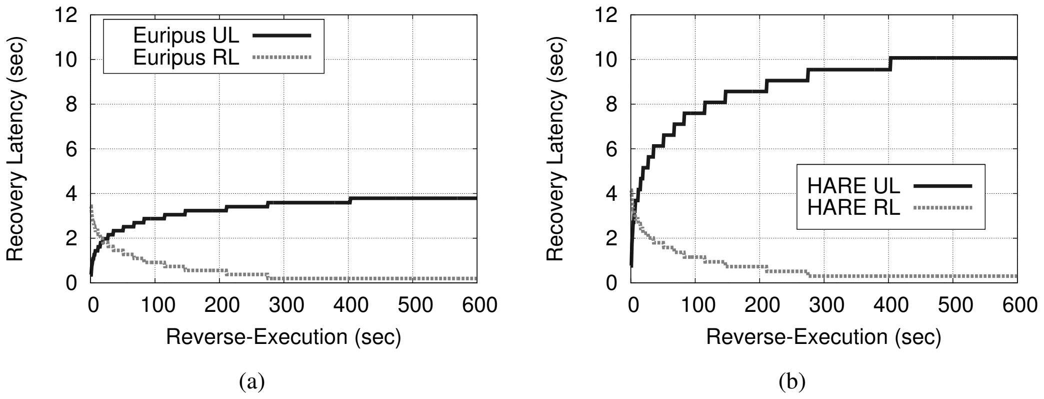


Figure 28: Program memory recovery latency of GemsFDTD when reverse-executing a number of seconds of original program execution using undo-log only (Euripus UL, HARE UL) and redo-log (Euripus UR, HARE UL) checkpoints.

managed to significantly reduce the total memory requirements of reverse-execution through consolidation. The two techniques have selected to create different types of checkpoints, HARE redo-log and Euripus both undo and redo-log, a design choice that directly affects the hardware cost and complexity of their hardware accelerators, with HARE having a lower hardware cost. This Section evaluates the reverse-execution latency that the two techniques can achieve, and verify if they can meet the goal of supporting the programmer's intuitive notion that reverse and forward execution should have similar "speed".

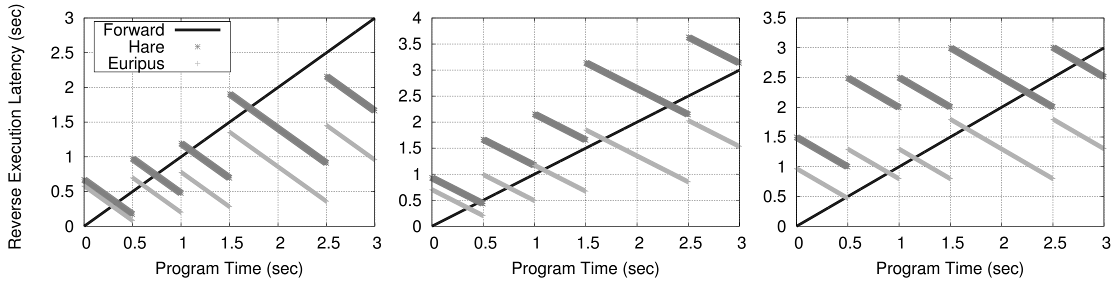
To evaluate the latency for each technique, additional experiments are performed, where is estimated the latency of varying amounts of reverse-execution (from back-stepping one instruction to undoing the entire execution), all starting from the same state (the very end of the application's execution). It would take years to get these results through cycle-accurate simulation, so for this evaluation the of reverse-execution latency is estimated as follows:

$$t_{\text{reverse-execution latency}} = t_{\text{restore application}} + t_{\text{replay application}}$$

I am assuming that the time to replay the application until the point of interest in the program's execution is the same as the original forward execution of the program. For the time to restore the application to a specific point in time I estimate the time to apply a checkpoint and restore the program to a prior point in time. The best type of checkpoint to use to recover the application to a specific point in time depends on the number of checkpoints necessary to recover the application's memory state at that specific point in time. For the case where undo-log checkpoints are being used the debugger will have to apply in reverse-order all undo-log checkpoints until we reach the point of interest, whereas when redo-log checkpoints are being created the debugger has to start from a full-application checkpoint and then apply all incremental redo-log checkpoints in forward creation order. Figure 28 shows the time in seconds the debugger will spend restoring the program state of GemsFDTD when reverse-executing a number of seconds of original program execution, when using undo-log or redo-log checkpoints for Euripus and HARE (HARE constructs undo-logs by converting the redo-logs). The further the programmer wants to back-track in the program's execution the higher the recovery cost for undo-log checkpoints, because more checkpoints have to be applied to recover the program state. Redo-log checkpoints have almost the inverse cost than undo-log checkpoints with back-tracking only a limited number of instructions having the highest cost. The interactivity sensitive area of bidirectional debugging is the one close to the program's end of execution, where the programmer is expected to perform most reverse-execution operations in order to identify the cause of a bug. Using undo-log checkpoints the state can be restored in less than a second, while when using redo-log checkpoints it will require several seconds, rendering bidirectional debugging less interactive. Executing further is the past, the program state recovery cost is going to have diminishing effects because the reverse-execution latency is going to be dominated by the program's replay time[15]. A specific selection algorithm cannot be proposed, because that would require an implementation of a reverse-debugger and profiling information, but a simple heuristic can be based on selecting the type of checkpoint that will need to restore the less amount of memory.

Euripus has the ability to create both types of checkpoints so the debugger could select the type of checkpoint with the lowest estimated recovery latency. HARE creates only redo-log checkpoints and has to convert them to undo-log checkpoints when the application back-tracks only a limited number of instructions. HARE performs the conversion by doing a linear search between checkpoints, similar to a checkpoint consolidation, and finds the oldest value of a block. To estimate the latency to recover application's memory state, shown in Figure 28, I profiled in the simulator the

[15]Checkpoints are distributed exponentially over time in the past, so the further the programmer back-tracks the more application instructions the debugger is going to replay.



(a) lbm (b) GemsFDTD (c) mcf

Figure 29: Comparison of reverse-execution latency with forward execution time for HARE and Euripus for different applications.

time the hardware engine of HARE and Euripus require to restore a checkpoint, for different sizes of checkpoints, as well as the time necessary to perform a linear search between two checkpoints. Using PIN [45], I estimated the distribution of checkpoints in time at the end of the programs's execution, as well as their sizes and the number of searches that will have to be performed in order to convert HARE's redo-log checkpoints. For the case of HARE, I found in my experiments that converting the redo-logs to undo-logs and applying the undo-logs has lower recovery latency when reverse-executing only a few instructions than using redo-logs, because the second approach will have to restore a lot more memory, which has higher cost than converting the checkpoints. Using the profiling information I estimated the reverse-execution latency that a user is going to experience when back-tracking a number of instructions. Figure 29 shows the reverse-execution latency of HARE and Euripus, that the user is going to experience when back-tracking up to 3 seconds in the application's past from the point of a program crash, for 3 representative applications.

Reverse-execution's latency appears to have a "choppy" form. This effect is caused by how the checkpoint-and-replay implementation of reverse-execution works: it restores a checkpoint that precedes a target position, then re-executes until that target position is reached. For target positions within the same checkpoint interval (the same checkpoint is restored), replay time *grows* as we reverse-execute *fewer* instructions – reversing to the beginning of the checkpoint interval results in negligible replay time, while reversing to the end of the checkpoint interval requires replay of the entire checkpoint interval. As a result, the reverse-execution latency shown in Figure 29 "jumps" whenever the increase in the number of instructions results in changing the checkpoint that needs

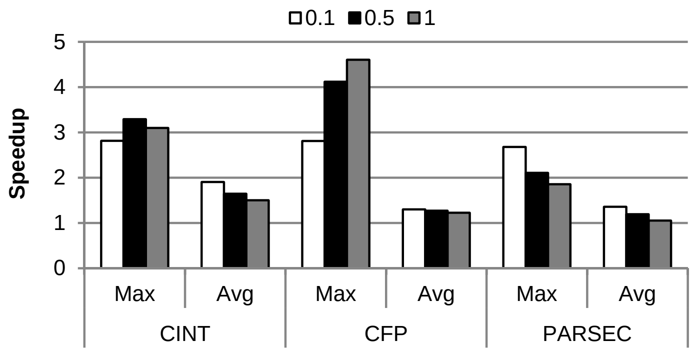


Figure 30: Maximum and average memory recovery latency speedup of Euripus over HARE for checkpointing intervals of 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

to be restored, but then smoothly improves as fewer instructions in that checkpoint interval are replayed.

The overall trend shown in Figure 29 for HARE and Euripus matches what the user (of the debugger) expects from long-term reverse-execution: its "speed" is similar to forward execution. For short-term reverse-execution, the intuitive expectation is for it to be "instantaneous", or have the same latency as forward execution. The largest difference in "speed" between forward and reverse-execution occurs when executing (or reversing) only one instruction (leftmost point in each chart): a forward single-step of one instruction takes only nanoseconds, whereas the worst latency we observe for reverse single-step is about 1 second for Euripus. Although not truly below the human perception threshold, this is still a highly interactive response that is unlikely to frustrate programmer, and it can be improved by checkpointing more often (at the cost of some additional performance loss in forward execution).

Even through HARE will appear interactive to the user, there are cases like mcf (Figure 29(c)) where the cost of converting the redo-log to undo-logs is high and the user is going to notice the increased reverse-execution latency. Euripus, on the other hand, is not suffering from the additional checkpoint conversion cost. Euripus creates both types of checkpoints and for the purpose of back-tracking only a few instructions in the past the debugger can use the undo-logs, which are small and can restore the application really quickly to a past state, whereas if the user wants to back track further in the past the redo-logs can be used. To demonstrate the advantage of Euripus, Figure 30

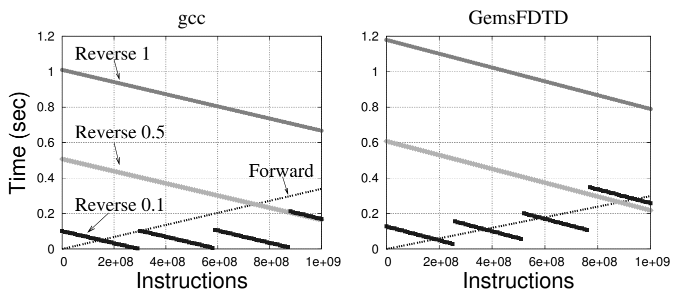
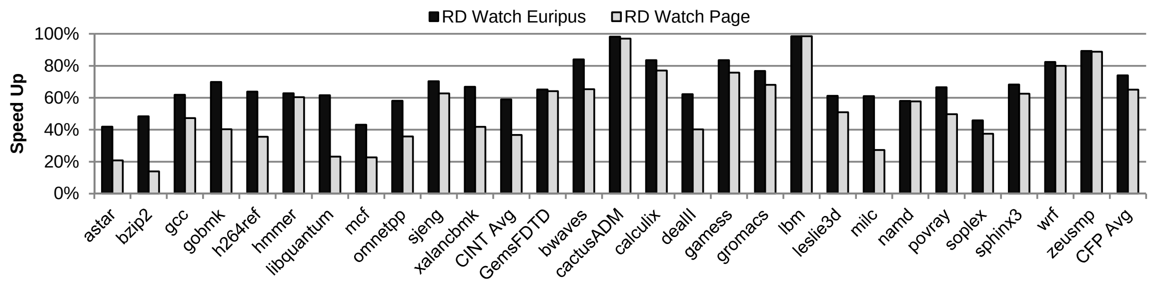


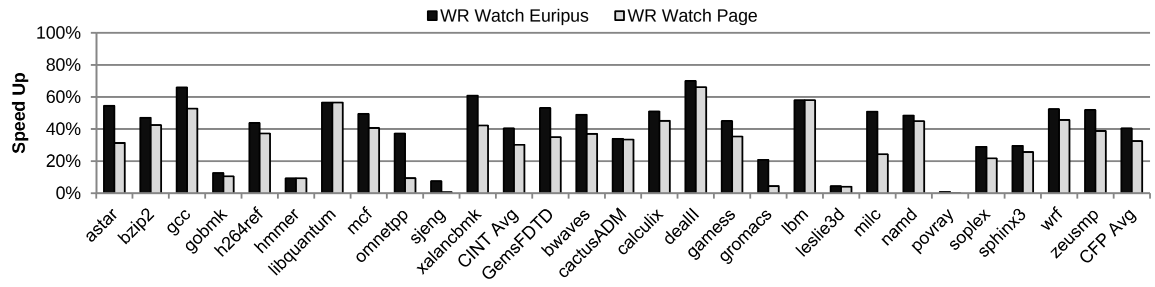
Figure 31: Forward and reverse-execution latency of Euripus for checkpointing intervals of 0.1, 0.5 and 1 seconds.

presents the average and maximum memory recovery speed, for all the benchmark suites for checkpointing intervals of 0.1, 0.5 and 1 sec, of Euripus over HARE that the user is going to experience when back-tracking within 3 seconds from the end of the application's execution. The 3 second window can be considered as representative of interactive critical operations that the user will try to perform. Euripus has up to two times less memory recovery latency on average compared to HARE, with the maximums exceeding three and four times. The overhead of checkpoint conversion of HARE depends on the overlap between checkpoints and the resulting number of checkpoints that have to be searched at the end of the execution. On average, there is lower overlap between consecutive checkpoints in higher checkpointing frequencies, and more checkpoints have to be search. The maximum values shown in Figure 30 do not follow necessarily this trend, e.g. for SPEC FP, because for specific applications when checkpointing at lower frequencies the cost of conversion can be higher, becasue the checkpoints to be searched are bigger. Note that the memory recovery cost of each mechanism is going to have diminishing effects on the reverse-execution latency when back-tracking further in the past, because it is going to be dominated by the program's replay time.

A solution to HARE's checkpoint conversion cost would be to checkpoint less frequently, but such option would affect directly the reverse-execution latency. Figure 31 presents the reverse-execution latency of Euripus for checkpointing intervals of 0.1, 0.5 and 1 second, when reverse-executing only a limited number of instructions. Decreasing the checkpointing frequency increases the number of instructions the debugger has to re-execute at reverse-execution time, resulting in higher latencies. Euripus does not suffer from the checkpoint conversion cost of HARE, and as it was demonstrated in the performance evaluation of Euripus, Euripus can deliver checkpoints at the same checkpointing frequencies.



(a) Read Watchpoint



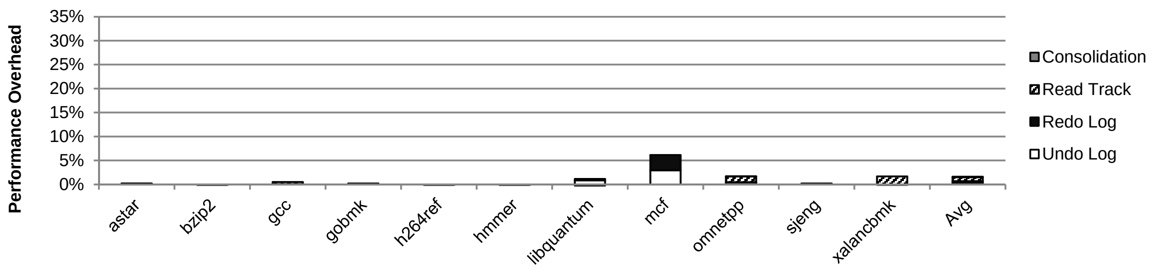
(b) Write Watchpoint

Figure 32: Reverse-execution latency speedup when Euripus's meta-data are used to accelerate "reverse-continue" for read or write watchpoints, and comparison with a technique that keeps track of the same information at the page granularity, for the SPEC INT (CINT) and SPEC FP (CFP) applications.

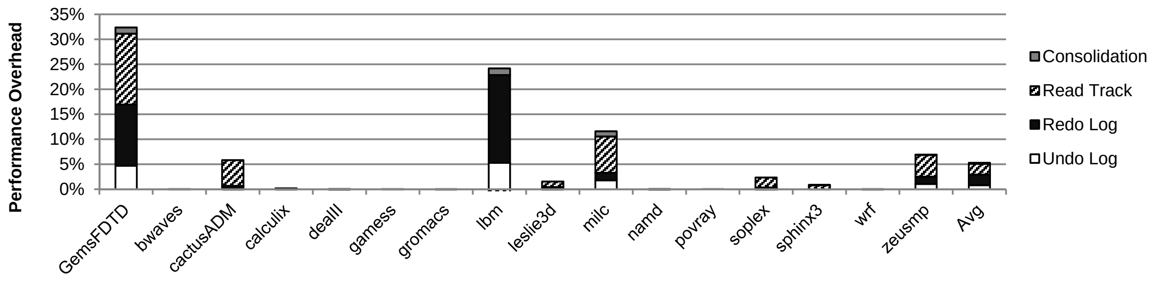
Overall, HARE and Euripus have achieved one of the primary goals, which was to make bidirectional debugging interactive to the user. Euripus has lower reverse-execution latencies than HARE because it can create both types of checkpoints, and the debugger can use undo-log checkpoints when the user wants to back-track only a few instructions, which needs to appear instantaneous to the user.

3.6.8 Enhanced Reverse Execution Functionality

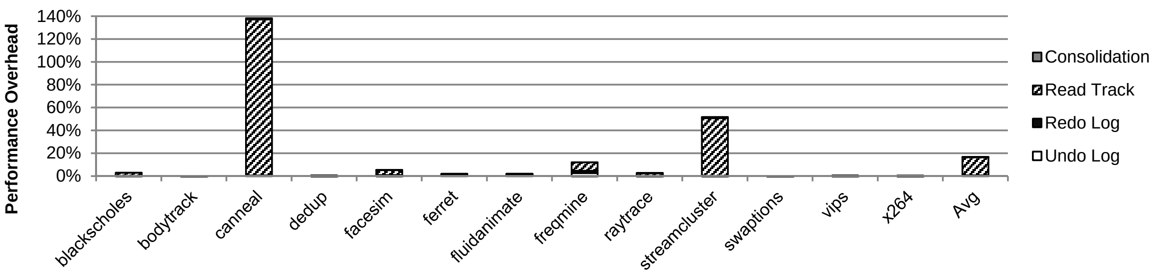
As described in Section 3.4.4, the reverse-execution latency of operations like "reverse-continue" can be reduced by exploiting the checkpoint meta-data and replaying only the intervals where the addresses monitored by watchpoints are actually read or written, thus finding a lot faster the last time a given address was accessed. To estimate the effect of such a technique, I profiled the SPEC applications (Figure 32), using PIN [45], and every one thousand load operations one was randomly selected. For this load operation the latency of a "reverse-continue" operation was estimated, when searching for the last time this address was read (simulating a read watchpoint) and written (for the



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 33: Performance overhead breakdown of Euripus with enabled read tracking for checkpointing interval of 0.5 sec.

case of a write watchpoint).

Figure 32 presents the speed-up that this technique can achieve when checkpointing every 0.1 seconds, over the typical approach that replays all intervals. Euripus keeps track of the read and write information at the granularity of blocks, since this is the default memory tracking granularity. Euripus is also compared against a technique that uses the same approach with the only difference that keeps track of read or writes at the granularity of pages, thus the probability of a false positive is higher. Overall, exploiting the Euripus meta-data can speed-up "reverse-continue" more than 60% percent on average across all applications for the case of read watchpoints and 40% for write watchpoints. Using fine memory tracking granularity (block) can have 20% - 40% higher speed-ups

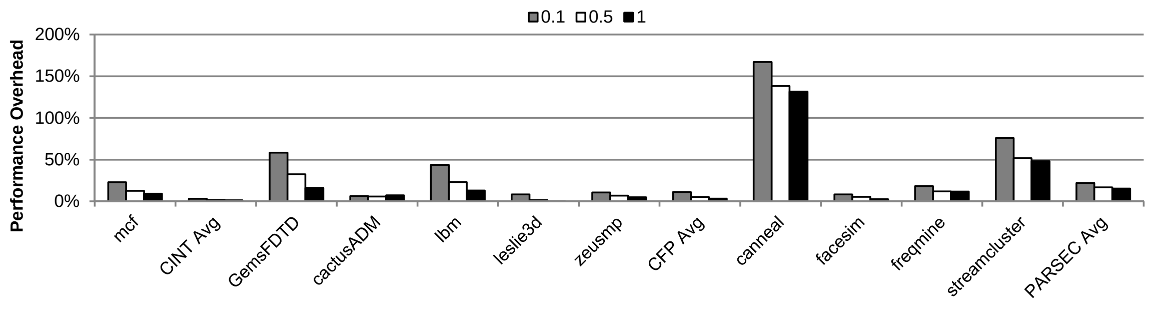


Figure 34: Performance overhead of Euripus with read tracking enabled for checkpointing intervals of 0.1, 0.5 and 1 sec, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

than coarse (page), especially for pointer intensive applications or applications with non-regular memory access patterns, characteristics which describe the majority of the SPEC INT applications. On the other hand, floating point applications experience little benefit from the finer granularity, primarily because they continuously read and write to the same data, and a write or a read operation to a page is frequently followed by another read operation within the same checkpointing interval.

Figure 33 presents the breakdown of the performance overhead when read-tracking is enabled for Euripus for a checkpointing interval of 0.5 sec. On average, the performance overheads of Euripus still remain less than 20% across all benchrmarks. The additional functionality increases the overheads for Euripus because now apart from the blocks that are written for the first time, the address of the read blocks are also sent to the engine. As a result the occupancy of the engine increases, causing the execution to stall when the engine if full both for the read and write instructions. This behavior hurts the performance of applications which already experience high overheads, such as GemsFDTD and milc. Unfortunately, I observed that two Parsec applications, canneal and streamcluster, suffer from prohibitively high overheads , more than 100% for the case of canneal. Both benchmarks are read intensive and suffer high read L2 cache miss-rates. When a read block gets replaced from the L2 cache, Euripus looses the read information, and when the block is read again, Euripus will assume that it did not update the meta-data for this block and send it to the engine again. This behavior results in increased traffic to the engine, which becomes full more often and delays read operations. Delaying a read typically has higher performance cost than a write, because reads are part of the critical path of the execution, and delaying a read results in delaying all following instructions, including instruction fetch, which are dependent on the read's value.

Finally, Figure 34 presents the performance overhead of Euripus with read-tracking enabled for the worst performing applications and the averages for checkpointing intervals of 0.1, 0.5 and 1 second. As expected, reducing the checkpointing frequency, reduces the performance overheads. For the case of read-tracking at higher checkpointing frequencies, the read-tracking meta-data of the caches are more frequently cleared, resulting in the same read blocks to be sent to the Euripus engine again, increasing the occupancy of the engine and causing the same performance degradation as previously described.

3.7 Summary

Bidirectional debugging is a powerful debugging technique that allows program execution to proceed both forward and in reverse. Many software-only techniques and tools have emerged that use checkpointing and replay to provide the effect of reverse-execution, although with considerable performance overheads in both forward and reverse-execution. Recent hardware proposals for checkpointing and execution replay minimize these performance overheads, but in a way that prevents checkpoint consolidation, a key technique for reducing memory use while retaining the ability to reverse long periods of execution.

This chapter presents HARE and Euripus, two low-cost hardware techniques that efficiently support both checkpointing and consolidation. My experiments show that HARE and Euripus incur small (<3%) performace overheads even when checkpointing one hundrent times per second, provide reverse-execution times similar to forward execution times, and reduce the total space used by checkpoints by a factor of 36 on average (this factor gets better for longer runs) relative to prior consolidation-less hardware schemes. Euripus also provides functionality for creating both undo and redo-log checkpoints allowing the debugger to further reduce reverse-execution latency, while its ability to track read operations as well, can be leveraged to futher accelerate operations such as "reverse-continue".

CHAPTER IV

HYBRID HARDWARE ACCELERATED CHECKPOINTING FOR RECOVERY FROM A WIDE RANGE OF ERRORS AND DETECTION LATENCIES

System reliability is an increasingly challenging problem. A running program or the entire system can be affected by errors that range from transient errors in a processor, to permanent hardware failures, to software bugs, to human mistakes. A common method for recovering from errors is to create checkpoints during the execution of the program and, when an error is detected, restore the program to a previous error-free state and resume execution.

Errors can be caused at any level of the system architecture from the processor to the application, and can propagate through the hardware/software stack resulting in erroneous program execution or an application or system crash. Decreasing device sizes, as a result of Moore's Law, render processors increasingly vulnerable [13] to errors and failures, e.g. transient failures from particle strikes and intermittent or permanent faults due to wear-out phenomena. The reliability problem is well-known in massively parallel processing (MPP) systems, where the probability of a hardware error increases with the number of cores of the system and the Mean Time to Failure (MTTF) of a system can be as low as 6.5 hours [74]. Processor errors, either transient or permanent, still comprise only a fraction of all possible error types a system has to recover from (responsible for 42% of hardware outages [83]), while the other most common cause of failures are memory errors, followed by network, software or environment (e.g. power failures), etc. For the case of DRAM more specifically, a recent study [85], that was based on field data, demonstrated that one third of the machines will suffer a recoverable error, while 1.3% of them will encounter an uncorrectable one, which corresponds to error rates that are orders of magnitude higher than previously estimated.

To identify and recover from the multiple types of errors, systems can use two approaches: forward error recovery (FER) or backward error recovery (BER). Solutions which rely on FER use resource redundancy [7], which replicates structures of the system (e.g. processor), and execute the same operation (e.g. program) multiple times, using voting to select the final result. Such solutions provide systems with the highest degree of availability, but are very expensive in terms of power, cost and performance overheads.

The lower-cost BER approach relies on checkpoints/logs to restore execution to a safe point that existed before the occurrence of the error, then restart execution from there. The recovery time of BER is a function of the time to restore the state of the system (processor, memory, etc) and the time necessary to re-execute the program until the point of the error. This recovery time can be minimized through frequent checkpointing and low-latency error detection supported by low-cost hardware and software techniques [28, 71, 82, 107] which identify processor errors by their effects on higher levels of the system. All these solutions can identify and recover from an error within several milliseconds of program execution, but still there is a small percent, $< 2\%$, of processor errors [82] which can escape that recoverability window. Other errors, such as memory corruption, partial power supply failures, etc. can have even longer detection latencies. Finally, some failures, such as total loss of power, knock out the detection and recovery mechanisms along with the rest of the system, so recovery from them requires full-state checkpoints in non-volatile storage. Although long-detection-latency and catastrophic failures represent a small percentage of errors, they result in much longer recovery times, so they still have a major impact on the overall availability and recoverability of the system.

There are two types of memory checkpoints that BER can use, undo-log and redo-log checkpoints. Undo-log checkpoints [1, 68, 94] save the memory state necessary to restore the program from the current point in time to the point T in the past. Frequent undo-log checkpoints, combined with low-latency error detection mechanisms [82] can provide low recovery latency. The main disadvantage of undo-log checkpoints is that they require the “current” state of the memory to be preserved in spite of the failure, rendering them unsuitable for recovery from hard errors such as DRAM or power failures, or from errors that may have corrupted a large amount of in-memory state. Redo-log checkpoints can overcome this problem by restoring the system state starting from a full checkpoint (or the beginning of execution), then applying redo-log information to update the system to a more recent state. The disadvantage of redo-logs is that they require periodic creation of a full checkpoint, which includes a copy of the entire memory state of the system, in addition to the redo-log that keeps updates for recent modifications. In contrast, undo-log checkpoints use the existing "current" state of the system as the starting point. When checkpoints are frequent the modifications represent only a minuscule fraction of the total memory state. As a result, redo-log checkpoints have traditionally been created infrequently and stored on hard-disks using high performance I/O infrastructures [114]. In the future, though, as the number of processors increases, the increased checkpointing requirements and the limitations of disk storage technologies are expected to constrain the scaling of application performance [61].

In this chapter I apply the memory checkpointing hardware accelerator Euripus (Section 3.4), to the problem of reliability and I present Kyma, a hybrid checkpointing mechanism where both undo-log and redo-log checkpoints are created in order to provide the maximum recovery coverage and limit the recovery time from different types of system errors. Kyma has the following characteristics:

- Exploits the synergies between consecutive undo- and redo-log checkpoints, eliminating the need for multiple memory tracking mechanisms and reducing the total performance overhead, by checkpointing memory addresses common across consecutive undo and redo-logs only once.
- Uses a specialized hardware engine to create checkpoints in parallel with the program execution.
- Creates undo-log checkpoints frequently (e.g. every 10ms), for quick recovery from transient errors with short detection latencies, but also establishes incremental redo-log checkpoints less frequently (e.g. every 1s) to enable recovery from hard system errors or errors that escape early detection.
- Employs meta-data structures which enable the consolidation of redo-log checkpoints, creating additional checkpoints at different frequencies (e.g. every minute or hour) for no additional memory copying cost, assisting future multi-level checkpointing techniques.
- Provides high availability with the assistance of multi-level checkpointing.
- Allows the system to adjust the number and frequency of checkpoints depending on its error detection latency characteristics and available memory for storing checkpoints.

Overall, Kyma incurs overheads of $\sim 2\%$ on average even at high checkpointing frequencies (every 10ms for undo-log and 1sec for redo-log checkpoints), while creating a wave[1] of checkpoints that follows program execution (e.g. at 10ms, 1s, 1 minute, and 1 hour distances), enabling the efficient recovery from both catastrophic failures and non-catastrophic ones, and across a wide variety of detection latencies (milliseconds to one hour).

The rest of this chapter describes the different errors types and error detection techniques (Section 4.1), reviews existing checkpointing techniques (Section 4.2), provides an overview of Kyma (Section 4.3) and discusses Kyma's implementation details (Section 4.4), then presents the quantitative evaluation of Kyma (Section 4.5) and summarizes (Section 4.6).

4.1 Fault Types and Detection Mechanisms

System reliability is an increasingly important problem that does not only involve mission critical systems, such as satellites or space shuttles, but also affects other systems which are used for commercial or scientific purposes.

4.1.1 Fault Types

Today's computing systems and processors can suffer from multiple types of faults. Faults are typically divided in three categories [93]: transient, permanent and intermittent.

Transient Faults Transient is a fault that occurs only once and does not have a persistent behavior. When such faults manifest as errors are called *soft errors* or *single event upsets*. Typical causes of such faults are cosmic radiation [115, 116] and alpha particles [48].

Permanent Faults Permanent faults, also known as *hard faults*, are caused at some point in time, and can keep generating errors until the faulty component is replaced. Causes of such faults are, e.g. for the case of processors, physical wear-out phenomena caused by electromigration [18, 29], or a break down of a transistor's gate oxide [43, 62]. Other causes of permanent faults are fabrication defects that were not discovered during testing after fabrication, or design bugs [9] that were not exposed during the processor's validation process.

[1]Kyma means wave in Greek.

Intermittent Faults Intermittent faults occur for a period of time, but not continuously, generating multiple errors. Causes of such faults are components that are close to the end of their life-cycle, and cause multiple errors before failing permanently [20].

The previous types of errors do not only describe processor errors, but also errors than can occur in any component of the system, e.g. network, storage, or power.

4.1.2 Error Detection Mechanisms

There is a plethora of error detection techniques, focused on specific components of the system, or providing coverage for multiple types of errors. The most common error detection technique is based on redundancy, which can be implemented either as component, temporal, or finally information redundancy. Redundancy replicates computation/information, repeating each computation/operation multiple times and comparing the generated results. When the results do not much then an error has been detected and the system initiates recovery.

Component Redundancy Component redundancy replicates structures multiple times , e.g. N for n-modular redundancy (NMR) [105]. The most common form of modular redundancy is triple-modular redundancy (TMR) which is usually employed in critical systems, such as planes [113] or reliable systems such as the Tandem S2 [49], or the HP NonStop [7]. Common characteristic of all such systems is that they replicate components of the system (processor, memory, disks), which execute the same operations independently in lockstep and compare their results at the end. The correct result is selected based on the majority. Systems such as the one employed by Boeing [113] go a step further and use different devices for each replicated component, e.g. they use three different processors from 3 different manufacturers, in order to identify any potential design bugs that a specific processor might have.

TMR generally incurs high overheads in terms of power/performance/cost. Recent processor designs have enabled alternative methods of redundant execution. AR-SMT [77] makes use of SMT processors, executing a redundant thread in parallel with the original thread and periodically comparing the results. Mukherjee *et al*. [57, 75] follow a similar approach, but instead they make use of the additional cores that today's processors have, executing again replicas of the original program and comparing the generated results. Another form of redundant execution is DIVA [3], which uses a specialized checker processor, that verifies the correctness of the executed instructions before they get committed by the main processor.

Temporal Redundancy Temporal redundancy attempts to reduce the cost of component redundancy by requiring the same hardware resources as non component-redundant systems, but executing the same operation multiple times in different points in time. Temporal redundancy is based on the observation that transient errors affect the behavior of the system for a limited amount of time. For example, executing the same operation in the same functional unit is not going to suffer again from the same error, thus detecting any fault in the original computation. Temporal redundancy reduces the hardware requirements of the system, but still has a significant performance cost; the system's performance is half of a system that would execute no redundant operations.

Information Redundancy Information redundancy uses additional information in order to detect errors. An example are error-detecting codes (EDC) [106] which use additional bit information in order to encode the original information. The additional information allow EDC to detect and recover from errors by reconstructing the original information. Examples of EDC's are parity, where an additional parity bit is stored with every block of data, and ECC. Such codes are extensively used in today's architectures such as Alpha [37] or Power 4 [14], and allow the system to detect and recover from transient errors that affect structures such as the register file, the caches, or main memory. A limitation of EDC's is that, given the additional information they use, they can recover from a given number of errors. Unfortunately, EDC codes of data are not checked unless the data is accessed, thus it is highly possible that multiple errors, beyond repair, might accumulate to a memory location. For this reason memory scrubbing has been proposed [56] as a method to limit this phenomenon. Moreover, for the purpose of detecting and recovering from specific categories of memory faults, such as stuck-at faults, specialized techniques have been proposed [88].

Given redundancy's high costs, alternative error detection mechanisms have been developed which try not necessarily to identify the error itself, but instead its effects on the normal execution of the system. Such mechanisms try to identify and verify program invariants [38, 50, 73], by checking that the control flow [21, 46, 86] or the data flow [46, 50] of the program follows the same regular patterns. Other techniques, such as ReStore [107], try to identify micro-architectural behavior anomalies by inspecting rare hardware events (e.g. page faults), or are concerned only with the errors that affect the software and cause it to crash, as in SWAT [42, 82]. There are also software or is identified beyond the expected detection latency. Including such error detectors in the system will require to provide recovery mechanisms that can still recover the system if an error escapes early detection.

System recovery can be accomplished using two methods: Forward Error Recovery (FER) and Backwards Error Recovery (BER). FER is typically implemented with the use of NMR. When an error is detected, the system can recover immediately from the error by isolating the faulty component and continue executing with any components still operational. In BER, the system stops execution, replaces the faulty component if necessary, recovers to a past error-free state and resumes execution. Such a recovery mechanism is typically implemented with the use of checkpoints.

4.2 Checkpointing for Reliability

Checkpointing, as described in Chapter 2, is typically used by reliability [1, 68, 94] and debugging [110, 111] mechanisms which require to roll-back the program/system to a past state S that existed at time T. Undo-logging uses a log with the old values of the addresses which have been modified between time T and the next checkpoint (or present time). To roll back to state S, undo-logs are applied to the current state, starting with the most recent log, effectively undoing changes that were made since time T. The other approach, redo-logging, saves at time T the new values of locations that were modified between the previous checkpoint and time T. To roll back, it restores the state from an older full checkpoint C, then updates it with subsequently created logs until state S is reached. Figure 35 illustrates the two approaches. For a given time interval, between time T and $T + 1$, both approaches track memory modifications. For an address which is written for the first time in the current interval (e.g. *B*), undo-logging copies old data to the undo-log and marks it as checkpointed. A following write to an address which has already been checkpointed (*B*) does not result in another entry in the undo-log checkpoint. Note that log entries are finally ordered according to the order data addresses are modified by the program. To create a redo-log checkpoint for the time interval between T and $T + 1$, we just note which addresses have been modified by

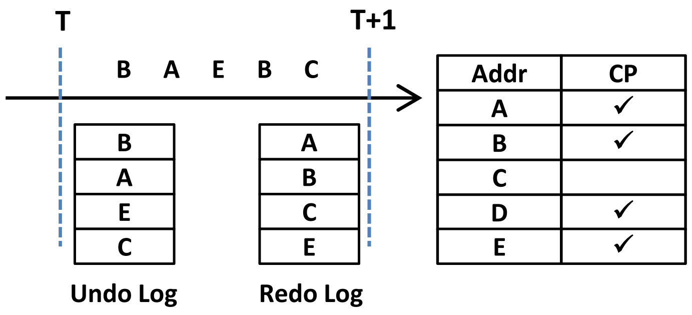


Figure 35: Undo-log and redo-log checkpoint creation.

the execution of the program and, at the end of the interval, copy the newest values of the modified addresses. This approach allows the log to be sorted by address (Figure 35).

Because rollback with undo-logging begins with the current state of the system, it can only be used if the "current" state is preserved, which limits its scope. In contrast, redo-logging provides recovery even if all of the current state is lost (e.g. by loss of power) but is less efficient because it saves data in bursts and needs to create full-state checkpoints. Note that redo-log checkpoints can only be used to recover from such catastrophic failures if they are saved to non-volatile memory. Conversely, undo-log checkpoints need not be saved to non-volatile memory, because they are rendered useless by failures that result in losing the current DRAM state.

In terms of implementation, checkpointing can be done at the level of the application [16], run time library [63] or the operating system [90, 95]. Software implementations typically keep track of the memory modifications at the page granularity, using existing page protection and dirty bit mechanisms, and suffer from significant overheads when checkpointing is done frequently, primarily because the frequent copying activity competes with the application for processor time, while frequent checkpointing causes some frequently modified blocks to be copied often (once to each checkpoint).

Hardware support is necessary for efficient and low overhead high frequency checkpointing. Hardware mechanisms [68, 89, 94] cause low performance overhead by efficiently tracking the memory modification at block based granularity and overlapping the program execution with the checkpoint creation. ReVive [68] and SafetyNet [94] create undo-log checkpoints at frequencies from 10ms to 100ms to recover from transient errors in the processors or the system (e.g. lost network packets). Both schemes modify the caches [94] or the coherency protocol [68] to keep track of the checkpointed blocks and log the old data of a block when it is modified for the first time in a checkpoint interval. The main difference between ReVive [68] and SafetyNet [94] is that it ReVive stores the checkpoints in memory, while SafetyNet uses only on-chip buffering, allowing ReVive to tolerate longer error detection latencies, as well as processor failures at the cost of more performance overhead.

To reduce the memory requirements of systems which need to maintain multiple checkpoints, as in bidirectional debugging, software solutions proposed the consolidation of checkpoints [12]. A consolidated checkpoint is the union of the set of addresses of the two input checkpoints, while duplicate entries (addresses common to both checkpoints) maintain the data of the latest checkpoint. In Chapter 3, I have presented two hardware accelerators, HARE and Euripus, which can create consolidation-friendly checkpoints. Both techniques are designed for the problem of bidirectional debugging, and have demonstrated that consolidatable checkpoints can be created at a low performance cost, while reducing by orders of magnitude the memory requirements of bidirectional debugging.

Another important consideration for checkpointing, especially redo-logging, is persistent storage. Frequent redo-log checkpointing would far exceed the available bandwidth of disk drives. A promising storage technology for frequent redo-logging is phase change memory (PCM), a non-volatile memory technology that is expected to scale beyond the technology limitations of DRAM in the future [35]. Compared to DRAM, PCM has a higher access latency and a limited number of writes that can be done over its lifetime. Still, in both aspects PCM is far superior to both disks and flash memory. Recent research results have demonstrated that the limitations of PCM can be overcome when it is used as the main system memory [40, 70], how its lifetime can be improved [87], and how to utilize PCM for constructing efficient file-systems [19]. Dong *et al.* [22] demonstrate through the use of analytical models the scalability of checkpointing schemes which use PCM memory in future exascale systems.

4.3 The Kyma Checkpointing Technique

Kyma is an adaptation of the Euripus hardware checkpointing technique for the purposes of reliability. In Kyma the goal is to provide multiple recovery points to the system in order to reduce the re-execution time during recovery and improve the efficiency/availability of the system. Unlike bidirectional debugging, where both types of checkpoints could be used interchangeably to restore a past program state and the type to be used was selected based on reducing reverse-execution's latency, in reliability the two types of checkpoints have limitations regarding the types of errors they can be used to recover the system from: undo-log checkpoints can be used to recover the system quickly from errors that have not corrupted the non-checkpointed system state, while redo-log checkpoints can recover the system from any type of error, at an increased recovery cost. The two types of checkpoints are used to recover different components of the system: undo-log checkpoints are used to recover quickly from transient errors, e.g processor errors, similar to ReVive [68], while redo-log are used to recover from catastrophic faults that happen in systems, e.g. non-recoverable memory errors, network errors, power failures, etc. Since undo-logs, are targeting transient or intermittent faults, which are typically detected quickly enough, I have decided in Kyma to maintain a limited number of undo-log checkpoints, similar to other hardware recovery techniques that target the same type of errors [1, 68, 94], and to rely on redo-logs to recover from errors that escape early detection and there is not an undo-log checkpoint to be used to recover from.

Kyma exploits the synergies between undo and redo-log checkpoints similar to Euripus (Section 3.4.2) to create both undo- and redo-log checkpoints efficiently, and to overcome the performance, storage bandwidth, and recovery latency limitations of prior checkpointing schemes. Kyma creates undo-log checkpoints frequently (10ms - 100ms) to provide low recovery latency from early-detected errors. It then exploits the synergy between undo- and redo-logs to efficiently create redo-logs at second-level intervals that enable recovery from more severe errors. Because some error detection latencies can be longer than one second, Kyma consolidates redo-log checkpoints to also create minute- or hour-level redo-log checkpoints. A full checkpoint is also maintained, created at the beginning of the execution and updated by the incremental redo-log checkpoints. These redo-log checkpoints allow Kyma to recover quickly from errors that escape the detection latency of mechanisms which rely on undo-log checkpoints [82], or which cannot be recovered from using undo-logs (e.g. because of a DRAM error).

Kyma is a checkpointing solution which is orthogonal to existing error-detection mechanisms

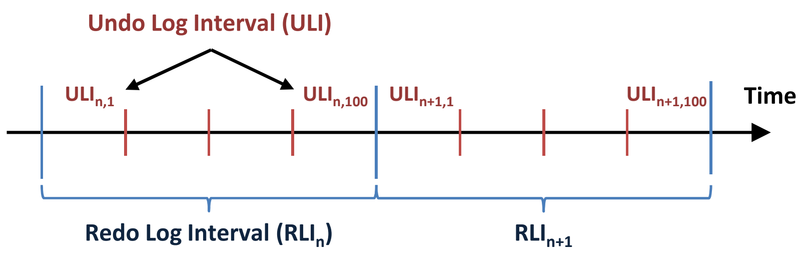


Figure 36: Distribution of undo-log and redo-log intervals over time.

and can adjust without incurring significant additional performance overheads to specific checkpointing configurations, as will be shown in the evaluation (Section 4.5). The exact number of undo- and redo-log checkpoints, the frequency at which they are created, and the distribution of incremental redo-log checkpoints over-time are decided by the recovery mechanism [22, 53] based on the following parameters: 1) the expected frequency of errors at every checkpointing level, 2) the resiliency of the checkpointing level to errors (e.g. checkpoints stored in DRAM) 3) the latency of error detection mechanisms, 4) the size, and 5) the available memory of the system. For the purpose of describing and evaluating Kyma I am assuming the frequencies and distribution of checkpoints shown in Figures 36 and 37.

Kyma creates undo-log checkpoints (UL) every 10ms and maintains a limited number of completed checkpoints (e.g. 2), plus one that is under construction; this approach allows Kyma to rollback the program state by several milli-seconds (e.g. 30) using undo-logs (Figure 37). Undo-log checkpoints older than the selected number are discarded (and their memory freed/recycled). While creating undo-logs, Kyma also tracks modified blocks for the next redo-log checkpoint (which is in the modification-tracking stage at the time) and leverages undo-log data copying to save data to the previous redo-log checkpoint (which is in the data-copying stage at the time). A redo-log checkpoint is completed every second; after a new redo-log checkpoint is created, the previous one is consolidated into a consolidated redo-log (CRL) checkpoint for the current minute-long interval. At the end of the minute, the minute-scale CRL is completed and the prior one is then consolidated into the hourly checkpoint for the current hour-long interval. At the end of the hour, the new hour-scale checkpoint is complete and the old one is used to update the full checkpoint, which is assumed to be stored on disk. To restore the system state after a hard failure (or an error whose latency is

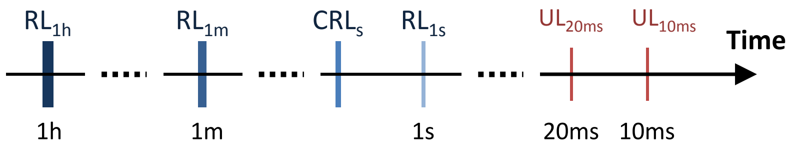


Figure 37: Distribution of undo-logs and redo-logs checkpoints over time.

longer than 30ms), Kyma restores the full checkpoint and then applies hour- , minute-, and second-scale checkpoints until the error latency interval is reached[2]. For example, if the error detection latency for a particular error is one hour or less, Kyma would just restore the full checkpoint. If the detection latency is one second, Kyma restores all but the second-scale checkpoints.

The hierarchy of checkpoints in Kyma also provides for bandwidth management for memory, PCM, and disk. Undo-log checkpoints are created frequently (10-100ms), so they require very high write bandwidth (up to 900MB/s for some applications (Figure 38). This bandwidth can be supported by DRAM memory, where Kyma stores undo-log checkpoints. Redo-log checkpoints created each second still require significant bandwidth (almost 800MB/s for the SPECFP application [96]), but are saved in PCM that can sustain this write bandwidth. Consolidation of redo-logs only requires meta-data changes (no copying of data blocks) and consumes far less bandwidth than data copying. Finally, hour-scale updates are sent to disk, but the bandwidth requirements for this can easily be sustained by modern disk drives – as shown in Figure 38, even if minute-scale checkpoints were sent to disk, the required bandwidth would be only up to 35MB/s. Most importantly the construction of the redo-logs at the different levels is not part of the critical path of the application's execution: the application does not have to stop execution in order to create the create the checkpoints, since checkpoint consolidation can be done in the background by Kyma's hardware engine, or an idle processor.

To create the two types of checkpoints, Kyma uses the following approach: A) For undo-log checkpoints, Kyma keeps track of the checkpointed blocks in a given undo-log interval (ULI) as described in Section 4.4.1. B) For establishing redo-log checkpoints Kyma takes advantage of the two properties of the undo-log and redo-log checkpoints, similar to Euripus. First, Kyma uses the

[2]To avoid over-writing multiple times addresses common across checkpoints Kyma's meta-data allow to consolidate all incremental redo-log checkpoints first into a single full checkpoint.

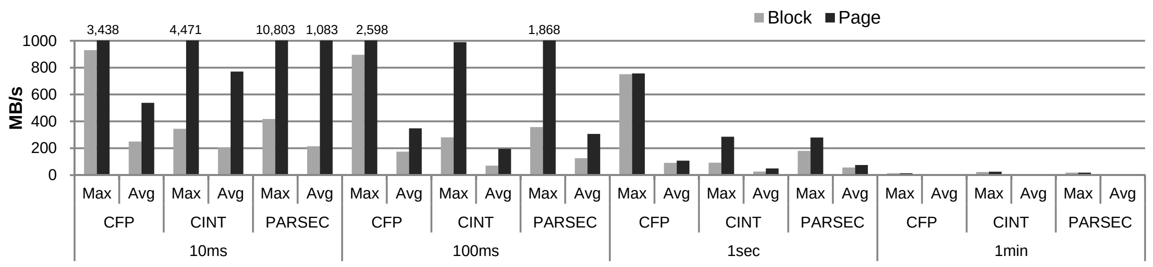


Figure 38: Maximum and average memory bandwidth requirements in MB/s of checkpointing mechanisms for different checkpointing intervals (in seconds) when using block or page tracking granularities for the SPEC 2006 [96] floating point (CFP), integer (CINT) and PARSEC 2.1 [8] benchmarks.

meta-data of the multiple undo-log checkpoints that are created during a redo-log interval (RLI) (Figure 36) in order to construct the set of modified addresses for the given RLI and create a redo-log checkpoint. Second, Kyma identifies the block addresses which belong both to the current undo-log intervals ($ULI_{n+1,1}, \ldots, ULI_{n+1,100}$) and to the previous redo-log interval (RLI_n). The first time such block is identified (e.g. in $ULI_{n+1,1}$) it is stored directly to the redo-log checkpoint for RLI_n in PCM, and the undo-log meta-data of $ULI_{n+1,1}$ is updated to point to the same block in PCM. If this address is found again in following ULIs, it is saved to DRAM because it does not represent the newest values for RLI_n. This method eliminates future reads and writes that Kyma would have to perform for creating the redo-log checkpoint of RLI_n, reducing the memory bandwidth requirements.

Another important problem when restoring a system is recovering I/O (e.g. network). Kyma is orthogonal to existing techniques for restoring I/O (e.g. ReViveI/O [58]). The support for frequent undo-log checkpointing and quick recovery from transient errors can reduce the necessary buffering. Section 4.4.4 further describes how Kyma interacts with I/O.

4.4 Implementation Details of Kyma

4.4.1 Undo-Log Checkpointing

Kyma's undo-logging uses a mechanism similar to the one used by Euripus's prior techniques, ReVive [68] and SafetyNet [94]. It uses an extra *checkpoint*-bit in the tag arrays for the L1 and L2 caches to identify which blocks were already saved in the current interval. When a block is written in L1, the checkpoint-bit is checked. If this bit is 1, the block has already been saved in the current undo-log interval and not further Kyma action is needed. If the checkpoint-bit is 0, the block's data is sent to the undo-log, the checkpoint-bit is set to 1, and only then the write proceeds to modify the data. When the block is written back to the L2 cache the checkpoint-bit is written back as well, but for blocks written back and replaced from L2 the checkpoint- bit information is not preserved (to avoid having to keep these bits for all blocks in memory). Thus, when a block is brought into the L2 cache on a miss, its checkpoint-bit is assumed to be 0. Since Kyma does not need to consolidate the undo-log checkpoints, the meta-data used is a simple list of address, and not a trie as in the case of Euripus, where every field contains the address checkpointed and a pointer to where the data is being store – the data checkpointed can be shared between the undo and the redo-logs and can reside in either DRAM of PCM.

The undo-log checkpointing approach that Kyma uses, creates global undo-log checkpoints: when a multi-threaded application crashes, then all the application's threads are restarted, even though they might have not suffered an error, or consumed incorrect data generated from a thread which experienced a fault. An alternative undo-log checkpointing technique is to create coordinated undo-log checkpoints, similar to Rebound [1]. Regardless of the underlying undo-log construction mechanism used, the only requirement for Kyma to function correctly is that the redo-log memory-tracking meta-data is updated, which can be implemented by sending to the Kyma hardware engine the addresses of the undo-log blocks that were checkpointed.

A second difference between Euripus's and Kyma's undo-log mechanisms is that Kyma has to write back to main memory all dirty cache-lines at the end of the interval. Unlike bidirectional debugging, in reliability the system has to ensure that the main memory has the full memory state of the system at the end of every interval. If an error happens in the processor, the assumption is that part, if not all, the memory state cached on the processor is not accessible any more, and it is possible that dirty memory locations, that were not checkpointed in the current interval, are not written back to memory. In such a case, the last undo-log will not be able to recover the system, because there are modified memory locations which were not checkpointed and main memory does not hold their latest values.

To write back all dirty lines, one approach is to stop execution until write back finishes, an approach that would affect the overall performance when trying to write-back the contents of big

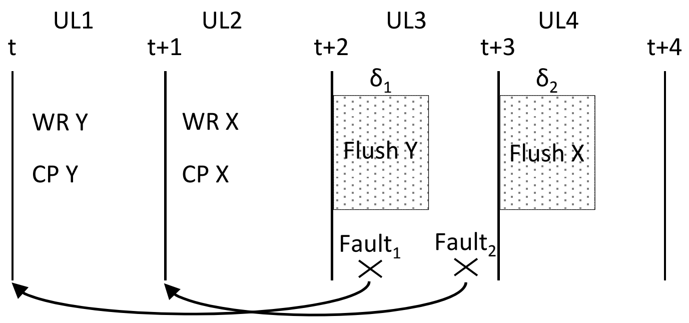


Figure 39: Implications of delayed cache-flushing on error recovery.

shared last level caches[3]. Kyma, similar to other techniques [1], does a delayed write-back, where the controller of the cache writes back all dirty lines while the execution continues. To simplify the complexity of the write back mechanism, Kyma forces all the private caches of each core to write back to the last level shared cache, during which time execution is paused. Following, the last level cache updates the main memory in a delayed fashion.

To reduce the number of write-backs, Kyma does not checkpoint the dirty blocks that have already been checkpointed in the current interval. To identify such blocks, at the end of the checkpointing interval, Kyma marks the blocks to be flushed and excludes the blocks that have the checkpoint-bit set. This approach is also applied in the private caches when they flush their data to the shared last level cache, resulting in only a limited number of blocks to be written back, since private caches tend to be small and the majority of the data they hold tend to be recently accessed and thus checkpointed. If a block marked to be written-back gets to be updated, e.g. from a write-back from a L1 cache, then that block is immediately written-back to memory. Kyma's optimization does not affect the correctness of the recovery mechanism. When the delayed cache-flush finishes, the memory has the oldest values of all memory locations at the beginning of the undo-log interval, with the exception of the memory locations which have been checkpointed, whose values are stored in the checkpoint log.

The delayed cache-flush approach affects the checkpoints which can be used to recover the system when an error occurs. Figure 39 shows the delayed cache-flush process for a number of intervals. Assume that the undo-log checkpoint under construction is *UL2*. At time *t+2* the contents

[3]Today's architectures have more than 8MB of shared L2 or L3 caches.

of the cache have to be written back to ensure the consistency of memory. The block to be written back is block Y, because it was modified in interval *UL1*, but was not replaced from the cache in the meantime. Block X does not need to be written back at the end of the interval because it is already checkpointed. The undo-log checkpoint for interval *UL2* is not formally established until all blocks to be flush are written to memory after time δ_1. If an error happens during time δ_1 then the main memory state is not going to be consistent, address *Y* is not going to have the correct value, and the undo log of *UL2* interval cannot be used for recovery to time *t+1*, instead the system will have to recover to time *t*. If an error happens after time δ_1 then the state of the main memory is going to be consistent and the system will be able to recover to time *t+1*. I will investigate the effects of cache-flush delay time in the reliability of the system in Section 4.5.3.

4.4.2 Redo-Log Checkpoint Creation and Organization

To construct a redo-log checkpoint at the end of a RLI_n, during that RLI_n we need to track the set of modified address for that interval. As we pointed out in Section 4.3, this set of modified addresses is the same as the set of addresses that were checkpointed by the (many) undo-logs during this redo-log interval. Therefore, whenever a block is sent to an undo-log checkpoint $ULI_{n,m}$, its address is also added to the redo-log meta-data for interval RLI_n. At the end of RLI_n, this meta-data is traversed and the included blocks are copied from DRAM to PCM. Kyma could be used to construct only redo-log checkpoints if necessary. This can be accomplished by using the undo-log mechanism to just update the redo-log meta-data and not copy the undo-log blocks and meta-data to memory.

As indicated in Section 4.3, a significant number of blocks that belong to the redo-log checkpoint RLI_n are also going to be copied by future undo-log checkpoints $ULI_{n+1,m}$. For this reason Kyma, similar to Euripus, does not start the copying of RLI_n data as soon as that interval ends. Instead, it waits for a period of time in RLI_{n+1}, during which blocks that belong to RLI_n are saved to PCM by undo-log activity. This “repurposing” of undo-log writes is done by checking, for each block that should be saved to the undo-log for $ULI_{n+1,m}$, if 1) the block’s address is present in the redo-log meta-data for RLI_n and 2) the corresponding data has not been saved yet. In such a case, the data is saved to PCM, the RLI_n meta-data is updated to point to the saved data, and the $ULI_{n+1,m}$ log is made to point to the same block in PCM.

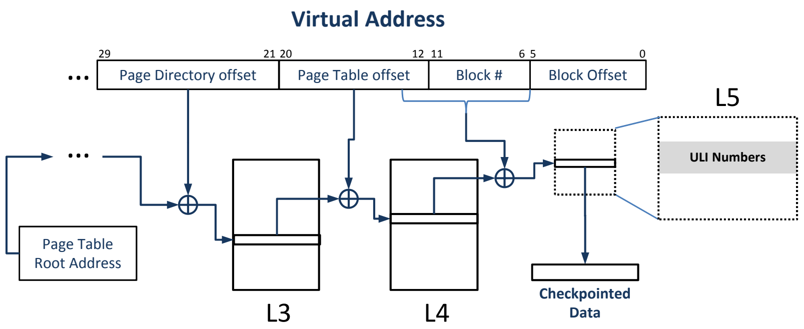


Figure 40: Redo-log checkpoint meta-data trie data structure used by Kyma.

Kyma uses the same trie data-structure (Figure 40) as Euripus, with the only difference that since Kyma is not creating consolidatable undo-logs – the trie is less complex and is used only as the memory tracking and checkpoint meta-data of the redo-log. Similar to Euripus, Kyma's redo-log meta-data serves as a secondary filtering mechanism for the undo-log blocks to be checkpointed. To perform this secondary filtering, each entry in the redo-log trie also keeps the number of the last undo-long interval that checkpointed that entry. If the check finds the number of the current undo-log interval in the redo-log entry, the block has already been saved and does not need to be saved again (neither in the undo-log nor in any redo-log). Since there is a limited number of undo-log intervals within a redo-log interval (up to 100, for 10ms undo-log and 1 second redo-log intervals), this only requires 7 bits in the redo-log entry, and Kyma uses the block offset bits and the unused bits of the virtual address [4] (which are not needed) to store the ULI number. The ULI number of every block is a lot more frequently accessed than the pointers to the checkpointed data, and similarly to Euripus, they are all stored in a header in the L5 node of the trie.

It is possible (and highly likely) that the sets of modified address of two consecutive redo-log intervals, RLI_n and RLI_{n+1}, are not exactly the same. Therefore, at some point Kyma must stop waiting for undo-logging during RLI_{n+1} to save data that belongs to RLI_n, and start actively copying the remaining data for RLI_n to PCM. As a first approach, I chose a static redo-log scheduling policy that starts redo-log construction at the mid-point of interval RLI_{n+1} (half a second after RLI_n ends). At that point, the Kyma engine traverses the RLI_n meta-data and, for each block that was not already copied (by undo-log activity), reads the block's data from the application's memory

[4] In x86 64-bit architectures the physical address-space is limited to 48 bits [33].

and saves it to PCM. Meanwhile, Kyma continues monitoring undo-log activity and saving to PCM any blocks that belong to RLI_n but have not yet been reached by Kyma's copying activity. This monitoring prevents the application from overwriting any data that should have been saved to RLI_n – any first modification in RLI_{n+1} will find the *checkpoint* bit in the cache to be zero, and thus sends the current (before the modification) value of the block to the undo-logging mechanism. The undo-log mechanism will in turn check the meta-data for the prior redo-log checkpoint, saving the block to PCM and updating the pointer in the redo-log.

The static policy that I just described, which is also used by Euripus, imposes two potential performance constraints: First, it entails the danger that the remaining time will not be sufficient to finish the construction of the redo-log checkpoint, and the application's execution will have to stall. Euripus did not suffer from this problems because the redo-log checkpoints were created in DRAM, while Kyma's are stored in PCM, which has approximately one order of magnitude higher write memory latency than DRAM. The best policy would be the one that adjusts the beginning of the redo-log construction processes based on the estimate of the expected redo log construction time. Second, the starting time of active redo-log copying is not based on an estimate of the synergistic copying that has taken place and can potentially happen in the future. The engine has a bursty memory access pattern, copying memory blocks aggressively, and leaving little opportunity for any further synergistic-copying. Ideally the engine should enable redo-log construction only after it can predict no further synergistic copying will take place. Making such a decision is difficult, because it requires the engine to maintain extensive profiling information about the behavior of the program, and be able to predict future memory accesses. A simpler approach is to throttle the redo-log copying process, allowing only a limited number of outstanding memory copy operations. By delaying the redo-log construction processes, but not completely stopping it, Kyma can increase the number of potential synergistic copies, while guarantying that the redo-log is constructed in time: if the time left to construct the redo-log is not sufficient under throttling, then it is disabled and the engine starts copying at full speed. Throttling not only increases the number of synergistic copies that take place, but it also reduces the competition for off-chip bandwidth between the application and the engine. Non-throttled checkpointing has a bursty memory access pattern, which allows the checkpoint construction to complete in time, but limits the available bandwidth to the application, a phenomenon that I highlighted in the comparison of Euripus with HARE (Figure 17).

To implement an adaptive redo-log construction policy that starts redo-log construction early enough, as well as throttles redo-log copying, while ensuring timely construction, the following profiling information is necessary: 1) the rate of synergistic copies, 2) the redo-log block copy rate at the current throttling level and 3) the average latency per checkpointed block when redo-log construction happens at full-speed. To estimate the three rates described, the Kyma engine periodically samples its behavior and keeps statistics about the average number of synergistic copies and blocks copied in the previous sample. The average latency of full-speed copying is computed by the engine by periodically creating checkpoints where throttling is disabled. Using this information the Kyma engine can estimate if under the current throttling rate the redo-log construction will finish in time. If not, then the engine increases the number of allowed outstanding memory copies. If the new throttling rate again cannot deliver the checkpoint in time, then it is further decreased until the point where the engine will not be able to create the checkpoint in time unless no throttling is applied. The adaptive redo-log construction policy is agnostic to the underlying memory system, DRAM or PCM, and could be directly applied to Euripus as well.

The system manages the available PCM memory as a free list of blocks maintained by the OS. When copying a block to PCM, Kyma gets the PCM location from one end of the free list. When redo-logs are consolidated, freed blocks are returned to the other end of the free list. This “rotation” of blocks through the free list helps prevent excessive writes to any one block of PCM memory.

Finally, the redo-log meta-data are stored in DRAM and the internal Kyma engine caches them during the time period when they are updated. Only after all the blocks of the redo-log checkpoint have been copied to PCM, Kyma starts copying the corresponding meta-data to PCM, starting from the leaves of the trie structure. When the root of the meta-data and the corresponding pointer to it have been stored to PCM, the checkpoint is complete and becomes part of the redo-log recovery state.

4.4.3 Kyma Engine Description

The Kyma hardware engine (Figure 41) is responsible for constructing both the undo-log and the redo-log checkpoints, and is structurally identical to Euripus’s engine. It is a structure separate from

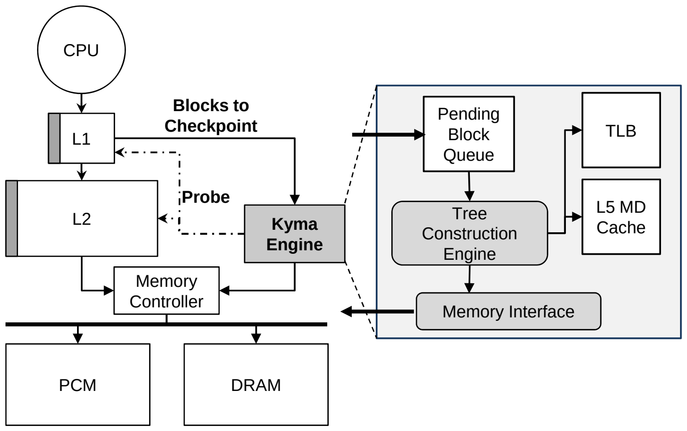


Figure 41: Architecture of the Kyma hardware engine.

the cores of the CMP, and is positioned close to the on-chip memory controller for minimizing the latency to memory. The number of Kyma engines on chip will depend on the number of cores, and the checkpointing requirements of future systems. The engine receives the blocks to be checkpointed from the L1 caches of the cores[5]. In order to differentiate between the processes/threads running concurrently on the CMP, along with the data to be checkpointed, the L1 also sends the number of the core which modified the block. Based on the core number the Kyma engine selects the appropriate undo-log and redo-log checkpoints to insert the data. The OS is responsible for programming and managing the Kyma engine. When a new process/thread is scheduled in one of the cores, the OS updates the Kyma engine registers with the following values: 1) the pointers of the undo-log data and meta-data, 2) the pointers to the roots of the redo-log meta-data for the current and the previous redo-log intervals (RLI_{n+1} and RLI_n), and 3) the core where it will be running. Threads of the same processes, which typically share the same address-space, will have the same set of pointers to undo and redo-log meta-data.

The differences between the Euripus and the Kyma engine are only limited to the algorithm processing incoming blocks to be checkpointed, based on the changes I made in the undo and redo-log construction process. Once a core sends the blocks to be saved in the Kyma engine, they are inserted in the pending queue. Every block in the pending queue is first processed by the *Tree Construction Engine* (TCE), which updates the meta-data for the current redo-log (RLI_{n+1}), performs secondary

[5]In our architecture we are assuming a snooping cache coherency protocol. In the case of a directory based protocol, the directory would be responsible for identifying the blocks to be checkpointed, similar to ReVive [68]

filtering of undo-log writes, and following uses RLI_n meta-data to decide whether the block will be saved to DRAM or PCM. Once the block is processed by the TCE it is forwarded to the memory interface, which writes the block to either DRAM or PCM and updates the undo-log meta-data, by issuing requests to the memory interface.

4.4.4 I/O and Multiprocessor Issues

System recovery in the presence of I/O operations poses challenges regarding the correctness of the state of a system. For example, if during a checkpointing interval the system writes to disk and an error happens, recovering only the memory state to a past state, while not also handling the disk state, will leave the system in an inconsistent state. This problem is known as the *output-commit problem* [26], and has been addressed for the case of undo-log checkpointing in prior hardware-assisted mechanisms, such as ReViveI/O [58], where the I/O is delayed until the end of the current checkpointing interval. This approach guarantees that no error is detected after the I/O is committed for the current intervals and the system will not have to recover to a point in time before the commit point.

Kyma is a checkpointing technique which is orthogonal to existing I/O recovery mechanisms. For the case of the output-commit problem, Kyma follows the same approach as prior techniques where the undo-log checkpoint is terminated early in the presence of I/O. When recovering the system using redo-log checkpoints, I am assuming that the Kyma technique is coordinating the redo-log construction frequencies and intervals with a global recovery mechanism: e.g. Kyma can be creating local checkpoints in a distributed system as part of a coordinated recovery mechanism, and the recovery mechanism is responsible for restoring individual nodes to the correct point in time and restarting network I/O.

Apart from the output commit problem, it is also required that the system correctly keeps track of the memory state modified by I/O, e.g. a DMA write. Assuming that the system is kept coherent, when DMA writes to memory any blocks cached on the processor get invalidated. When a DMA invalidation is sent, the old value of the block is sent to the Kyma engine which updates the undo-log. As part of Kyma's operation the redo-log meta-data is also going to be updated, and the modified memory location is going to be checkpointed at the end of the redo-log interval as well.

Multi-processor systems also pose a challenge for constructing global checkpoints. A common characteristic of today's architectures which can assist, is that the memory controller has moved on-chip [34]. In such multi-processor systems the memory controller of each processor is responsible for managing a specific subset of the physical memory. Since the Kyma engine is located close to a specific memory controller, it is going to be responsible for checkpointing only modifications to the physical memory managed by a specific memory controller. Kyma will construct a redo-log checkpoint per memory controller, while the redo-log trie is partitioned across all memory controllers/Kyma engines, with every Kyma engine managing a different segment of the trie. The problem which arises, because of this design decision, is how Kyma handles memory locations, managed by a specific memory controller, which get modified in a core on a different processor. For the case of global undo-log checkpointing, to ensure correctness, the old value of the modified block should not be sent to the local Kyma engine, but to the one managing this specific physical memory subset. Such a solution would consume more off-chip interconnect bandwidth, and an undo-log checkpointing approach which creates co-ordinated undo-log checkpoints would be more appropriate in such a configuration. Regarding the redo-log meta-data, Kyma has to ensure that they are updated correctly. A first approach is to send only the address of the modified block to the managing Kyma engine, which will update the correct segment of the trie. Another approach is that the address of the block is sent to the local processor's Kyma engine, and the checkpointing engines maintain the consistency of the redo-log meta-data by participating in cache coherence. Note that the Kyma engine already has to participate in cache-coherence, because it needs to send probes to the caches during redo-log construction time in order to checkpoint the latest values. The second approach is expected to have lower interconnect traffic overhead, if accesses to data that belong to a different memory controller are frequent, and caching of the redo-log meta-data has high hit rates. Yet this approach increases the complexity of the Kyma engine, which will also have to ensure the consistency of the redo-log meta-data updates: e.g. when nodes are added to the trie the operation has to happen atomically, and the same child-node is not added multiple times.

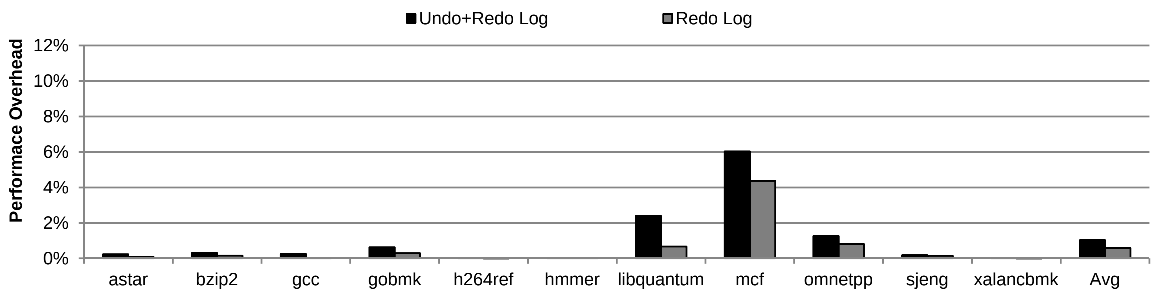
4.5 Evaluation of Kyma

I quantitatively evaluate the hardware cost, the performance overhead and the memory requirements of Kyma.

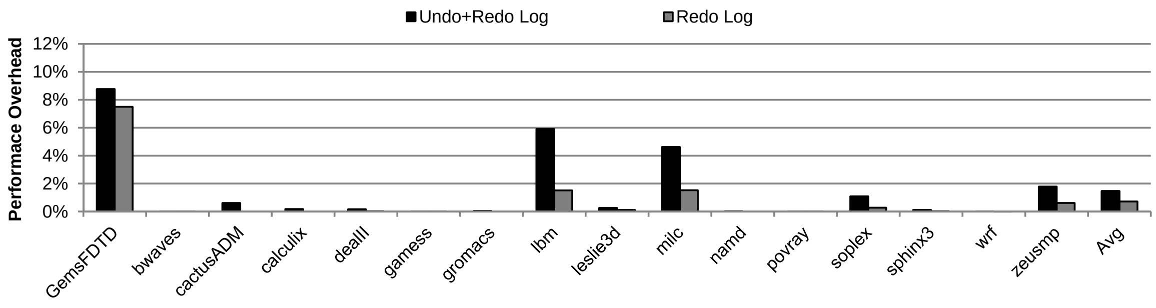
4.5.1 Evaluation Setup

For Kyma's evaluation, I use SESC [76], an open source execution driven simulator, to model a four-core CMP system with Core2-like parameters: 4-issue, out-of-order cores running at 2.93GHz. Each core has a private dual-ported 32KB 8-way associative L1 data cache. All cores share a 4MB, 16-way associative, single-ported L2 cache. The block size is 64 bytes. I use a detailed DRAM-model to simulate a DDR3-800-like memory system. PCM memory uses a different memory channel than DRAM and has an average read latency of 150ns and write latency of 450ns (450 and 1350 cycles respectively) [87]. The simulated Kyma engine has a 64 entry pending block queue, a 256 entry fully associative trie TLB, a 64KB 16-way associative single-ported L5 MD cache, and the memory interface has a 32 entry read queue and a 128 entry write queue. In total, the Kyma engine requires $\sim$82KB of on-chip state, which is lower than 256KB in SafetyNet [94]. Because this state is kept in area-optimized, single ported arrays, its area is approximately 20% smaller than the area of a single core's L1 cache (estimated using CACTI 5.3 [99])

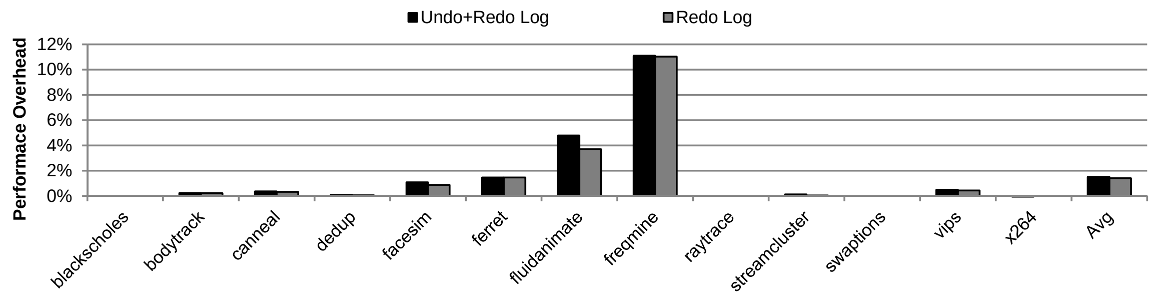
The evaluation uses 27 of the 29 SPEC 2006 [96] benchmarks, shown in Figure 42. The only benchmarks omitted are tonto and perl because of incompatibilities with our simulator infrastructure.I simulate SPEC benchmarks using reference inputs, by fast-forwarding through 5% (up to a maximum of 20 billion instructions) of the execution in order to skip the program's initialization, then simulating 10 billion instructions. I also evaluate Kyma with all 13 multi-threaded benchmarks from PARSEC 2.1 [8], using native inputs and four threads. The exception is dedup where the sim-large input is used, because the native input exceeds the addresses-space of the 32-bit MIPS-Linux simulated machine in SESC. I fast-forward the PARSEC applications to the beginning of the parallel execution, fast-forward an additional 21 billion instructions to warm-up the memory tracking mechanisms of all undo- and redo-logging techniques, then simulate 20 billion instructions in detail. The number of instructions I am simulating give me the opportunity to approximate a few seconds (1-5 depending on the application's IPC) of the program's execution. As described in Section 4.4.2,



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 42: Performance overhead comparison of Kyma when both undo and redo-logs and only redo-log are created for undo-log interval of 0.01 sec and redo-log interval of 1 sec.

Kyma's adaptive redo-log creation policy relies on profilling information that are collected at run-time. Given the limited number of redo-log checkpointing intervals than can be simulated, I decided the full-speed copy rate information that the Kyma-engine needs to be collected off-line, and not use a redo-log intervals to collect them, so that the steady state of the system is simulated in the experiments, where all profiling information is available.

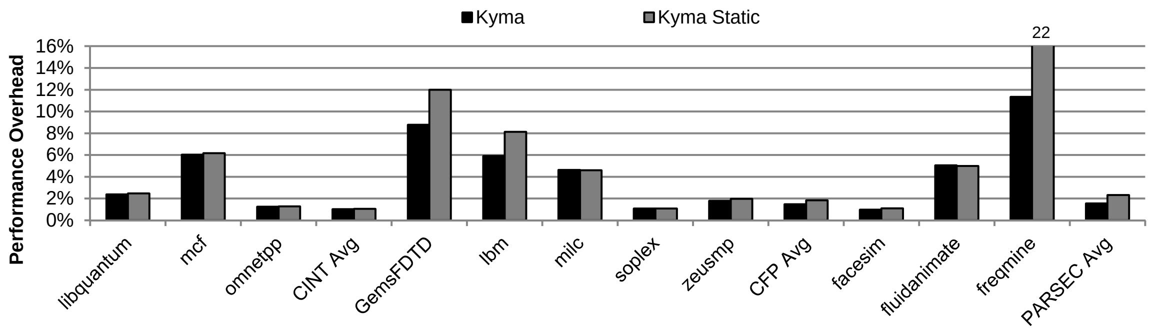


Figure 43: Performance overhead comparison of Kyma's adaptive checkpointing policy with the static policy, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

4.5.2 Performance Overhead of Kyma

Figure 42 presents the performance overhead of Kyma, for all simulated benchmarks, when both undo and redo-log checkpoints are created (Undo+Redo Log) and when only redo-logs are constructed (Redo Log). In this experiment the undo-log checkpointing interval is 10ms and the redo-log 1sec and consolidation of redo-log checkpoints is enabled. For the case of Kyma a breakdown of the overhead cannot be provided for the undo and the redo log construction costs, as for the case of Euripus (Figure 21), because the secondary filtering and elimination of duplicate undo-log blocks is not performed by the undo-logging mechanism, as in Euripus, but by the redo-log meta-data instead. As a result, the performance cost of undo-logging is higher when redo-logging is disabled. For Kyma, the additional cost of writing the undo-log data and meta-data to memory can be inferred by comparing the overheads of the full Kyma scheme with the redo-log only checkpointing variation.

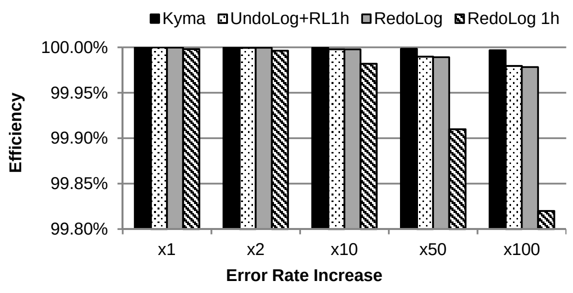
Overall, Kyma has average performance overheads of $\sim$2%, and the maximum overheads are $\sim$9% for GemsFDTD and $\sim$11% for freqmine. The benchmarks where Kyma has the highest overheads are applications which are already memory intensive, memory bandwidth constrained and are creating the largest checkpoints across all applications I have evaluated. Kyma's overheads are still low in these applications, because Kyma efficiently reduces the memory bandwidth needs of redo-log checkpoints when compared to other checkpointing techniques.

Figure 43 shows the performance overhead of the worst performing applications for Kyma's adaptive redo-log checkpointing policy (where redo-log construction starts early if necessary and is throttled in order to increase the effect of synergistic copying) with the static policy (which starts copying the remaining redo-log blocks at full-speed after the first half of the redo-log interval). The static policy further increases the performance overheads of the worst behaving applications such as GemsFDTD, lbm and freqmine. The performance improvement of the adaptive over the static policy can be attributed for the cases for GemsFDTD and lbm to the increased synergistic copying that takes place because of throttling the redo-log construction rate, while for the case of freqmine to the early start of the redo-log construction, which finishes in time and prevents any unnecessary pauses of the application's execution. Further details about the advantages of the adaptive policy over the static will be provided in Section 4.5.5.

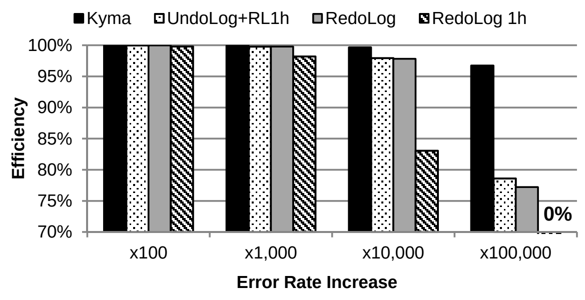
4.5.3 Availability Analysis

To estimate the efficiency/availability of a system that employs Kyma as its checkpointing mechanism an analytical model is used, that estimates the availability of the systems that create multiple levels of checkpoints. The model is described in detail in Appendix A and is an extension of the model proposed by Moody *et al.* [52], adapted to describe a system that creates both undo and redo-log checkpoints, which are constructed in a lazy fashion, as Kyma does. The model assumes an infinite amount of spare resources, if a failure happens then the recovery cost includes the cost of restoring the checkpoint, and not waiting for the faulty component to be replaced. The parameters of the model are: First, the error rate that the system experiences, which is distributed across the different checkpointing levels. The model assumes that each checkpointing level can recover a specific subset of errors, and when multiple checkpoints can recover the same type of error, then the latest one created will be used for recovery, even though there might exist an older one which can restore the system at a lower cost. Second, the checkpoint creation and recovery costs. Kyma creates checkpoints in parallel with the application's execution, so there is no checkpoint creation cost, but instead the incurred performance overhead. Finally, the distribution of error rates across the different levels.

The efficiency that a system using Kyma can achieve (by maintaining two undo log checkpoints constructed every 10ms and redo-logs every 1 second, minute and hour) is compared with three different checkpoint organizations: 1) a system that creates undo log checkpoints at the same frequency as Kyma but creates only one redo-log checkpoint every 1 hour (UndoLog+RL1h), 2) a



(a)



(b)

Figure 44: Achieved efficiency of a system using Kyma, compared to other checkpointing approaches, for increasing error rates.

system that creates only redo-log checkpoints at the same frequencies as Kyma (RedoLog) ,and 3) a system that creates a single redo-log checkpoint every 1 hour (RedoLog 1h).

Regarding the configuration of the system, given the limitations of the simulator/profiling infrastructure, I am assuming that all types of checkpoints are stored in PCM. I profiled in the simulator the time to restore a full checkpoint from PCM and the average latency is 1 second across all applications. Moreover, when recovering to the latest incremental redo-log, multiple checkpoints have to be restored. For this reason, I estimated the average percentage that the 1 second and 1 minute checkpoint are of the full checkpoints (Figure 54), and their respective recovery latencies are 1.75 and 1.5 seconds. Regarding the error rate that the system is going to experience, the average rate per second per processor was estimated based on the results reported by Moody *et al.* [52] and Schroeder *et al.* [83], which is 10^{-8}, which corresponds to approximately 3 errors per year. Regarding the distribution of errors through the system, it is assumed that they follow an exponential distribution. This distribution is approximated by computing the error rate r_i at level i as $r_i = \alpha \cdot r_{i-1}$ where $\alpha \leq 1$ and $r_{total} = \sum_{i=0}^{l} r_i = \sum_{i=0}^{l} r\alpha^i$. For the checkpointing configurations which have fewer levels, the errors are recovered based on the checkpoint type similar to Kyma, and if some original Kyma checkpoint levels are omitted then their errors are recovered by the closest checkpoint still constructed: e.g. for the UndoLog+RL1h the error rates of the undo-log levels is the same as Kyma's, and the redo-log level's error rate is the sum of the error rates of the redo-log levels of Kyma.

Figure 44 presents the efficiency of the system for Kyma and the other checkpointing configurations for increasing error rates (the α parameter is 0.5): "1x" corresponds to the availability of a single processor system, and in the following experiments, "2x" for example means that the error rates of all levels have been multiplied by two. This experiment can also be interpreted as the efficiency of an "x" processors system that can create checkpoints and recover at the same latencies as Kyma. Kyma delivers the highest efficiency across all other checkpoint configurations, providing 99.99% efficiency up to 100x error rates, while it can still assist the system achieve availability higher than 96% even when the error-rates increase by 100,000 times. Such high error rates correspond to an error every approximately 15 minutes, which is close to the expected error rates that future exascale systems are going to suffer from [81, 84]. The UndoLog+RL1h and RedoLog configurations can support similar availability is Kyma at low error rates, but for the case of extreme error rates their availability would be only $\sim$76%. The reason for this decrease is that both configurations lack a number of checkpoints that Kyma creates. The UndoLog+RL1h creates frequent undo-log checkpoints which enables the system to recover quickly from frequent errors, especially since the higher error rates are recovered using the higher-level checkpoints, the undo-logs. As the error rates increase though, a higher number of errors has to be recovered using the redo-log checkpoint, which is created every hour, resulting in more time spent recovering the system, and reducing the efficiency of the system. Similarly, for the RedoLog configuration, at high error rates, an increasing number of errors has to be recovered using the less frequently (1 second) checkpoints instead of the frequent undo-log checkpoints. Finally, creating checkpoints infrequently, e.g. every 1 hour, is the worst strategy for large systems, where the system will not be making practically forward process: the error frequency is higher than then checkpointing frequency, and after an error occurs it is highly possible that a second one will happen during recovery, not allowing effectively the system to recover.

Figure 45 presents the efficiency of the system for the different checkpointing policies for the original error rate (Error 1x) and error rates 10,000 times higher (Error 10,000x), for increasing error recovery costs. In this experiment the recovery time of each checkpointing level is multiplied by the same factor, and the experiment approximates the effect that checkpoint storage solutions with latencies higher than PCM would have to the efficiency of the system. For simplicity, the

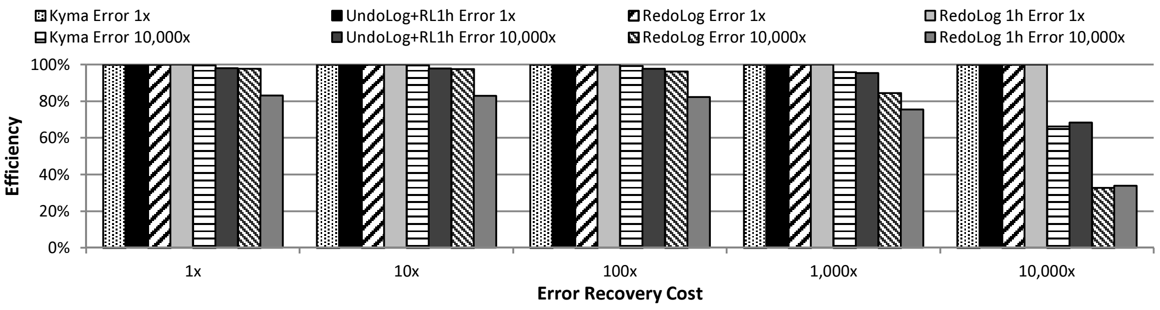


Figure 45: Achieved efficiency of a system using Kyma, compared to other checkpointing approaches, for different error rates and increasing error recovery cost.

checkpoints are assumed to be constructed in parallel with the application execution and cause the same performance overheads as the baseline. Increasing recovery costs have little effect at low error rates, since recoveries are infrequent. At high error rates though, the impact of recovery time is significant, limiting the efficiency of the system, even for Kyma, to $\sim$65%. Actually, at such point Kyma has worse availability than UndoLog+1RL, because the cost of recovery of the latest redo-log is higher than the 1 hour one, and recovery time dominates re-execution during recovery.

Figure 46 presents the improvement of efficiency that Kyma can support compared to the UndoLog+RL1h checkpoint configuration. This experiment studies the effect that varying the distribution of errors across the checkpointing levels, by modifying the α parameter, has on the availability of the system. When $\alpha = 1$ means that the error rates are equal across all checkpointing levels, while $\alpha = 0.1$ implies that the error rate of the current level is 10% the error rate of the previous higher one (the highest-level checkpoints are the undo-log). When the errors are concentrated on the higher levels of the checkpoint hierarchy (the undo-log checkpoints), then both Kyma and UndoLog+RL1h have similar availability because both can recover the majority of the errors using the frequent undo-log checkpoints[6]. As the value of the α parameter increases, the efficiency of Kyma improves over the UndoLog+RL1h, especially at higher error rates because more errors have to be recovered from lower level redo-log checkpoints, and Kyma can benefit from the frequent 1sec and 1 minute checkpoints it maintains, while UndoLog+RL1h has to resort to the 1 hour checkpoint, increasing the total recovery time.

[6]There is only a single experiment where UndoLog+RL1h performs better the Kyma by 0.005%.

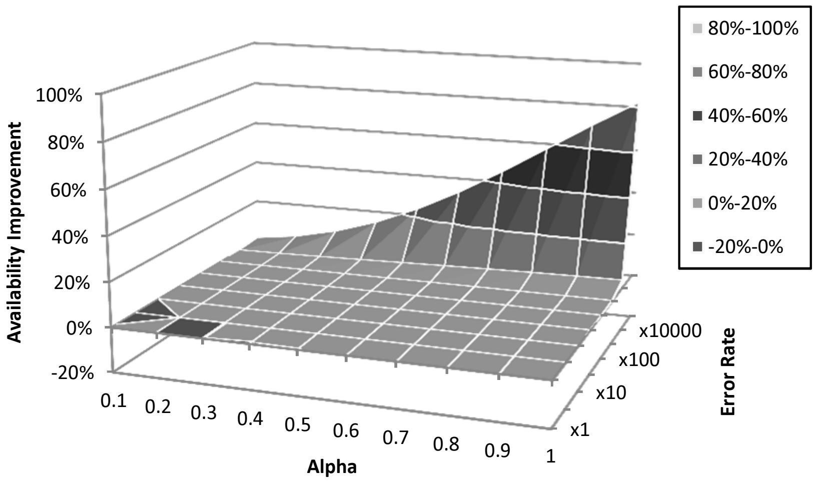


Figure 46: Improvement of efficiency of Kyma over UndoLog+RL1h for different distributions of errors across checkpoint levels, and different error rates.

Finally, it was investigated the effect that the time δ of delayed cache flushing has to the efficiency of the system. Undo-log checkpoints are so frequent, that the additional re-execution time that the delayed undo-log construction will incur is negligible, even at high error rates.

4.5.4 Performance Comparison With Other Checkpointing Techniques

To demonstrate the benefits of Kyma, in the evaluation Kyma is compared against a number of possible alternative hardware and software checkpointing techniques. Some of these alternatives correspond to state-of-the-art implementations and some represent partial implementations of Kyma-like enhancements for the purpose of isolating the contributions of each enhancement. All these mechanisms use hardware undo-log similar to ReVive [68] and SafetyNet [94], and they differ in how they create redo-logs:

Software Page Based (SW-Page): This implementation uses a software thread for redo-logging in parallel with the program's execution. It leverages existing dirty-bit information embedded in page-tables to identify modified pages of a given RLI and then copies those pages to PCM. To prevent the application from modifying data that have not been checkpointed yet, it write-protects such pages; when the application attempts a write, an exception handler immediately copies the page to PCM and resumes the application's execution.

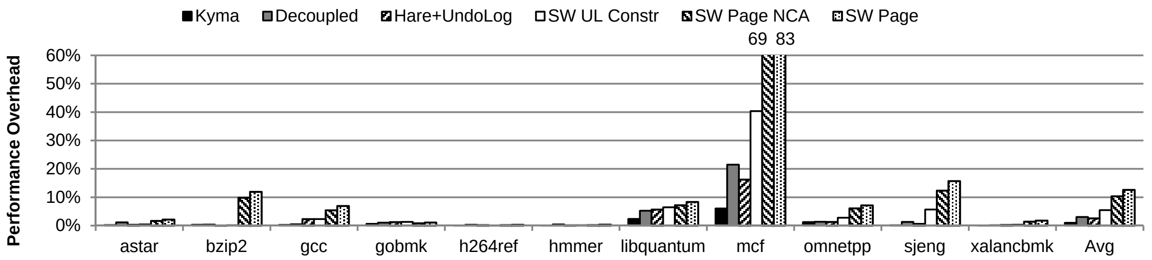
Software Page Based No Cache Allocate (SW-Page-NCA): This implementation is similar to the previous one with the only exception that data to be read or written for creating the redo-log checkpoint are not allocated in the caches, but are being copied with the help of streaming buffers. This improves cache performance because redo-log data has little or no temporal locality.

Software Block Based (SW-Block): This approach is practically a software implementation of the *decoupled engine* (which will be described), but uses a software thread instead of a hardware redo-logging engine to 1) read undo-log meta-data to construct the list of blocks to copy to the redo-log and 2) copy these blocks to PCM. Just like the page-based schemes that were described previously, it uses memory protection to prevent the corruption of the data that is yet to be copied, and just like SW-Page-NCA, cache pollution with low-locality data is avoided by using stream buffers and bypassing the caches for such data.

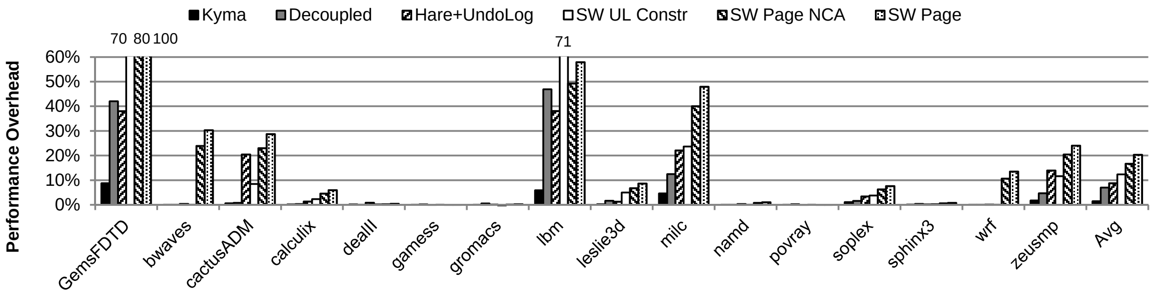
Hare and Undo-Log (HARE+Ulog): This approach leverages the HARE (Section 3.2) hardware mechanism which establishes and consolidates redo-log checkpoints.

Decoupled Engine (Dec-Eng): The *decoupled engine* is similar in structure to the Kyma engine, but the undo and redo-logging are performed independently. The decoupled engine saves undo-log data and meta-data into logs without updating the redo-logging trie structure. The redo-logging engine reads undo-log meta-data and builds its trie, then copies data to the redo-log. This decoupled approach leverages the first observation, that undo and redo-logging save data for the same addresses, but cannot benefit from the second one, that undo-log data copying can help redo-log data copying. It also cannot benefit from the trie-based secondary filtering in undo-logs. Because the trie structure needs no data pointers prior to active redo-log copying, the decoupled engine does not need a cache for the L5 nodes of the trie. Instead, the trie TLB entries are used as 64 bit-masks to mark modified blocks in that page. This reduces the cost of the engine – its on-chip state is only 22KB.

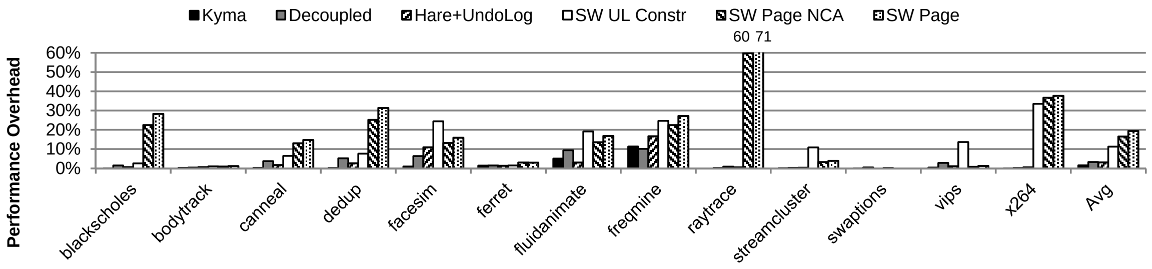
In the performance evaluation of the described software-based checkpointing techniques for multi-threaded workloads, checkpointing threads always have higher priority than application threads so one of the application threads is suspended when the checkpointing thread is active in case of no "free" cores.



(a) SPEC INT



(b) SPEC FP



(c) PARSEC

Figure 47: Performance overhead comparison of Kyma with alternative checkpointing techniques, with undo-log interval of 0.01 sec and redo-log interval of 1 sec.

Figure 47 compares the performance overheads of the five alternative techniques described. All schemes create undo-log checkpoints every 10ms and redo-log checkpoints every 1 second. Kyma outperforms all alternative techniques across all benchmarks, reducing overheads by one order of magnitude in some cases. The only exceptions are fluidanimate and freqmine which have an additional $\sim$2% overhead compared to the decoupled engine, but still lower than the other techniques.

To understand why Kyma has an advantage over other techniques, the comparison from Figure 47 is used to identify the causes of performance overhead. Software techniques compete with the application for space in the shared caches. This competition for cache space can be quantified by comparing the SW-Page and SW-Page-NCA techniques. Examples of applications where cache space contention has a significant impact are GemsFDTD, mcf and raytrace, whose overhead reduces by $>10\%$. However other applications and the averages indicate the elimination of cache contention is not the only main reason for Kyma's good performance.

Another source of overhead is contention for memory bus bandwidth. The first step to reduce the bandwidth consumed for checkpointing is to use finer memory tracking granularities and eliminate unnecessarily copied data. SW-Block tries to do this in software, providing a significant benefit in milc and mcf – in these applications page-based redo-logging copies twice as much data as block-based SW-Block does. However SW-Block achieves its finer granularity at a cost of doing much more work to construct its set of data to be copied, which results in significant new overheads in benchmarks like lbm, facesim[7], etc.

After eliminating cache contention and reducing bus bandwidth contention by using finer tracking granularities, the next step to improve performance of redo-logging is to use specialized hardware. In my experimentation I observed that more than 50% of the overhead of software implementations comes from pausing the application's execution in order to serve a page-fault caused when the application tries to modify data which have not been checkpointed yet. Hardware techniques do not suffer from such pauses, because they handle writes to data that have not been checkpointed yet internally: named "collisions" for the case of HARE, and "synergistic" copies for Kyma.

Note that Decoupled and HARE+UndoLog have similar overheads because both need to copy the same amount of data in order to establish a redo-log checkpoint. The cases where the Decoupled technique has higher overheads than HARE+UndoLog, such as GemsFDTD, lbm or mcf, are caused by the increased number of undo-log meta-data that the Decoupled's engine has to read in order to construct the redo-log meta-data. Note that the Decoupled engine does not eliminate any duplicate undo-log entries, resulting in reading more undo-log meta-data than necessary. The higher overheads of HARE+UndoLog compared to Decoupled engine are caused by the caching of HARE's memory modification tracking meta-data in the L2 cache and the additional cost of sorting the collision list. The increased PCM write latency, compared to DRAM, prolongs HARE's redo-log construction time, which results in more collisions, and higher sort collision time.

[7]This cost cannot be hidden for the case of the PARSEC benchmarks because the checkpointing thread preempts the application's threads, while for the SPEC benchmarks the checkpointing thread runs on one of the idle cores

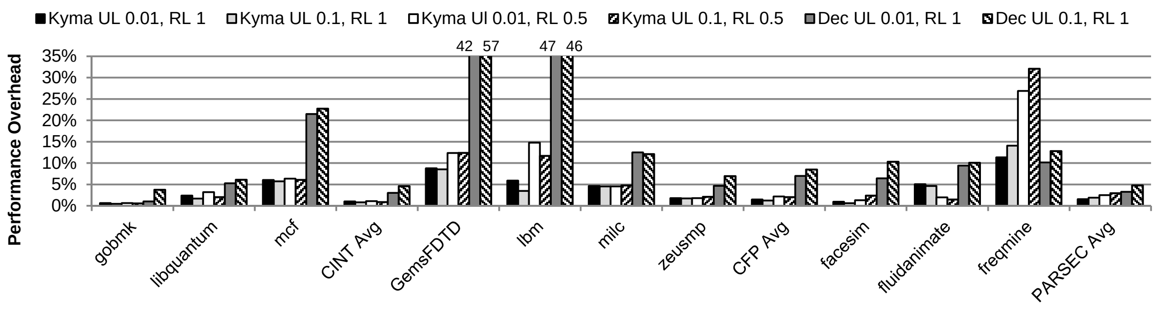


Figure 48: Performance overhead comparison of Kyma for different undo-log (UL) and redo-log (RL) checkpointing intervals and comparison with the decoupled engine (Dec), for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

Kyma outperforms both Decoupled and HARE+UndoLog techniques because it further reduces the bandwidth requirements by 1) eliminating the reads of undo-log meta-data that Decoupled does 2) removing duplicate entries in the undo-logs via secondary trie-based filtering, and 3) avoiding many reads and writes during redo-log copying, by leveraging copying activity of undo-logging. As shown in Figure 47, these benefits can be very significant, especially for the applications that suffer the highest overheads in Decoupled and HARE+UndoLog, such as mcf, GemsFDTD, lbm, and milc.

4.5.5 Checkpointing Frequency Sensitivity Analysis

Another way to reduce the memory bandwidth requirements of a combined undo-log and redo-log checkpoint mechanism is to decrease checkpointing frequency of undo-logs. The intuition behind this approach is that blocks, that are checkpointed multiple times across consecutive undo-log checkpoints, are going to be checkpointed only once, and the memory bandwidth requirements are going to decrease. Figure 48 presents the highest overhead benchmarks along the with the averages for Kyma and the Decoupled engine (Dec) for undo-log checkpointing intervals (UL) of 0.01sec and 0.1sec, and for redo-log intervals (RL) of 0.5 and 1 second. Kyma benefits from the reduced checkpointing frequency, although only marginally. Contrary to expectations, the overhead of the Decoupled engine actually increases when undo-log checkpoints are created less often. The cause for this behavior is the lack of secondary undo-log block filtering which results in no decrease, if not an increase, of the checkpointed undo-log blocks, resulting in more bandwidth consumed for checkpointing unnecessary blocks and reading the undo-log meta-data for constructing the redo-log

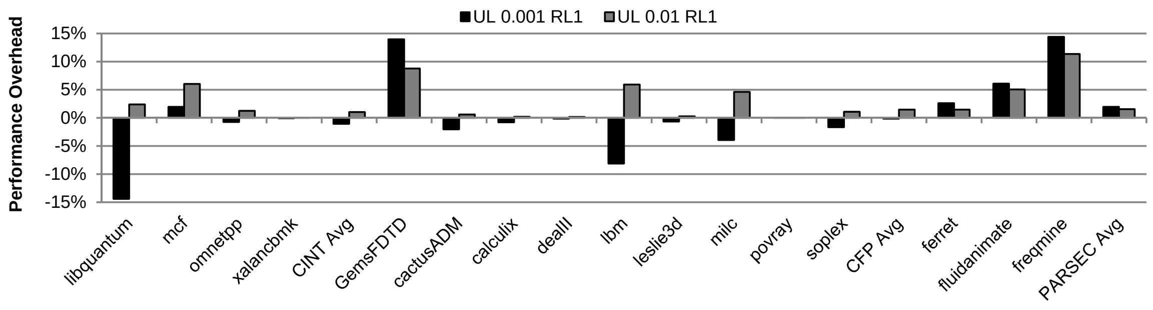


Figure 49: Performance overhead of Kyma for undo-log intervals of 0.001 and 0.01 sec and redo-log interval of 1sec , for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

meta-data.

Increasing the redo-log checkpointing frequency results, as expected, in increased overheads, which are not necessarily proportional to the increase of the checkpointing frequency. There are applications where the overheads remain practically the same, e.g. mcf, GemsFDTD and milc, and can be justified by the application's behavior and the periodicity by which it modifies all of its memory. For example, if the application iterates over its data-set every 1 second, checkpointing every 0.5 seconds will checkpoint half of the application's data-set, and the overall memory checkpointed is going to be the same regardless of whether Kyma creates redo-log checkpoints every 0.5 or 1 second. If the application's period though is 0.5 seconds, then checkpointing every 0.5 seconds will result in copying double the amount of memory that 1 second redo-log checkpointing would.

I also performed experiments where I increased the undo-log checkpointing frequency to cover the case of I/O intensive applications, where the undo-log checkpoint has to be terminated before the I/O operations is performed. Figure 49 presents the performance overhead of Kyma for undo-log checkpointing interval of 0.001sec (0.001UL) and 0.01sec. The expectation is that when the system is creating undo-log checkpoints more frequently, the performance overhead will increase because of the higher number of checkpointed blocks and cache-flushes. This observation explains the increased overheads for the cases of GemsFDTD, fluidanimate and freqmine. There are some benchmarks though where the overheads decrease (mcf), or we can even observe speed-ups, e.g. libquantum and lbm. This behavior can be explained by the fact that, when the caches are flushed frequently enough, they result in a decrease of the overall number of write-backs. As a result, the write-back latency is eliminated from the critical path for a L1 cache miss when it is accessing L2, because it does not have to wait for a dirty block to be written back to memory first, before it gets replaced. At the same time the overall consumed off-chip bandwidth remains approximately the same, because a significant number of L2 cache write-backs have been flushed to main memory ahead of time.

To gain better insight into the benefits of synergistic copying and the adaptive redo-log construction policy, Figure 50 shows for Kyma's static redo-log policy (Kyma S), Kyma and the Decoupled engine (Dec), for undo-log intervals of 0.01 and 0.1 sec and a redo-log interval of 1 second, the breakdown of the average number of memory accesses per redo-log interval into: accesses generated by the application (App), necessary undo-log writes (UL-Nec), unnecessary undo-log writes (UL-Unnec), redo-log construction reads and writes (RL), and engine accesses (EA) (such as the reads of the undo-log meta-data for Dec-Eng and the misses from the engine's caching structures in both Dec-Eng and Kyma). All numbers are normalized to application accesses (App) of the slowest configuration (Decoupled engine with 100ms undo-logging). The experiments where the number of application memory accesses are higher than 100% can be attributed to the fact that these applications experience lower overheads, have higher IPC and perform more memory accesses than the base line.

We observe that, by decreasing the undo-log frequency, the number of unnecessary checkpointed undo-log blocks increases for the Decoupled engine. The reason is that long undo-log checkpointing intervals increase the probability of a block getting replaced for the L2 cache after it has been checkpointed. This removes the *checkpoint* bit information, resulting in the same block being checkpointed again when it is written again in the same undo-log interval. Kyma does not suffer from this behavior because of its secondary trie-based filtering that prevents duplicates from being saved to the undo-log.

Additional performance benefit of Kyma over the Decoupled engine (and similarly HARE+Undo-Log) comes from the synergistic undo-log and redo-log checkpoint construction. This approach eliminates almost all redo-log copying activity in lbm, and cuts such activity by 90% percent in GemsFDTD. Overall, synergistic copying eliminates 30% to 50% of redo-log copying activity on average across all applications. The increased number of synergistic copying also explains the performance benefit of Kyma's adaptive redo-log policy over the static one (Figure 43). By not

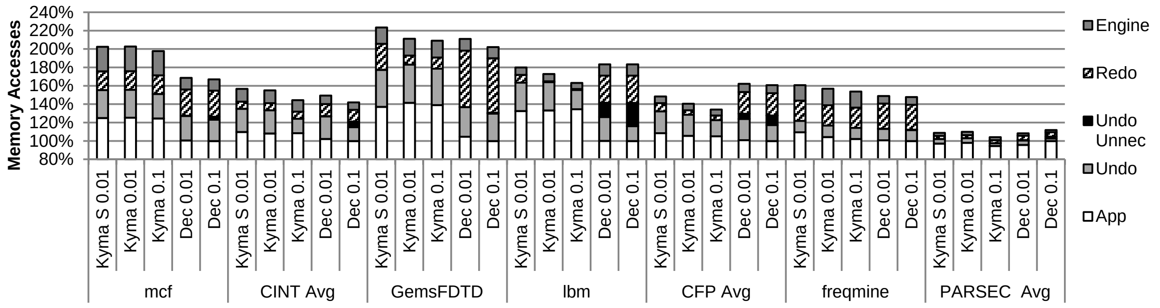


Figure 50: Break-down of the type memory accesses of the Kyma, Kyma with a static redo-log policy (Kyma S) and the Decoupled engine (Dec) for undo-log checkpointing intervals of 0.01 and 0.1 sec.

throttling the redo-log construction the static policy eliminates a number of synergistic copies and consumes more memory bandwidth, in benchmarks such as GemsFDTD and lbm.

The only benchmark where Kyma offers no performance improvement over the other techniques, e.g. Decouple, is freqmine. Freqmine has a memory access pattern such that: 1) no unnecessary undo-log entries are created (no benefit from secondary filtering), 2) there is a limited number of synergistic copying opportunities and 3) Kyma caches exhibit high miss-rates. The lack of synergistic copying means that no redo-log blocks have been synergistically checkpointed during the first half of the redo-log interval. This behavior, in conjunction with the fact that freqmine is among the applications that create the biggest checkpoints, results in the redo-log construction not finishing in time and pausing the application's execution, causing increased overheads when the static redo-log construction policy is used (Figure 43). The adaptive redo-log construction policy identifies this behavior and starts the redo-log construction early enough, resulting in overheads similar to the Decoupled engine's. This behavior also explains freqmine's increased overheads when the redo-log checkpointing frequency increases (Figure 48): the application's execution has to pause for the redo-log construction to finish.

4.5.6 Kyma Engine Hardware Cost Sensitivity Analysis

Kyma's hardware cost is minimal, yet is higher than HARE's (Section 3.6.2) and HARE could be selected as the redo-log checkpointing mechanisms despite its higher performance overheads. The structure that requires the most hardware state as part of Kyma's engine is the cache of the L5 metadata nodes (L5MD cache) whose size is 64KB. Figure 51 shows the performance overhead of the

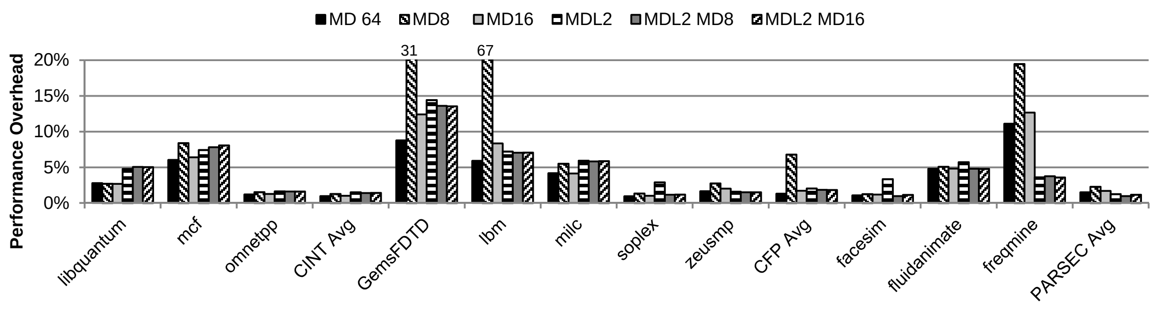


Figure 51: Performance overhead of Kyma for different L5MD cache configurations, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

worst performing applications along with the averages for Kyma for different cache configurations for undo-log interval of 0.01 seconds and redo-log of 1 second. This figure compares the performance of the original configuration which has 64KB of L5MD Cache (MD64) with one that has 8KB (MD8) and another one that has 16KB (MD16). After reducing the size of the L5MD cache, the next step is to remove it completely and have the redo-log trie meta-data be cached together with the application's data in the shared L2 cache (MDL2). To reduce the competition between the application's data and the redo-log meta-data I am also trying configurations where the L5MD cache is added again, but any misses from this cache, instead of going directly to memory, are going to the L2 cache, and I am reporting numbers for L5MD caches of 8KB (MDL2 MD8) and 16KB (MDL2 MD16).

Reducing the size of the L5MD cache to 8KB, hurts significantly the performance of the worst behaving applications such as GemsFDTD (31%), lbm (67%) and freqmine (19%) and proves not to be sufficiently big to hold all the redo-log meta-data. Increasing the size to 16KB reduces the overhead which is 2%-5% higher than the original configuration. Removing the L5MD cache completely, and having the redo-log meta-data share the L2 with the application's data, results in higher overheads, especially for cache sensitive applications such as mcf, GemsFDTD and lbm, by 2%-5%. The only exception is freqmine where the overheads actually drop. This proves that freqmine is not a L2 cache sensitive application, and that the original 64KB L5MD cache was not sufficiently big. As a result the L5MD cache was missing frequently, competing with the application for off-chip bandwidth. Finally, introducing again the L5MD cache does not significantly improve performance, because the number of compulsory misses in L2 caused by the redo-log meta-data remains the same.

Overall, the hardware cost of Kyma's engine can be reduced by removing the L5MD cache, assuming that the system can tolerate the additional performance overhead. Otherwise, if performance is critical, the current configuration proves sufficient.

4.5.7 3D-Stacked Memory Performance Evaluation

Another design option is to exploit 3D-stacked memory when it gets introduced in future microprocessors. Work in 3D-architecture [44] has demonstrated that several gigabytes of memory can be available on-chip. In the presence of such significant amount of on-chip memory, the Kyma technique could benefit and further reduce the consumed off-chip bandwidth by storing the constructed undo-logs and caching the redo-log meta-data on-chip. The sizes of 0.01sec undo log checkpoints is less than 10MB and 0.001sec ones is less than 1MB across all applications, so storing 2 of them on on-chip DRAM will have limited memory requirements. Undo-log blocks that also belong to a redo-log are still copied to PCM, and flushed L2 cache blocks are written to memory. The storage of the undo-log on-chip limits the type and number of errors the system can use them to recover from, because if the on-chip memory becomes non-accessible then the system cannot use the stored undo-logs to recover from a faulty core.

Figure 52 presents the performance overhead of Kyma for the worst performing applications for undo-log intervals of 0.001 and 0.01 sec, and redo-log interval of 1 sec for the original configuration and a 3D memory configuration. I am simulating a 3D-stacked DRAM with latencies similar to a conventional DDR3-800 DRAM. In this experiment the application is not benefiting from the 3D-memory and all its accesses still go to the main memory. Thus the baseline remains the same and the results are directly comparable with the original configuration. The goal of this experiment is to demonstrate the extra overhead caused because of bandwidth consumed for storing the undo-logs and fetching the redo-log meta-data from memory.

The performance overheads for Kyma decrease when the engine exploits the on-chip 3D memory, and become $\sim$6% even at high undo-log checkpointing frequencies, e.g. GemsFDTD and freqmine. Moreover, the speedups we have observed before in some benchmarks, such as libquantum and mcf, improve further. The difference in speedups/overheads shows the cost of undo-logging and accessing the redo-log meta-data, and the remaining overheads is the minimum performance

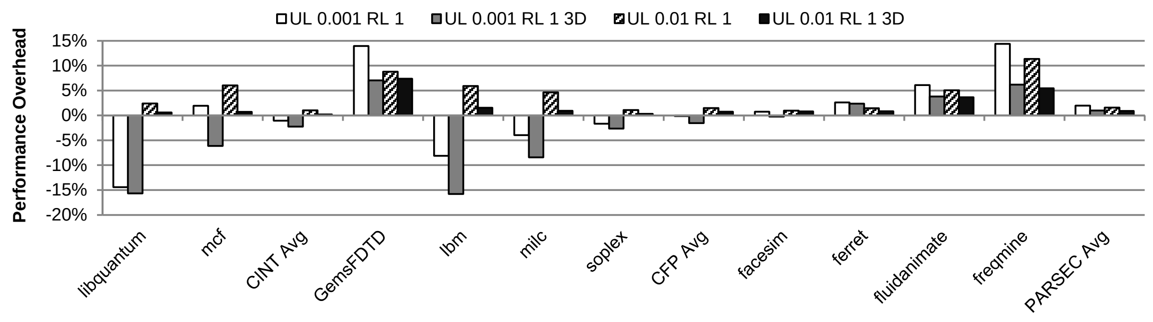


Figure 52: Performance overhead of Kyma for the original and 3D-stacked (3D) memory configuration, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

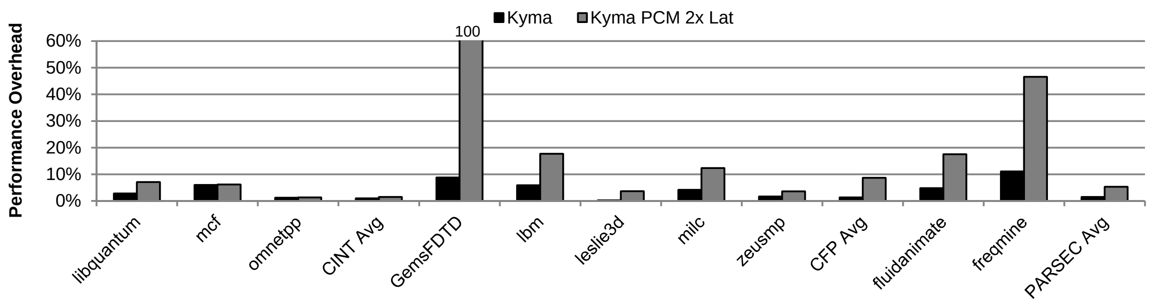


Figure 53: PCM latency sensitivity analysis of Kyma's performance overhead, for the SPEC INT (CINT), SPEC FP (CFP) and PARSEC benchmarks.

cost that Kyma can achieve for constructing redo-log checkpoints which is $\sim$1% on average across all applications.

4.5.8 PCM Latency Sensitivity Analysis

Another parameter of the system that I tested is the sensitivity of the Kyma engine to the increase of the write latency of PCM. PCM is not a standardized yet technology, and it is possible that because of fabrication/implementation limitations initial iterations of the technology might have higher latency characteristics than currently estimated. For this reason I am also conducting experiments where the write latency of PCM was doubled. Figure 53 presents the performance overheads of Kyma for the worst performing applications along with the averages for the different benchmark suites when the latency of PCM doubles (PCM 2x Lat) and compares it to the original configuration.

We can observe that there are applications which suffer minimal if not zero performance overheads such as omnetpp and zeusmp, there are other applications that suffer overheads around 5%

to 10% (libquantum, lbm, milc and fluidanimate) and finally there are applications which suffer extremely high overheads (GemsFDTD and freqmine). The first category of benchmarks are applications which create relatively small checkpoints, and the Kyma engine can tolerate gracefully the increased PCM latency. The second category of benchmarks are ones that have high checkpointing activity, and the increased PCM latency causes the queues of the Kyma engine to become full more often, resulting in execution stalls because a core needs to checkpoint a block but the engine is not available. Finally, the third category of applications creates the biggest checkpoints across all benchmarks. The increased PCM latency delays the redo-log construction process to the extent that it does not finish in time and the application's execution has to pause until the construction finishes. Overall, Kyma's performance does depend, as expected, on the latency of the non-volatile memory technology used, and slower memory technologies can be used only if the targeted applications do not have high checkpointing requirements.

4.5.9 PCM Memory Requirements and Power Considerations

To estimate PCM space requirements of checkpointing I profiled the SPEC and PARSEC applications using PIN [45] (I used the reference and native inputs respectively and the applications ran to completion) and estimated the average redo-log checkpoint size for redo-log intervals of 1-second and 1-minute. The benchmarks used in the evaluation do not run long enough to estimate the size of 1-hour checkpoints, so it is assumed that a 1-hour checkpoints is as large as the allocated address space of the application. The results are shown in (Figure 54). I find that 4GB of PCM would be sufficient on average to store all redo-log checkpoints (1s, 1min, 1h). Kyma can copy the old one-hour checkpoints to other storage media (SSD or hard-disks). The experiments indicate that this requires a maximum bandwidth of 35MB/sec, which can be provided by existing storage solutions. Note that 4GB PCM requirement is not fundamental – if needed Kyma could adopt more aggressive consolidation policies, such as keeping just a single consolidated checkpoint, but that would affect the availability of the system as described in Section 4.5.3. I also estimated the lifetime of a 4GB PCM with Kyma – 17.7 years in the worst case. Kyma writes to PCM only one checkpoint per second, consolidations update only meta-data, and writes can easily be distributed over PCM to avoid waring out a single block.

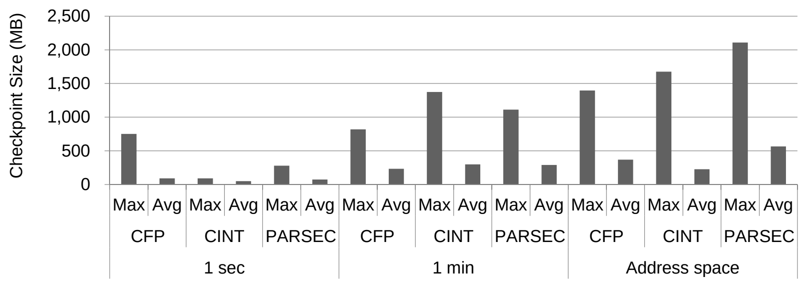


Figure 54: Maximum and average memory requirements in MB of the one second, one minute checkpoints and the address-space of the application.

Finally, the power overhead of Kyma to the system is estimated, using CACTI 5.3 [99] to determine the dynamic power consumed by the caches/queues of the Kyma engine and the DRAM memory of the system. PCM is expected to have higher write power requirements than DRAM and we used the power estimates from Lee *et al.* [40]. These are expected to be the dominant sources of additional power consumption caused by Kyma. The results show that the majority of the power overhead is caused by the additional memory accesses for creating the checkpoints, and that the power requirements of the system increase by 1-2 Watts on average.

4.6 Summary

Reliability is going to become an increasing problem in the future, and recovery mechanisms will play a critical role in the availability of future systems. The two main approaches to checkpointing each have important limitations – redo-logs can recover from catastrophic faults but have poor recovery latencies for the most common errors, while undo-logs offer quick recovery from common errors but cannot recover from catastrophic faults.

This chapter presented Kyma, a mechanism that synergistically combines the two approaches. It creates undo-log and redo-log checkpoints together, using their inherent relationships to minimize the performance overhead, bandwidth consumption, and memory space needed for this purpose. The experiments indicate that Kyma incurs minimal overheads ($\sim 2\%$) on average and 11% worst-case across all benchmarks, while supporting efficient recovery from catastrophic and non-catastrophic errors that have a wide range of detection latencies (from a few milliseconds to one hour). Kyma also can be used to support a high availability system providing 99.999% availability for today's systems, while allow future systems to have more than 95% efficiency.

CHAPTER V

CONCLUSIONS

Checkpointing is a technique that has uses in multiple areas, e.g. in debugging and in reliability. This dissertation examined the common functionality requirements and the causes of performance overheads of memory checkpointing when it is used in bidirectional debugging and for improving the reliability of the system.

Common characteristic of both bidirectional debugging and reliability is that both techniques can benefit from frequent checkpointing. Frequent checkpoints allow bidirectional debugging to reduce reverse-execution latency, and improve the associated interactivity of the debugger. The same characteristic applies for system reliability as well, where frequent checkpoints can improve the availability of the system by limiting the time a system spends recovering from an error. Another technique that can benefit both bidirectional debugging and reliability is checkpoint consolidation. Checkpoint consolidation can efficiently manage the constructed checkpoints and reduce the overall memory requirements of the system. Consolidation can distribute the checkpoints over time, allowing reverse-execution to appear interactive to the user and reverse-execution to have latencies similar to forward execution. Consolidation can also be used in reliability in order to create multiple levels of checkpoints that can provenly improve the availability of the system.

I proposed three hardware accelerators, HARE and Euripus for bidirectional debugging and Kyma for reliability, all designed to construct checkpoints frequently at a minimal performance overhead and enable checkpoint consolidation for efficient memory management. HARE has the lowest hardware requirements and meets the interactivity goals of bidirectional debugging, but needs frequent software intervention in order to implement its functionality. Euripus provides improved functionality compared to HARE, at a somewhat higher hardware cost. Euripus can construct two different types of checkpoints (undo and redo-log) simultaneously, by exploiting synergies that develop when both types of checkpoints are being created. The two types of checkpoints allow Euripus to provide improved reverse-execution latency at minimal additional performance and memory overheads. Finally, Kyma is an accelerator which shares the same architecture as Euripus, but its functionality is adjusted to the requirements of reliability. Similar to Euripus, Kyma can construct two types of checkpoints (where each type serves the recovery of different types of errors), and it consolidates checkpoints in order to recover multi-level checkpointing. The Euripus and the Kyma hardware accelerators can become part of a unified accelerator that can serve the requirements of both bidirectional debugging and reliability.

Experiments show that the performance overheads of all three accelerators are directly comparable with other hardware checkpointing techniques, which do not provide checkpoint consolidation functionality, and significantly lower than software based solutions. The memory requirements of HARE and Euripus are orders of magnitude lower compared to techniques which do not support checkpoint consolidation, while consolidation allows Kyma to maintain multiple checkpoints at minimal additional memory cost.

In the past, performance improvement was a result of the changes in the processor's microarchitecture and innovations in the process technology that accelerated single-threaded software performance. Today we are focusing on parallel architectures and on the problem of efficiently dividing the work and executing previously single-threaded applications across multiple cores on the same chip. In the future, performance and power considerations will drive architects to design and implement specific hardware accelerators for performance critical tasks. I hope that the three accelerators that I studied in this dissertation will serve as a motivation to further improve checkpointing, which plays a critical role in both debugging and reliability. I also hope that this work will inspire others to study problems which can benefit from the presence of dedicated hardware solutions.

APPENDIX A

MULTI-LEVEL CHECKPOINTING RELIABILITY MODEL

For the purpose of estimating the availability of a system that would be using the Kyma checkpointing mechanism (Section 4.5.3) I am using an analytical model that, based on the estimated error rates and recovery costs for every checkpoint, computes the expected execution time of the system and its efficiency/availability. The model used is an extension of availability model for multi-level checkpointing system proposed by Moody *et al.* [52]. This chapter presents all final mathematical results necessary for implementing the analytical model, and all necessary extensions for adapting the original to the characteristics of the Kyma checkpointing technique. For the complete mathematical proofs the reader is encouraged to refer to the work of Moody *et al.* [52].

A.1 Analytical Reliability Model

The analytical model is based on a Markov Model (MM) which describes the system. The model (Figure 55) consists of computation (c), undo-log (UL) and redo-log (RL) recovery states. The transitions between computation to recovery states represent the probability that an error happens during computation. Transitions from a recovery to a computation state represent the successful recovery of the system, and from a recovery to another recovery state represent the probability of an error occurring during the recovery process or a recovery level not being the appropriate one to recover from a specific type of error. Note that Figure 55 shows the recovery states only for the last compute state, and that those recovery states are replicated for every compute state.

The assumptions of the model are: First, that failures are independent: an error during the current execution of a program does not affect the probability of another error happening during the execution of the same program or during a future execution. Second, errors are taken at fixed intervals during the execution of the application, events like I/O might force a checkpoint to terminate early, but I am expecting the checkpoints at a given level have an average frequency that corresponds to the steady state of the system. Third, when an error happens the recovery mechanism

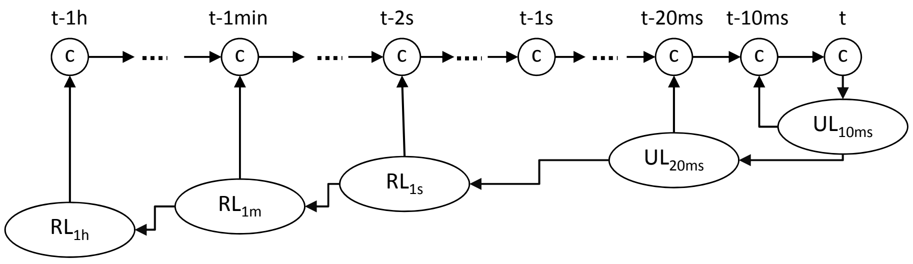


Figure 55: Markov Queue model of a multi-level checkpointing system.

rolls back the system to the latest checkpoint that can recover from the error, and not to the one that will have the lower recovery cost. The goal of the model is to minimize the re-execution time of the system and not the system state recovery time. Finally, the model assumes that there are infinite spare resources and there is zero repair cost.

The main differences between the original Markov Model for multi-level checkpointing proposed by Moody *et al.* [52] and Kyma's multi-level checkpointing are: First, Kyma apart from redo-log it also creates undo-log checkpoints. The main difference between the recovery of undo and redo-log checkpoints is, as described in Section 4.4.1, when using an undo-logs for recovery if an error happens in the current compute interval the system cannot recover to the beginning of the last interval, but instead it recovers to the previous interval, because the last undo-log checkpoint has not been constructed yet. Second, the top-level redo-log checkpoint (RL_{1s}) is also not established at the beginning of the interval, but is constructed in a lazy fashion, so the system cannot use it to recover the system and instead has to use the previous 1second level checkpoint. Third, at any point in time the checkpoints at all levels are present. In Kyma, the 1 minute and 1 hour are constructed through consolidation in parallel with the execution and the system does not have to stop execution in order to create a specific type of checkpoint. For simplicity it is assumed that consolidation has zero latency. Finally, in Kyma all types of checkpoints are constructed in parallel with the execution and it does not have to stop the application's execution. As a result the checkpoint construction cost is zero, and instead the overhead parameter α has been introduced, which corresponds to the average performance overhead of Kyma as estimated in its evaluation (Section 4.5.2).

In order to estimate the overall execution time in the steady state of the system, Moody *et al.* [52] solve the model analytically. By exploiting the nested behavior of the Markov model, they create a recursive representation of the model (Figure 56) that consists of basic blocks Y, X and Z. In the

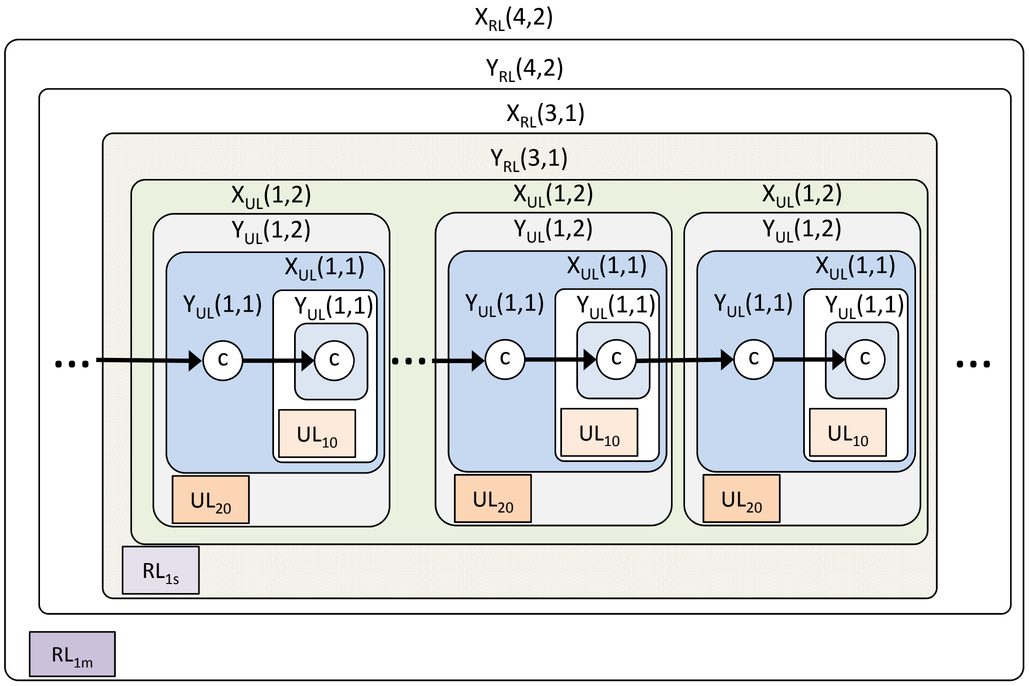


Figure 56: Recursive representation of the Markov Model.

extension of the original model I am using only the Y and X states which correspond to computation and recovery blocks respectively, and I differentiate them based on the type of checkpoint used for recovery: Y_{UL} and X_{UL} correspond to compute and recovery blocks where undo-logs are used for recovery and Y_{RL}, X_{RL} where redo-logs are used.

$Y_{UL}(1)$ corresponds to the basic compute state and $X_{UL}(1)$ to a recovery state that relies on the last established undo-log checkpoint for recovery. $Y_{UL}(k > 1, c)$ is a compute state which internally consists of a basic compute state $Y_{UL}(1, c)$ and a recovery node $X_{UL}(k-1, c)$, while $X_{UL}(k > 1, c)$ is a recovery state which internally has a $Y_{UL}(k > 1, c)$ state. The last two states represent the recovery where the system has to roll-back multiple undo-log checkpoints in the past. $Y_{RL}(k \geq 1, c)$ is a compute state that internally consists of multiple undo or redo-log recovery states. Finally, $X_{RL}(k \geq 1, c)$ is the redo-log recovery state and internally includes a single Y_{RL} state.

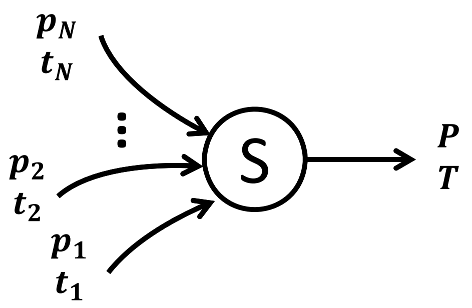


Figure 57: Transition state with multiple incoming and a single out-going edge.

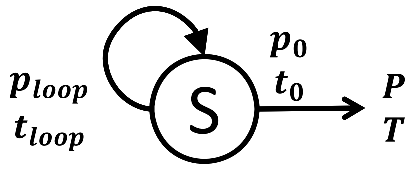


Figure 58: Transition state with a loop edge.

A.1.1 Basic Probabilities and Execution Times

Assuming that probabilities are independent and follow a Poisson distribution the probability that no error happens in a period of time $t = 0$ to $t = T$:

$$p_0(T) = e^{-\lambda T} \tag{1}$$

and the execution time is:

$$t_0(T) = T \tag{2}$$

The probability that an error happens in level k before a failure happens in any other level in the time interval $t = 0$ to $t = T$:

$$p_k(T) = \frac{\lambda_k}{\lambda}(1 - e^{-\lambda T}) \tag{3}$$

and the time the system executed before a failure happened in level k is:

$$t_k(T) = \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{\lambda T})} \tag{4}$$

where λ_k is the average failure rate at level k and $\lambda = \lambda_1 + \lambda_2 + \cdots + \lambda_L$

For every state S describing the system I am computing the probability p_{S_0} of the system experiencing no error and the respective error free execution time t_{S_0} and the vectors $\overrightarrow{p_S}$ and $\overrightarrow{t_S}$ where p_k present the probability an error happened at level k and t_k the lost execution time.

A.1.2 Basic building blocks

Multiple Edges When in an out-going state transition there are multiple incoming edges, such as in Figure 57, the output total probability P and execution time T, as proven by Moody *et al.* [52], are:

$$P = \sum_{i=1}^{N} p_i \tag{5}$$

$$T = \frac{\sum_{i=1}^{N} p_i \cdot t_i}{P} \tag{6}$$

Loop-back Edges When a state has a loop back edge, with probability p_{loop} and execution time t_{loop} and departing probability p_0 and time t_0 then the total probability P and execution time T are:

$$P = \begin{cases} \frac{p_0}{1-p_{loop}} & \text{for } p_{loop} < 1 \\ 0 & \text{for } p_{loop} = 1 \end{cases} \tag{7}$$

$$T = t_0 + \frac{p_{loop}}{1 - p_{loop}} \cdot t_{loop} \tag{8}$$

A.1.3 The $Y_{UL,RL}(k = 1, c = 1)$ State

The state $Y_{UL,RL}(k = 1, c = 1)$ corresponds to the basic compute state c of the Markov Model, and can be recovered from either an undo or a redo-log checkpoint. Assuming that the checkpointing execution overhead is α, then the probability that no error happens is:

$$p_{Y_0} = p_0(\alpha t) = e^{-\lambda \cdot \alpha t} \tag{9}$$

and the execution time if not error happens is:

$$t_{Y_0} = t_0(\alpha t) = \alpha t \tag{10}$$

The probability that an error happens at level k is:

$$p_{Y_i} = p_i(\alpha t) = \frac{\lambda_k}{\lambda}(1 - e^{-\lambda \cdot \alpha t}) \tag{11}$$

and the respective execution time is:

$$t_{Y_i} = t_i(\alpha t) = \frac{1 - (\lambda \cdot \alpha t + 1) \cdot e^{-\lambda \alpha t}}{\lambda \cdot (1 - e^{\lambda \alpha t})} \tag{12}$$

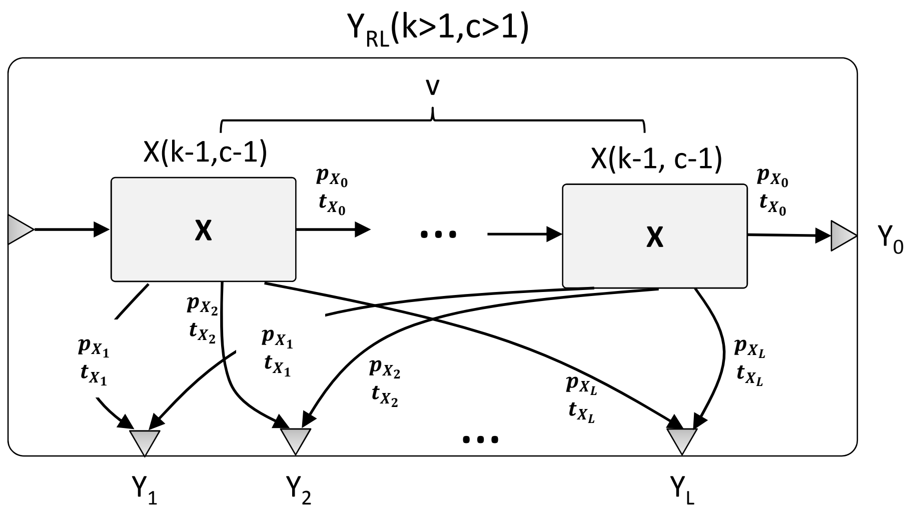


Figure 59: The $Y_{RL}(k)$ computation state for $k, c > 1$.

A.1.4 The $Y_{RL}(k > 1, c > 1)$ State

The Y_{RL} computation state for the redo-log, consists internally of v X recovery states as show in Figure 59. This state represents for example the computation that the system is going to perform in one minute. The one minute interval internally consists of 60 one second intervals which are encapsulated in their respective X recovery states.

The probability p_{Y_0} that no error happens is the probability that no X state suffers an error, which is the product of all the p_{X_0} probabilities:

$$p_{Y_0} = (p_{X_0})^v \tag{13}$$

and the executed time t_{Y_0} is the sum of executed time of all X states:

$$t_{Y_0} = v \cdot t_{X_0} \tag{14}$$

The probability that an error at level k happens, based on equation 5, is the sum of the probabilities an X states suffers from a k level error. The probability to reach the ith X state is $p = (p_{X_0})^{i-1}$, so:

$$\begin{aligned} p_{Y_i} &= p_{X_i} + p_{X_0} \cdot p_{X_i} + (p_{X_0})^2 \cdot p_{X_i} + \ldots + (p_{X_0})^{v-1} \cdot p_{X_i} \\ &= p_{X_i} \cdot (1 + p_{X_0} + (p_{X_0})^2 + \ldots + (p_{X_0})^{v-1}) \end{aligned}$$

Given that $\sum_{i=0}^{N} x^i = \frac{1-x^{N+1}}{1-x}$

$$p_{Yi} = \frac{1 - (p_{X_0})^v}{1 - p_{X_0}} \cdot p_{X_i} \tag{15}$$

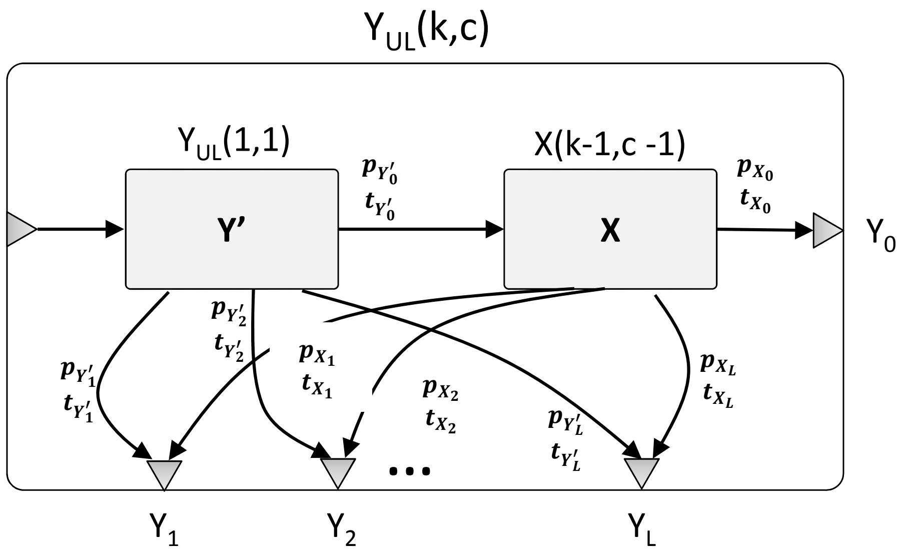


Figure 60: The $Y_{UL}(k)$ computation block for $k, c > 1$.

The total lost computation time t_{Y_i} is :

$$t_{Y_i} = \frac{PenaltyTime}{p_{Y_i}}$$

Where $PenaltyTime$ is the sum of the penalty time that each X state has suffered because of an error at level k. The lost execution time until we have reached the X_i state is the total execution time until state X_i ($(v-1) \cdot t_{X_0}$), plus the time lost in X_i (t_{X_i}), times the probability to reach state X_i ($(p_{X_0})^{v-1}$), times the probability to suffer an error (p_{X_i}).

$$\begin{aligned}
PenaltyTime &= p_{X_i} \cdot t_{X_i} + (p_{X_0})^1 \cdot p_{X_i} \cdot (1 \cdot t_{X_0} + t_{X_i}) + (p_{X_0})^2 \cdot p_{X_i} \cdot (2 \cdot t_{X_0} + t_{X_i}) \\
&\quad + \ldots + (p_{X_0})^{v-1} \cdot p_{X_i} \cdot ((v-1) \cdot t_{X_0} + t_{X_i}) \\
&= p_{X_i} \cdot t_{X_i}(1 + (p_{X_0})^1 + (p_{X_0})^2 + \ldots + (p_{X_0})^{v-1}) \\
&\quad + p_{X_i} \cdot t_{X_0}(1 \cdot (p_{X_0})^1 + 2 \cdot (p_{X_0})^2 + \ldots + (v-1) \cdot (p_{X_0})^{v-1}) \\
&= p_{X_i} \cdot t_{X_i} \frac{1 - p_{X_0}^v}{1 - p_{X_0}} + p_{X_i} \cdot t_{X_0} \frac{p_{X_0} - v \cdot (p_{X_0})^v + (v-1) \cdot (p_{X_0})^{v+1}}{(1 - p_{X_0})^2}
\end{aligned}$$

Given that $\sum_{i=0}^{N} i \cdot x^i = \frac{x-(N+1)\cdot x^{N+1} + N \cdot x^{N+2}}{(1-x)^2}$

A.1.5 The $Y_{UL}(k > 1, c > 1)$ State

The computation state for an interval to be recovered using an undo-log checkpoint where $c > 1$ (Figure 60) is similar to the respective redo-log state (Section A.1.4). The $Y_{UL}(k > 1, c > 1)$ state internally has a $Y_{UL}(1,1)$ state that is followed by an $X_{UL}(k-1,c-1)$. The $Y_{UL}(k>1,c>1)$ state in conjunction with the $X_{UL}(k>1,c>1)$ simulate the case where the system will have to recover using multiple undo-log checkpoints.

The $Y_{UL}(k>1,c>1)$ is similar to the $Y_{RL}(k>1,c>1)$ state, where there are only two internal states, Y' and X, and the values of p_{Y_0}, t_{Y_0}, $\overrightarrow{p_Y}$ and $\overrightarrow{t_Y}$ are derived using the approach described in Section A.1.4 and are:

$$p_{Y_0} = p_{Y'_0} \cdot p_{X_0} \tag{16}$$

$$t_{Y_0} = t_{Y'_0} + t_{X_0} \tag{17}$$

$$p_{Y_i} = p_{Y'_i} + p_{Y'_0} \cdot p_{X_0} \tag{18}$$

$$t_{Y_i} = \frac{p_{Y'_i} \cdot t_{Y'_i} + p_{Y'_0} \cdot p_{X_i}(t_{Y'_0} + t_{X_i})}{p_{Y'_i}} \tag{19}$$

A.1.6 Basic Recovery State at Level k

Similar to the base compute state (Section A.1.3) when recovering the system using a checkpoint at a given level, there is a probability that an error happens during recovery. Assuming a checkpointing restoration time r_k, then the probability that no error happens is:

$$p_{R_0} = p_0(r_k) = e^{-\lambda r_k} \tag{20}$$

and the execution time is:

$$t_{R_0} = t_0(r_k) = r_k \tag{21}$$

The probability that an error at level k happens is:

$$p_{R_i} = p_i(r_k) = \frac{\lambda_k}{\lambda}(1 - e^{-\lambda r_k}) \tag{22}$$

and the lost compute time is:

$$t_{R_i} = t_i(r_k) = \frac{1 - (\lambda r_k + 1) \cdot e^{-\lambda r_k}}{\lambda \cdot (1 - e^{\lambda r_k})} \tag{23}$$

A.1.7 The $X_{UL,RL}(k > 1, c > 1)$ State

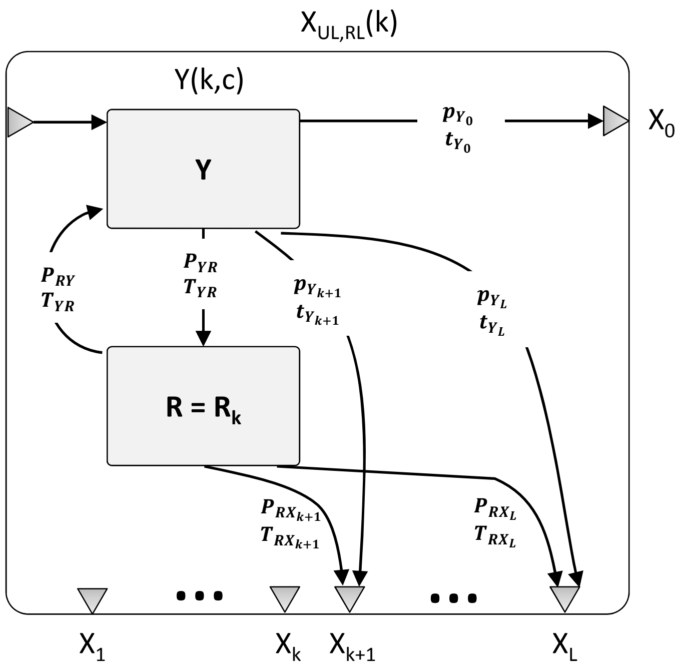


Figure 61: The X block for redo-log checkpoint recovery.

The $X_{UL,RL}(k > 1, c > 1)$ state (Figure 61) is identical to the $X(k, c)$ state originally proposed by Moody *et al.* [52]. The X state consists internally of a compute state $Y(k, c)$ and a recovery state R_k. If an error happens in the compute state Y, then the system transitions to the recovery state R_k. If an error happens during recovery and can it can be recovered from the current level then the system restarts the recovery using the current R_k recovery state, otherwise this error will be recovered from a checkpoint of at higher level.

The probability that no error happens is:

$$p_{X_0} = \begin{cases} \frac{p_{Y_0}}{1-P_{YR}\cdot P_{RY}} & \text{for } P_{YR}\cdot P_{RY} < 1 \\ 0 & \text{for } P_{YR}\cdot P_{RY} = 1 \end{cases} \tag{24}$$

and when $p_{x0} > 0$ the executed system time is:

$$t_{X_0} = t_{Y_0} + \frac{P_{YR}\cdot P_{RY}}{1 - P_{YR}\cdot P_{RY}} \cdot (T_{YR} + T_{RY}) \tag{25}$$

The probability that a non-recoverable error by this state has happened is for each level i

$$p_{X_i} = \begin{cases} 0 & \text{for } i \le i \le k \text{ or } P_{YR}\cdot P_{RY} = 1 \\ \frac{p_{Yi}+P_{YR}\cdot P_{YX_i}}{1-P_{YR}\cdot P_{RY}} & \text{for } i > k \text{ or } P_{YR}\cdot P_{RY} < 1 \end{cases} \tag{26}$$

and when $p_{Xi} > 0$ the lost computation time is:

$$t_{X_i} = \frac{p_{Y_i} \cdot t_{Y_i} + P_{YR} \cdot P_{RX_i} \cdot (T_{YR} + T_{RX_i})}{p_{Y_i} + P_{YR} \cdot P_{RX_i}} + \frac{P_{YR} \cdot P_{RY}}{1 - P_{YR} \cdot P_{RY}} \cdot (T_{YR} + T_{RY}) \qquad (27)$$

Where

$$P_{YR} = \sum_{i=1}^{k} p_{Y_i}$$

$$T_{YR} = \frac{\sum_{i=1}^{k} p_{Y_i} \cdot t_{Y_i}}{P_{YR}}$$

$$P_{RR} = \begin{cases} 0 & \text{for } k = 1 \\ \sum_{i=1}^{k-1} p_{R_i} & \text{for } 1 < k < L \\ \sum_{i=1}^{k} p_{R_i} & \text{for } k = L \end{cases}$$

$$T_{RR} = \begin{cases} \frac{\sum_{i=1}^{k-1} p_{R_i} \cdot t_{R_i}}{P_{RR}} & \text{for } 1 < k < L \\ \frac{\sum_{i=1}^{k} p_{R_i} \cdot t_{R_i}}{P_{RR}} & \text{for } k = L \end{cases}$$

$$P_{RY} = \begin{cases} \frac{p_{R_0}}{1 - P_{RR}} & \text{for } P_{RR} < 1 \\ 0 & \text{for } P_{RR} = 1 \end{cases}$$

$$T_{RY} = t_{R0} + \frac{P_{RR}}{1 - T_{RR}} \cdot T_{RR}$$

$$P_{RX_i} = \begin{cases} 0 & \text{for } 1 \le i \le k \text{ or } P_{RR} = 1 \\ \frac{p_{R_k} + p_{R(k+1)}}{1 - P_{RR}} & \text{for } i = k+1 \text{ and } P_{RR} < 1 \\ \frac{p_{R_i}}{1 - P_{RR}} & \text{for } i > k+1 \text{ and } P_{RR} < 1 \end{cases}$$

$$T_{RX_i} = \begin{cases} \frac{p_{R_k} \cdot t_{R_k} + p_{R_{k+1}} \cdot t_{R_{k+1}}}{p_{R_k} + p_{R_{k+1}}} + \frac{P_{RR}}{1 - P_{RR}} \cdot T_{RR} & \text{for } i = k+1 \text{ and } P_{RR} < 1 \\ t_{R_i} + \frac{P_{RR}}{1 - P_{RR}} \cdot T_{RR} & \text{for } i > k+1 \text{ and } P_{RR} < 1 \end{cases}$$

A.1.8 The $X_{UL,RL}(k, c = 1)$ State

The state $X_{UL,RL}(k, c = 1)$ (Figure 62) corresponds to the recovery state for the last undo and redo-log intervals. Because of Kyma's lazy approach when constructing undo and redo-log checkpoints, when an error occurs the system cannot recover to the beginning of the current undo or redo-log interval, but instead to the beginning of the previous interval. The $X_{UL,RL}(k, c = 1)$ state is similar internally to the $X_{UL,RL}(k > 1, c > 1)$ described in the previous section, with the only exception

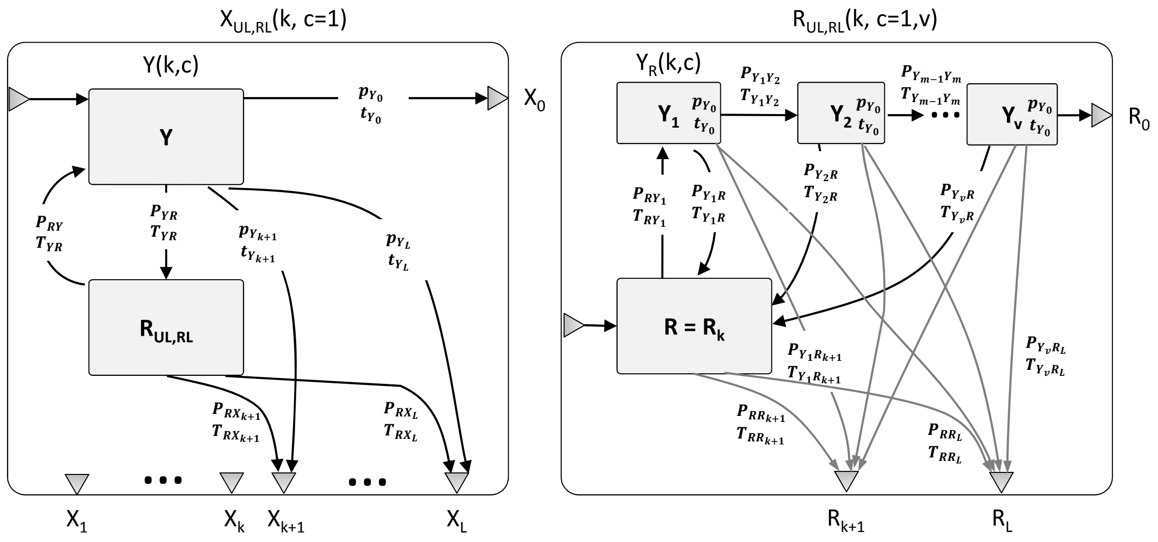


Figure 62: The $X_{UL,RL}(k, c = 1)$ state.

that the recovery state $R_{UL,RL}$ includes a number v of computation states $Y_R(k, c)$ which are necessary for restoring the system state at the beginning of the interval where the fault happened. A $Y_R(k, c)$ state is identical to the regular $Y(k, c)$ with the only exception that when an error happens it rolls back to the recovery state R_k of $R_{UL,RL}$ of the state $X_{UL,RL}$ under recovery (Figure 62).

For the generic case then we have v Y_R states, the probability that no error happens during recovery is :

$$P_{RY}(v) = p_{R_0}(v) = \begin{cases} \frac{p_{Y_0}}{1-P_{Y_{v-1}Y_v} \cdot P_{Y_vR}} & \text{for } P_{Y_{v-1}Y_v} \cdot P_{Y_vR} < 1 \\ 0 & \text{for } P_{Y_{v-1}Y_v} \cdot P_{Y_vR} = 1 \end{cases} \tag{28}$$

where:

$$P_{Y_mR} = \sum_{i=1}^{k} p_{Y_i} \text{ for } 1 \leq m \leq v$$

Since any error in Y_R of level $\leq k$ is going to be recovered from R_k, and

$$P_{Y_{v-1}Y_v} = \begin{cases} \frac{p_{Y_0}}{1-P_{Y_{v-2}Y_{v-1}} \cdot P_{Y_{v-1}R}} & \text{for } P_{Y_{v-2}Y_{v-1}} \cdot P_{Y_{v-1}R} < 1 \\ 0 & \text{for } P_{Y_{v-2}Y_{v-1}} \cdot P_{Y_{v-1}R} = 1 \end{cases}$$

$$P_{Y_1Y_2} = \begin{cases} \frac{p_{Y_0}}{1-P_{Y_1R} \cdot P_{RY_1}} & \text{for } P_{Y_1R} \cdot P_{RY_1} < 1 \\ 0 & \text{for } P_{Y_1R} \cdot P_{RY_1} = 1 \end{cases}$$

$$P_{RY_1} = \begin{cases} \frac{p_{R_0}}{1-P_{RR}} & \text{for } P_{RR} < 1 \\ 0 & \text{for } P_{RR} = 1 \end{cases}$$

$$P_{RR} = \begin{cases} 0 & \text{for } k = 1 \\ \sum_{i=1}^{k-1} p_{R_i} & \text{for } 1 < k < L \\ \sum_{i=1}^{k} p_{R_i} & \text{for } k = L \end{cases}$$

When $P_{Y_{v-1}Y_v} > 0$ the recovery time of the system is going to be:

$$t_{RY}(v) = t_{R_0}(v) = t_{R_{k_0}} + t_{y_0} + \frac{P_{Y_{v-1}Y_v} P_{Y_v R}}{1 - P_{Y_{v-1}Y_v} P_{Y_v R}} (T_{Y_{v-1}Y_v} + T_{Y_v R}) \tag{29}$$

Where:

$$T_{Y_{v-1}Y_v} = t_{y_0} + \frac{P_{Y_{v-2}Y_{v-1}} P_{Y_{v-1}R}}{1 - P_{Y_{v-2}Y_{v-1}} P_{Y_{v-1}R}} (T_{Y_{v-2}Y_{v-1}} + T_{Y_{v-1}R})$$

$$T_{Y_1Y_2} = t_{y_0} + \frac{P_{Y_1R} P_{RY_1}}{1 - P_{Y_1R} P_{RY_1}} (T_{Y_1R} + T_{RY_1})$$

$$T_{Y_mR} = \frac{\sum_{i=1}^{k} p_{Y_i} t_{y_i}}{P_{Y_mR}}$$

$$T_{RY1} = t_{R_0} + \frac{P_{RR}}{1 - P_{RR}} T_{RR}$$

$$T_{RR} = \begin{cases} \frac{\sum_{i=1}^{k-1} p_{R_i} \cdot t_{R_i}}{P_{RR}} & \text{for } 1 < k < L \\ \frac{\sum_{i=1}^{k} p_{R_i} \cdot t_{R_i}}{P_{RR}} & \text{for } k = L \end{cases}$$

The probability that an error non-recoverable by the current state happens is:

$$p_{RX_i}(v) = p_{R_i}(v) = \begin{cases} 0 & \text{for } 0 \le i \le k \text{ or } P_{YR} \cdot P_{RY} = 1 \\ \frac{P_{Y_1Ri} + P_{Y_1R} \cdot P_{RR_i}}{1 - P_{RY_1} \cdot P_{Y_1R}} + \sum_{j=2}^{v} \frac{P_{Y_jR_i}}{1 - P_{Y_{j-1}Y_j} \cdot P_{Y_jR}} & \text{for } i > k \text{ and } P_{RY_1} \cdot P_{Y_1R} \\ & \text{and } P_{Y_{j-1}Y_j} \cdot P_{Y_jR} < 1 \end{cases} \tag{30}$$

where $P_{Y_kR_i} = p_{y_k}$

$$P_{RR_i} = \begin{cases} 0 & \text{for } 1 \le i \le k \text{ or } P_{RR} = 1 \\ \frac{p_{R_k} + p_{R(k+1)}}{1 - P_{RR}} & \text{for } i = k+1 \text{ and } P_{RR} < 1 \\ \frac{p_{R_i}}{1 - P_{RR}} & \text{for } i > k+1 \text{ and } P_{RR} < 1 \end{cases}$$

and when $p_{Xi} > 0$

$$\begin{aligned} t_{RX_i}(v) = t_{R_i}(v) &= \frac{PenaltyTime}{p_{R_i}} + \text{Average Wait Time in R} \\ &= \frac{PenaltyTime}{p_{R_i}} + \frac{P_{Y_1R}P_{RY_1}}{1 - P_{Y_1R}P_{RY_1}}(T_{Y_1R} + T_{RY_1}) \end{aligned} \tag{31}$$

Where

$$\begin{aligned} PenaltyTime = &\frac{P_{Y_1R} \cdot P_{RR_i}}{1 - P_{Y_1R} \cdot P_{RR_i}} \cdot (T_{Y_1R} + T_{RX_i}) + \frac{P_{Y_1R_i}}{1 - P_{Y_1R} \cdot P_{RY_1}} \cdot T_{Y_1R_i} \\ &+ \frac{P_{Y_2R_i}}{1 - P_{Y_1Y_2} \cdot P_{Y_2R_i}} \cdot T_{Y_2R_i} + \ldots + \quad + \frac{P_{Y_vR_i}}{1 - P_{Y_{v-1}Y_v} \cdot P_{Y_vR_i}} \cdot T_{Y_vR_i} \end{aligned} \tag{32}$$

The $X_{RL}(k, c = 1)$ State For the specific case of redo-log recovery the above equations can be simplified and $R_{RL}(k, c = 1, v = 1)$ becomes identical to the $X_{UL,RL}(k > 1, c > 1)$

The $X_{UL}(k, c = 1)$ State Because of Kyma's delayed cache flush there is an additional time δ when an error cannot be recovered by the most recent undo-log checkpoint but has to be recovered from the previous one. The probability that the latest undo-log checkpoints cannot be used for recovery is:

$$p_\delta = \frac{\delta}{IntervalDuration}$$

The $X_UL(k, c = 1)$ internally has two $R_{UL,RL}$ states, one that can recover the system to the latest undo-log checkpoint with probability $(1 - p_\delta)$ and another one where the previous checkpoint will have to be used with probability p_δ. The probabilities and recovery times are as follows:

$$p_{RY} = p_{R_0} = (1 - p_\delta) \cdot p_{R_0}(v = 1) + p_\delta \cdot p_{R_0}(v = 2)$$

$$t_{RY} = t_{R_0} = \frac{(1 - p_\delta) \cdot t_{R_0}(v = 1) + p_\delta \cdot t_{R_0}(v = 2)}{p_{R_0}}$$

$$p_{RX_i} = p_{R_i} = (1 - p_\delta) \cdot p_{R_i}(v = 1) + p_\delta \cdot p_{R_i}(v = 2)$$

$$t_{RX_i} = t_{R_i} = \frac{(1 - p_\delta) \cdot p_{R_i}(v = 1) \cdot t_{R_i}(v = 1) + p_\delta \cdot p_{R_i}(v = 2) \cdot t_{R_i}(v = 2)}{p_{R_i}}$$

REFERENCES

[1] AGARWAL, R., GARG, P., and TORRELLAS, J., “Rebound: Scalable Checkpointing for Coherent Shared Memory,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[2] AHMED, R., FRAZIER, R., and MARINOS, P., “Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems,” in *Proceedings of 20th International Symposium Fault-Tolerant Computing*, pp. 82–88, IEEE Comput. Soc. Press, 1990.

[3] AUSTIN, T., “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design,” in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, (Haifa, Israel), pp. 196–207, IEEE Computer Society, 1999.

[4] BANATRE, M., GEFFLAUT, A., JOUBERT, P., MORIN, C., and LEE, P., “An Architecture for Tolerating Processor FaiIures in Shared-Memory Multiprocessors,” *IEEE Transactions on Computers*, vol. 45, no. 10, pp. 1101–1115, 1996.

[5] BANATRE, M. and JOUBERT, P., “Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor,” in *Proceedings of 20th International Symposium Fault-Tolerant Computing*, pp. 89–96, IEEE Comput. Soc. Press, 1990.

[6] BECK, M., PLANK, J. S., and KINGSLEY, G., “Compiler-Assisted Checkpointing,” in *Proceedings of 25th Annual International Symposium on Fault-Tolerant Computing*, 1995.

[7] BERNICK, D., BRUCKERT, B., VIGNA, P., GARCIA, D., JARDINE, R., KLECKA, J., and SMULLEN, J., “NonStop Advanced Architecture,” *Proceedings of 2005 International Conference on Dependable Systems and Networks*, pp. 12–21, 2005.

[8] BIENIA, C., KUMAR, S., SINGH, J. P., and LI, K., “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[9] BLUM, M. and WASSERMAN, H., “Reflections on the Pentium Division Bug,” *IEEE Transactions on Computers*, vol. 45, pp. 385–393, Apr. 1996.

[10] BOEHM, B. and BASILI, V., “Software Defect Reduction Top 10 List,” *Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[11] BOEHM, B. and PAPACCIO, P., “Understanding and Controlling Software Costs,” *Software Engineering, IEEE Transactions on*, vol. 14, no. 10, pp. 1462–1477, 1988.

[12] BOOTHE, B., “Efficient Algorithms for Bidirectional Debugging,” in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, (Vancouver, British Columbia, Canada), pp. 299–310, ACM, 2000.

[13] BORKAR, S., “Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation,” *IEEE Micro*, vol. 25, pp. 10–16, Nov. 2005.

[14] BOSSEN, D., TENDLER, J., and REICK, K., "Power4 system design for high reliability," *IEEE Micro*, vol. 22, pp. 16–24, Mar. 2002.

[15] BRONEVETSKY, G., MARQUES, D., PINGALI, K., and STODGHILL, P., "Automated Application-level Checkpointing of MPI Programs," in *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (San Diego, California, USA), p. 94, ACM, 2003.

[16] BRONEVETSKY, G., MARQUES, D., PINGALI, K., SZWED, P., and SCHULZ, M., "Application-level Checkpointing for Shared Memory Programs," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, New York, USA), p. 235, ACM Press, 2004.

[17] CHEN, Y., PLANK, J., and LI, K., "CLIP: A Checkpointing Tool for Message-Passing Parallel Programs," in *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pp. 1–11, ACM, 1997.

[18] CLEMENT, J., "Electromigration Modeling for Integrated Circuit Interconnect Reliability Analysis," *IEEE Transactions on Device and Materials Reliability*, vol. 1, pp. 33–42, Mar. 2001.

[19] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., and COETZEE, D., "Better I/O Through Byte-Addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, (New York, New York, USA), p. 133, ACM Press, 2009.

[20] CONSTANTINESCU, C., "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, pp. 14–19, July 2003.

[21] DELORD, X. and SAUCIER, G., "Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors," *Proceedings of International Test Conference*, p. 936, 1991.

[22] DONG, X., MURALIMANOHAR, N., JOUPPI, N., KAUFMANN, R., and XIE, Y., "Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, (New York, New York, USA), p. 1, ACM Press, 2009.

[23] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., and CHEN, P. M., "ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, (New York, New York, USA), p. 211, ACM Press, 2002.

[24] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., and CHEN, P. M., "Execution Replay for Multiprocessor Virtual Machines," in *Proceedings of 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, (New York, New York, USA), p. 121, ACM Press, 2008.

[25] ELNOZAHY, E. N. M., ALVISI, L., WANG, Y.-M., and JOHNSON, D. B., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, pp. 375–408, Sept. 2002.

[26] ELNOZAHY, E. and ZWAENEPOEL, W., "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast output Commit," *IEEE Transactions on Computers*, vol. 41, pp. 526–531, May 1992.

[27] FELDMAN, S. I. and BROWN, C. B., "IGOR: A System for Program Debugging Via Reversible Execution," *SIGPLAN Not.*, vol. 24, no. 1, pp. 112–123, 1989.

[28] FENG, S., GUPTA, S., ANSARI, A., and MAHLKE, S., "Shoestring: Probabilistic Soft Error Reliability on the Cheap," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, (Pittsburgh, Pennsylvania, USA), pp. 385–396, ACM, 2010.

[29] FISCHER, A., VON GLASOW, A., PENKA, S., and UNGAR, F., "Electromigration Failure Mechanism Studies on Copper Interconnects," in *Proceedings of the IEEE 2002 International Interconnect Technology Conference*, pp. 139–141, IEEE, 2002.

[30] GARY L. MULLEN-SCHULTZ, C. S., *IBM System Blue Gene Solution: Application Development*. 2007.

[31] GELENBE, E., "A Model of Roll-Back Recovery With Multiple Checkpoints," in *Proceedings of the 2nd International Conference on Software Engineering*, vol. 251, pp. 251–255, IEEE Computer Society Press, 1976.

[32] HOWER, D. and HILL, M., "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," in *Computer Architecture, International Symposium on*, pp. 265–276, June 2008.

[33] INTEL, "Intel 64 and IA-32 Architectures Application Note TLBs, Paging-Structure Caches, and Their Invalidation," *http://www.intel.com/design/processor/applnots/317080.pdf*, 2008.

[34] INTEL, "Intel Xeon Processor 5500 Series," tech. rep., 2009.

[35] INTERNATIONAL TECHNOLOGY ROADMPA FOR SEMICONDUCTORS, "Process intergration, devices & structures," 2007.

[36] KERMARREC, A., CABILLIC, G., GEFFLAUT, A., MORIN, C., and PUAUT, I., "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability," in *ftcs*, p. 0289, Published by the IEEE Computer Society, 1995.

[37] KESSLER, R., "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[38] KIM, S. and SOMANI, A., "On-Line Integrity Monitoring of Microprocessor Control Logic," *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, pp. 314–319, 2001.

[39] KOLAWA, A., "The Evolution of Software Debugging," in *http://www.parasoft.com/jsp/products/article.jsp?articleId=490*, 1996.

[40] LEE, B. C., IPEK, E., MUTLU, O., and BURGER, D., "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, (New York, New York, USA), p. 2, ACM Press, 2009.

[41] LEVESON, N. and TURNER, C., "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, pp. 18–41, July 1993.

[42] Li, M.-L., Ramachandran, P., Sahoo, S. K., Adve, S. V., Adve, V. S., and Zhou, Y., "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, New York, USA), p. 265, ACM Press, 2008.

[43] Linder, B., Stathis, J., Frank, D., Lombardo, S., and Vayshenker, A., "Growth and scaling of oxide conduction after breakdown," in *Proceedings of 41st IEEE International Reliability Physics Symposium Proceedings*, pp. 402–405, Ieee, 2003.

[44] Loh, G., "3D-Stacked Memory Architectures for Multi-Core Processors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pp. 453–464, IEEE Computer Society, 2008.

[45] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 40, (New York, New York, USA), p. 190, ACM Press, 2005.

[46] Mahmood, A. and McCluskey, E., "Concurrent error detection using watchdog processors-a survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988.

[47] Masubuchi, Y., Hoshina, S., Shimada, T., Hirayama, B., and Kato, N., "Fault recovery mechanism for multiprocessor servers," *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pp. 184–193, 1997.

[48] May, T. and Woods, M., "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE Transactions on Electron Devices*, vol. 26, pp. 2–9, Jan. 1979.

[49] McEvoy, D., "The Architecture of Tandem's NonStop System," in *Proceedings of the ACM '81 conference on - ACM 81*, (New York, New York, USA), p. 245, ACM Press, 1981.

[50] Meixner, A., Bauer, M., and Sorin, D. J., "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proceedings of 40th Annual IEEE International Symposium on Microarchitecture*, pp. 210–222, IEEE Computer Society, 2007.

[51] Montesinos, P., Ceze, L., and Torrellas, J., "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[52] Moody, A. T., Bronevetsky, G., and Mohror, K. M., "Detailed Modeling , Design , and Evaluation of a Scalable Multi-level Checkpointing System," *Technical Report*, 2010.

[53] Moody, A., Bronevetsky, G., and Mohror, K., "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," *Proceedings of the 2010 IEEE/ACM Conference on Supercomputing*, 2010.

[54] Morin, C., Kermarrec, A.-M., Banatre, M., and Gefflaut, A., "An Efficient and Scalable Approach for Implementing Fault Tolerant DSM Architectures," *IEEE Transactions on Computers*, vol. 49, pp. 414–430, May 2000.

[55] MORIN, C., GEFFLAUT, A., BANÂTRE, M., and KERMARREC, A.-M., "COMA: An Opportunity for Building Fault-Tolerant Scalable Shared Memory Multiprocessor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, vol. 24, (New York, New York, USA), pp. 56–65, ACM Press, May 1996.

[56] MUKHERJEE, S., EMER, J., FOSSUM, T., and REINHARDT, S., "Cache Scrubbing in Microprocessors: Myth or Necessity?," *10th IEEE Pacific Rim International Symposium on Dependable Computing, 2004. Proceedings.*, pp. 37–42, 2004.

[57] MUKHERJEE, S., KONTZ, M., and REINHARDT, S., "Detailed Design and Evaluation of Redundant Multithreading Alternatives," in *Proceedings 29th Annual International Symposium on Computer Architecture*, pp. 99–110, IEEE Comput. Soc, 2002.

[58] NAKANO, J., MONTESINOS, P., GHARACHORLOO, K., and TORRELLAS, J., "ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers," in *Proceedings of 12th International Symposium on High-Performance Computer Architecture*, IEEE, 2006.

[59] NARAYANASAMY, S., PEREIRA, C., and CALDER, B., "Recording shared memory dependencies using strata," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 40, (New York, New York, USA), p. 229, ACM New York, NY, USA, ACM Press, Oct. 2006.

[60] NARAYANASAMY, S., POKAM, G., and CALDER, B., "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," in *Proceedings of the 32nd International Symposium on Computer Architecture*, pp. 284–295, IEEE, 2005.

[61] OLDFIELD, R. A., ARUNAGIRI, S., TELLER, P. J., SEELAM, S., VARELA, M. R., RIESEN, R., and ROTH, P. C., "Modeling the Impact of Checkpoints on Next-Generation Systems," in *Proceedings of 24th IEEE Conference on Mass Storage Systems and Technologies*, no. Msst, pp. 30–46, Ieee, Sept. 2007.

[62] OUSSALAH, S. and NEBEL, F., "On The Oxide Thickness Dependence of the Time-Dependent-Dielectric-Breakdown," in *Proceedings 1999 IEEE Hong Kong Electron Devices Meeting*, pp. 42–45, IEEE, 2002.

[63] PLANK, J. S., BECK, M., KINGSLEY, G., and LI, K., "Libckpt: Transparent Checkpointing Under Unix," (New Orleans, Louisiana), pp. 18–18, USENIX Association, 1995.

[64] PLANK, J. S., CHEN, Y., LI, K., BECK, M., and KINGSLEY, G., "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software: Practice and Experience*, vol. 29, pp. 125–142, Feb. 1999.

[65] PLANK, J., "Faster checkpointing with N+1 parity," in *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, pp. 288–297, IEEE Comput. Soc. Press, 1994.

[66] PLANK, J. and PUENING, M., "Diskless Checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[67] POULSEN, K., "Software Bug Contributed to Blackout," *http://www.securityfocus.com/news/8016*, 2004.

[68] PRVULOVIC, M. and TORRELLAS, J., “ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors,” in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 111–122, IEEE Comput. Soc, 2002.

[69] PRVULOVIC, M. and TORRELLAS, J., “ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, vol. 30, (New York, New York, USA), p. 110, ACM Press, 2003.

[70] QURESHI, M. K., SRINIVASAN, V., and RIVERS, J. A., “Scalable High Performance Main Memory System Using Phase-Change Memory Technology,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, vol. 37, (New York, New York, USA), p. 24, ACM Press, 2009.

[71] RACUNAS, P., CONSTANTINIDES, K., MANNE, S., and MUKHERJEE, S. S., “Perturbation-based Fault Screening,” *Proceedings of 13th International Symposium on High Performance Computer Architecture*, Feb. 2007.

[72] RANDELL, B., “System Structure for Software Fault Tolerance,” *ACM SIGPLAN Notices*, vol. 10, pp. 437–449, June 1975.

[73] REDDY, V. and ROTENBERG, E., “Coverage of a Microarchitecture-level Fault Check Regimen in a Superscalar Processor,” *Proceedings of IEEE International Conference on Dependable Systems and Networks*, pp. 1–10, 2008.

[74] REED, D., “High-End Computing: The Challenge of Scale,” *Director’s Colloquium*, 2004.

[75] REINHARDT, S. K. and MUKHERJEE, S. S., “Transient Fault Detection via Simultaneous Multithreading,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, vol. 28, (New York, New York, USA), pp. 25–36, ACM Press, 2000.

[76] RENAU, J. and OTHERS, “SESC,” *http://sesc.sourceforge.net*, 2006.

[77] ROTENBERG, E., “AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors,” in *Proceedings of 29th Annual International Symposium on Fault-Tolerant Computing*, (New York, New York, USA), p. 84, Published by the IEEE Computer Society, 1999.

[78] RTI,NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY , “The Economic Impacts of Inadequate Infrastructure for Software Testing,” Tech. Rep. 7007.011, RTI, 2002.

[79] SAITO, Y., “Jockey: A User-Space Library for Record-Replay Debugging,” in *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging - AADEBUG’05*, (New York, New York, USA), pp. 69–76, ACM Press, 2005.

[80] SAMUEL T. KING, GEORGE W. DUNLAP, and PETER M. CHEN, “Debugging Operating Systems with Time-Traveling Virtual Machines,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pp. 1–15, 2005.

[81] SARKAR, V., “ExaScale Software Study: Software Challenges in Extreme Scale Systems,” *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[82] Sastry Hari, S. K., Li, M.-L., Ramachandran, P., Choi, B., and Adve, S. V., "mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems," in *Proceedings of the 42nd Annual International Symposium on Microarchitecture*, (New York, New York, USA), p. 122, ACM Press, 2009.

[83] Schroeder, B. and Gibson, G., "A Large Scale Study of Failures in High-Performance-Computing Systems," *IEEE Transactions On Dependable And Secure Computing*, no. November, 2009.

[84] Schroeder, B. and Gibson, G. A., "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, July 2007.

[85] Schroeder, B., Pinheiro, E., and Weber, W.-D., "DRAM Errors in theWild: A Large-Scale Field Study," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, p. 193, ACM Press, 2009.

[86] Schuette, M. A. and Shen, J. P., "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Transactions on Computers*, vol. C-36, pp. 264–276, Mar. 1987.

[87] Seong, N. H., Woo, D. H., and Lee, H.-h. S., "Security Refresh: Prevent MaliciousWear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, (New York, New York, USA), p. 383, ACM Press, 2010.

[88] Seong, N., Woo, D., Srinivasan, V., Rivers, J., and Lee, H., "SAFER: Stuck-At-Fault Error Recovery for Memories," in *Proceedings of 43rd Annual International Symposium on Microarchitecture*, pp. 115–124, IEEE, 2010.

[89] Shi, W., Lee, H.-H., Falk, L., and Ghosh, M., "An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors," in *Proceedings of the 33rd International Symposium on Computer Architecture*, pp. 102–113, IEEE, 2006.

[90] Shi, W., Lee, H.-H., Gu, G., Falk, L., Mudge, T. N., and Ghosh, M., "An Intrusion-Tolerant and Self-Recoverable Network Service System Using A Security Enhanced Chip Multiprocessor," in *Proceedings of 2nd International Conference on Autonomic Computing*, pp. 263–273, IEEE, 2005.

[91] Silberschatz, A., Korth, H., and Sudarshan, S., *Database Systems Concepts*. New York, NY, USA: McGraw-Hill, Inc., 5 ed., 2006.

[92] Silva, L. and Silva, J., "Using two-level stable storage for efficient checkpointing," *IEE Proceedings - Software*, vol. 145, no. 6, p. 198, 1998.

[93] Sorin, D. J., "Fault Tolerant Computer Architecture," *Synthesis Lectures on Computer Architecture*, vol. 4, pp. 1–104, Jan. 2009.

[94] Sorin, D. J., Martin, M., Hill, M., and Wood, D., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 123–134, IEEE Comput. Soc, 2002.

[95] SRINIVASAN, S. M., KANDULA, S., and ANDREWS, C. R., "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," in *USENIX Annual Technical Conference, General Track*, pp. 29–44, 2004.

[96] STANDARD PERFORMANCE EVALUATION CORPORATION, "SPEC Benchmarks," *http://www.spec.org*, 2006.

[97] SULTAN, F. and NGUYEN, T., "Scalable Fault-Tolerant Distributed Shared Memory," in *ACM/IEEE 2000 Conference on Supercomputing*, Published by the IEEE Computer Society, 2000.

[98] SUNADA, D., GLASCO, D., and FLYNN, M., "Multiprocessor Architecture Using an Audit Trail for Fault Tolerance," in *Proceedings of 29th Annual International Symposium on Fault-Tolerant Computing*, pp. 40–47, IEEE Comput. Soc, 1999.

[99] THOZIYOOR, S. and OTHERS, "Cacti 5.3," *http://quid.hpl.hp.com:9081/cacti/*, 2008.

[100] U.S.-CANADA POWER SYSTEM OUTAGE TASK FORCE, "Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations," 2004. www.nerc.com/docs/pc/spctf/Full_Final_Report.pdf.

[101] VAIDYA, N. H., "A Case for Multi-Level Distributed Recovery Schemes," in *Technical Report*, vol. 23, pp. 64–73, Texas A & M University College Station, TX, USA 2001, May 1995.

[102] VAIDYA, N. H., "A Case for Two-Level Recovery Schemes," *IEEE Transactions on Computers*, vol. 47, pp. 656–666, June 1998.

[103] VENKATARAMANI, G., ROEMER, B., SOLIHIN, Y., and PRVULOVIC, M., "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging," *Proceedings of 13th International Symposium on High Performance Computer Architecture*, pp. 273–284, Feb. 2007.

[104] VMWARE, "VMware Fault Tolerance Recommendations and Considerations on VMware vSphere 4," tech. rep., 2010.

[105] VON NEUMANN, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[106] WAKERLY, J., *Error Detecting Codes, Self-Checking Circuits and Applications*. 1978.

[107] WANG, N. and PATEL, S., "ReStore: Symptom-Based Soft Error Detection in Microprocessors," in *IEEE Transactions on Dependable and Secure Computing*, vol. 3, pp. 188–201, July 2006.

[108] WITCHEL, E., CATES, J., and ASANOVIĆ, K., "Mondrian Memory Protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, New York, USA), p. 304, ACM Press, 2002.

[109] WU, K.-L., FUCHS, W., and PATEL, J., "Error Recovery in Shared Memory Multiprocessors Using Private Caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 231–240, Apr. 1990.

[110] Xu, M., Bodik, R., and Hill, M. D., "A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, vol. 31, p. 122, ACM Press, 2003.

[111] Xu, M., Hill, M. D., and Bodik, R., "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 34, (New York, New York, USA), p. 49, ACM Press, 2006.

[112] Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., and Weissman, B., "ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay," in *Third Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[113] Yeh, Y., "Triple-Triple Redundant 777 Primary Flight Computer," *Proceedings of 1996 IEEE Aerospace Applications Conference*, pp. 293–307, 1996.

[114] Yu, H., Sahoo, R., Howson, C., Almasi, G., Castanos, J., Gupta, M., Moreira, J., Parker, J., Engelsiepen, T., Ross, R., Thakur, R., Latham, R., and Gropp, W., "High Performance File I/O for The Blue Gene/L Supercomputer," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, (Austin, Texas), pp. 190–199, IEEE, 2006.

[115] Ziegler, J. F., "Terrestrial cosmic rays," *IBM Journal of Research and Development*, vol. 40, pp. 19–39, Jan. 1996.

[116] Ziegler, J. F., Curtis, H. W., Muhlfeld, H. P., Montrose, C. J., Chin, B., Nicewicz, M., Russell, C. a., Wang, W. Y., Freeman, L. B., Hosier, P., LaFave, L. E., Walsh, J. L., Orro, J. M., Unger, G. J., Ross, J. M., O'Gorman, T. J., Messina, B., Sullivan, T. D., Sykes, a. J., Yourke, H., Enger, T. a., Tolat, V., Scott, T. S., Taber, a. H., Sussman, R. J., Klein, W. a., and Wahaus, C. W., "IBM experiments in soft fails in computer electronics (1978–1994)," *IBM Journal of Research and Development*, vol. 40, pp. 3–18, Jan. 1996.