PROVIDING APPLICATION-AWARE RELIABILITY THROUGH OS/HYPERVISOR-LEVEL TECHNIQUES

BY

LONG WANG

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Associate Professor Steven S. Lumetta
Assistant Professor Madhusudan Parthasarathy
Assistant Professor Shobha Vasudevan
ABSTRACT

Operating systems and hypervisors enable the collection and extraction of rich information on application and system execution characteristics. This thesis describes a Reliability MicroKernel (RMK) architecture, which provides an infrastructure that enables the design and deployment of software modules for providing application-aware error detection and recovery.

The purpose of the RMK is to provide an automatic approach for low-latency crash/hang detection and rapid recovery via checkpoint. We first demonstrate how the RMK works in a native system and then enhance the RMK to work in VMs. In a native system, the RMK is installed as a device driver, while in a virtualized system, the RMK is both installed as a device driver in VMs and deployed as a hypercall (which is like a system call) in a hypervisor. Our approach is transparent to applications and VMs, i.e., it is not required to modify or recompile the kernel source code in a native system or in a VM.

The implemented RMK modules include OS/application crash detection, system hang detection, and transparent checkpoint. Traditionally, an external hardware watchdog is used to force a system reboot whenever the watchdog is not reset within a predefined timeout interval. The detection latency might be significant because the timeout interval for resetting the watchdog timer is usually a matter of seconds to reduce false alarms. The approach in this thesis enables low-latency OS-hang detection (within hundreds of milliseconds or less) by measuring the count of instructions executed between two consecutive context switches and checking if the count exceeds a predefined threshold value.

The RMK is enhanced to support virtualized environments. Specifically, we present the description, implementation, and experimental assessment of VM-µCheckpoint, a VM
checkpointing framework to protect both the guest OS and applications against runtime errors. Compared with the existing VM checkpoint techniques, our VM-µCheckpoint has small overhead and rapid recovery, handles non-fail-stop errors, and runs at high frequency (tens of checkpoints per second) to reduce the recomputation necessary when recovering a VM from a failure. The key point of VM-µCheckpoint is that we do an incremental checkpoint by considering the whole memory of the protected VM as part of the checkpoint.

The RMK prototype has been implemented in both Linux and Windows systems on a Pentium 4 processor and is also implemented in the Xen VMM. (The Xen hypervisor is recompiled for installing RMK, but the OS of a native system or a VM is not recompiled.)

Error injection experiments show that our RMK detects all the crashes and system hangs, and VM-µCheckpoint successfully recovers VMs from all the crashes. Moreover, the experimental evaluation of the RMK using real-world applications shows that we achieve high coverage and low false-positive rates for error detection (e.g., no false positives for system hang detection) as well as low overhead in providing checkpoint and recovery (e.g., an average of 6.3% overhead in VM-µCheckpoint for SPEC benchmark programs with 50 ms checkpoint intervals).

We also apply a formal method and analytical/probabilistic models to verify the capability of our system hang detection and to study the availability enhancement provided by the RMK.
To Weili, Father, and Mother
ACKNOWLEDGMENTS

I am heartily thankful to my adviser, Prof. Ravishankar K. Iyer, whose guidance, supervision, and support were indispensible to the completion of this dissertation. Prof. Iyer’s insightful advice and suggestions have enabled me to develop a clear understanding of the dissertation topic. I also thank Prof. Zbigniew Kalbarczyk for his guidance during my PhD program. This dissertation would not have been possible without the frequent discussions I had with Prof. Kalbarczyk.

I also had many discussions and chats with my officemates Karthik Pattabiraman and Weining Gu while I worked on this dissertation. Arun Iyengar also helped me with the VM checkpoint project when I started this work as an intern at IBM Watson Research Center. Here I thank Karthik, Weining, and Arun for their great help.

My PhD committee members Prof. Steven S. Lumetta, Prof. Madhusudan Parthasarathy, and Prof. Shobha Vasudevan have contributed valuable comments and opinions to my PhD work. Moreover, as a member of the DEPEND group, I received a great deal of help from the members of this group. The weekly group meeting gave me many hints and inspirations during the past years. I am grateful to my committee and to all the members of the DEPEND group for their help.

Now I offer my thanks to Weili, my dear wife, and to my parents. It would have been impossible for me to complete the seven years of my PhD program and this dissertation without their understanding, encouragement, and patience throughout the entire time.

Lastly, I thank all of those who supported me in any way during my PhD program and with this dissertation.
TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION ..........................................................................................................................1
  1.1. Low-Latency System Hang Detection .................................................................................................2
  1.2. High-Frequency Checkpoint of VMs ....................................................................................................3
  1.3. Approach Evaluation ..........................................................................................................................5
  1.4. Thesis Organization .............................................................................................................................6

CHAPTER 2 RELIABILITY MICROKERNEL ARCHITECTURE .................................................................8
  2.1. Introduction ........................................................................................................................................8
  2.2. RMK Framework ...............................................................................................................................10
  2.3. RMK Implementation on Linux and Windows ....................................................................................18

CHAPTER 3 RMK MODULES .........................................................................................................................22
  3.1. System Hang Detection Module (SHD) ..........................................................................................22
  3.2. Application Hang Detection Module (AHD) ..................................................................................29
  3.3. Transparent Application Checkpoint Module (TAC) ....................................................................33
  3.4. Experimental Evaluation ...............................................................................................................38

CHAPTER 4 FORMALIZING SYSTEM BEHAVIOR FOR EVALUATING SHD .........................................51
  4.1. Introduction .......................................................................................................................................51
  4.2. Related Work ....................................................................................................................................53
  4.3. System Hang Detector ......................................................................................................................54
  4.4. System Abstraction and Modeling ....................................................................................................55
  4.5. Modeling System Behavior ..............................................................................................................59
  4.6. Implementation ...............................................................................................................................71
  4.7. Experimental Results ......................................................................................................................76
  4.8. Conclusions .....................................................................................................................................79

CHAPTER 5 CHECKPOINTING VMS AGAINST TRANSIENT ERRORS ................................................81
  5.1. Introduction .......................................................................................................................................81
  5.2. VM-µCheckpoint Design ..................................................................................................................84
  5.3. Implementation ...............................................................................................................................95
  5.4. Experimental and Measurement Results .........................................................................................98

CHAPTER 6 INTEGRATING VM CHECKPOINTING INTO RMK .............................................................107
  6.1. RMK Deployment in Virtualized Environment .................................................................................107
  6.2. Detecting and Recovering from VM/Application Crashes .............................................................108
  6.3. Detecting and Recovering from VM Hangs ....................................................................................109
  6.4. Error Injection Experiments ..........................................................................................................110

CHAPTER 7 MODEL-BASED ANALYSIS OF VM CHECKPOINTING ....................................................115
  7.1. Checkpoint Corruption Model .......................................................................................................115
  7.2. Availability Model ............................................................................................................................119
CHAPTER 1
INTRODUCTION

Operating systems enable the collection and extraction of rich information on application execution characteristics, including counts of executed instructions, program counter traces, memory access patterns, and OS-generated signals. Similarly, hypervisors enable monitoring of virtual machine (VM) execution and collection of information on characteristics of the execution. Such information can be exploited to design highly efficient, application-aware, error detection and recovery mechanisms that are transparent to applications or VMs.

This thesis describes the design, implementation, and demonstration of a Reliability MicroKernel (RMK) architecture. The RMK provides an infrastructure that enables the design and deployment of software modules for providing application-aware error detection and recovery.

The purpose of the RMK is to provide an automatic approach for low-latency crash/hang detection and rapid recovery via checkpoint. While some of the existing operating systems may, by design, provide reliability support, e.g., IBM AIX [1] and High-Availability Linux [2], their emphasis is on achieving system reliability rather than exploiting execution characteristics of applications or VMs for reliability improvements.

We first demonstrate how the RMK works in a native system and then enhance the RMK to work in VMs. In a native system the RMK is installed as a device driver, while in a virtualized system, the RMK is both installed in VMs as a device driver and deployed as a hypercall (which is like a system call) in a hypervisor. Our approach is transparent to applications and VMs; i.e., it is not required to modify or recompile the kernel source code in a native system or in a VM.
The implemented RMK modules include OS/application crash detection, system hang detection, and transparent checkpoint. The architecture exploits processor-level features (debugging and monitoring facilities available in the current generations of processors), OS-exported interfaces, and hypervisor features (e.g., virtual CPU, shadow paging, etc.) to define a set of basic services. These basic services are called RMK pins, which are analogous to hardware pins in providing clearly defined functionalities and inputs/outputs. The pins are employed to design specific error detection and recovery mechanisms, referred to as RMK modules.

The attributes of the currently implemented architecture allow the following:

- Design and deployment of application-aware reliability techniques;
- On-demand configuration and customization of reliability techniques; and
- Platform independence of the RMK architecture and modules.

In addition to the RMK design and implementation as an automatic approach for combining error detection, checkpoint, and error recovery, this thesis makes the contributions summarized in the following two sections.

1.1. Low-Latency System Hang Detection

The operating system may suffer from hangs due to poorly written drivers (e.g. unreleased locks). Traditionally an external hardware watchdog is used to force a system reboot whenever the watchdog is not reset within a predefined timeout interval. In this case, however, one cannot determine whether the OS has crashed/hung, or the heartbeat process, designated to reset the watchdog periodically, has crashed/hung. Also, the detection latency might be significant because
the timeout interval for resetting the watchdog timer is usually a matter of seconds (to reduce false alarms, and the time overhead of the heartbeat).

The approach in this thesis enables low-latency OS-hang detection. In a system executing a set of same-priority tasks, the count of instructions executed between two consecutive context switches is a finite number. The underlying fault model for a system hang is that an operating system in a hang state does not relinquish the processor, and does not schedule any processes. Based on this fault model, if the system hangs, this instruction count grows beyond a limit. So we detect system hangs by measuring this instruction count and checking if it exceeds a predefined threshold value.

Hardware counters in current generation of processors are used in the detection for counting executed instructions. In a native system we configure the processor to raise a non-maskable interrupt (NMI) when the threshold value is exceeded; in a virtualized environment we instrument the hypervisor to periodically check if the threshold value is exceeded. As the time-slice allocated by the OS for a process between two consecutive context switches is tens or hundreds of milliseconds (100 ms for typical processes in Linux 2.6, and 20/40/60 ms in Windows XP), a system hang can be detected within hundreds of milliseconds.

1.2. High-Frequency Checkpoint of VMs

Virtual machines play an increasingly significant role in today’s computing environment for IT services (e.g., web services, virtual desktops, and databases). Checkpoint and rollback of VMs is essential to ensure continuous service availability. Virtual machine monitors (VMMs), such as VMware and Xen, provide mechanisms to (a) save a VM state by stopping the VM and dumping the execution state into persistent storage and (b) migrate the VM to a remote node (e.g., [3]).
This thesis presents the description, implementation, and experimental assessment of VM-\(\mu\)Checkpoint, a VM checkpointing framework to protect both the guest OS and applications against runtime errors. Advantages of using VM-\(\mu\)Checkpoint include (i) *small overhead* compared with the VM replica-based failover approach, (ii) *alleviation of checkpoint corruption due to error-detection latency* by taking advantage of knowledge of error detection latency (we deal with non-fail-stop errors), (iii) *high checkpointing frequency*—tens of checkpoints per second—which reduces the size of each increment when taking a checkpoint and reduces the recomputation when recovering a VM from a failure, and (iv) *rapid recovery*—within one second—compared to the stop-and-dump approach provided by VMMs.

The key point of VM-\(\mu\)Checkpoint is that we do an incremental checkpoint by considering the whole memory of the protected VM as part of the checkpoint. Specifically, if a memory page is updated during a checkpoint interval, we preserve the original state of the page before any write is done; then the memory pages not updated during the checkpoint interval are considered to be the VM checkpoint, which significantly reduces checkpoint overhead. This incremental checkpoint (i) leverages the fact that when a VM fails its whole memory is still accessible by the hypervisor for recovery purposes and (ii) is based on the observation that the chance of a transient error in a VM crashing all the VMs and the hypervisor on the same node is relatively small (one case out of thousands of experiments that inject errors into VMs). \(^1\)

In addition to the high-frequency checkpointing of VM memory, we also have a design for disk-based checkpointing to deal with permanent hardware failures or physical machine failures. The disk-based checkpoint is taken in much lower frequency, e.g., once every couple of hours.

\(^{1}\) My colleague, Weining Gu, conducted this error injection campaign for IBM Power series. The result of the campaign is not published yet.
1.3. Approach Evaluation

The RMK prototype has been implemented in both Linux and Windows systems on a Pentium 4 processor and is also implemented in the Xen VMM. (The Xen hypervisor is recompiled for installing RMK, but the OS of a native system or a VM is not recompiled.)

The effectiveness and performance of the RMK is assessed by two means: experimental evaluation and model-based analysis. Error injection experiments show that our RMK detects all the crashes and system hangs, and VM-µCheckpoint successfully recovers VMs from all the crashes and most injected system hangs. Split driver mode is used for I/O operations in Xen. Because I/O checkpoint is not implemented in the current VM-µCheckpoint prototype, there are cases when the shared states between split drivers in two VMs cause inconsistency and cause the recovery to fail.

Moreover, the experimental evaluation of the RMK using real-world applications shows that, by exploiting characteristics of application and system behavior, we are able to achieve high coverage and low false-positive rates for error detection (e.g., no false positives for system hang detection) as well as low overhead in providing checkpoint and recovery (e.g., an average of 6.3% overhead in VM-µCheckpoint for SPEC benchmark programs with 50 ms checkpoint intervals).

We also apply formal method to verify the capability of our system hang detection in the RMK. Several corner scenarios that lead to system hang and escape detection are identified by the formal method, but these scenarios are unlikely to occur in real systems (e.g., an interrupt coinciding with a context switch has a low probability).

We construct an analytical model and a Markov model to study the availability enhancement
provided by VM-µCheckpoint. The availability values computed from these models show that we achieve better results than existing migration-based VM checkpointing. For example, for an average job duration of 8 hours in a 99%-available VM (on top of a hypervisor with the MTTF of 1.7 years), we achieve an availability of 99.7%, while the migration-based VM checkpointing achieves 97.7%.

1.4. Thesis Organization

The thesis is organized into the following chapters: Chapter 2 describes the architecture of the RMK. Chapter 3 discusses a number of RMK modules (i.e., error detection or checkpoint techniques) on standalone systems. Chapter 4 presents formal method analysis to verify the system hang detection technique proposed in Chapter 3. Chapter 5 describes our algorithm of VM checkpointing. Chapter 6 discusses how the RMK is enhanced in the virtualized environment and how the error detection techniques and VM checkpointing are integrated to enhance VM availability. Chapter 7 provides an analysis of the VM checkpointing in enhancing VM availability by employing an analytical model and a Markov model. Chapter 8 lists the related work, and Chapter 9 concludes the thesis.

I completed several projects prior to my thesis work. Appendix A presents stochastic models of coordinated checkpointing to study the error-present behavior of large-scale supercomputers. We found that the resource utilization is largely limited if we only apply the coordinated checkpointing in the large supercomputer (more than 50% of resources are spent in checkpointing and recovery). So it is crucial to enhance the single-node reliability rather than just apply coordinated checkpointing. This motivates my thesis work to provide an automatic approach for low-latency error detection and rapid checkpointing/recovery.
Appendix B presents my previous work on incremental checkpointing in a main-memory database application. My experience is that the system downtime is reduced by 5 times through exploiting the knowledge of application semantics. This shows the need for monitoring application/system behavior and collecting the information on application/system semantics, and leads to the RMK framework in this thesis.
CHAPTER 2

RELIABILITY MICROKERNEL ARCHITECTURE

2.1. Introduction

The Reliability MicroKernel (RMK) architecture, deployed as a device driver, provides an infrastructure that enables the design and deployment of software modules for providing application-aware reliability services. The implemented RMK modules include OS/application crash detection, system hang detection, and transparent checkpoint. The architecture exploits processor-level features (debugging, and monitoring facilities available in the current generations of processors), and OS-exported interfaces to define a set of basic services. These basic services are called RMK pins, which are analogous to hardware pins in providing clearly defined functionalities, and inputs/outputs. The pins are employed to design application-specific mechanisms, referred to as RMK modules, for the runtime system monitoring, and low-latency error detection.

The attributes of the currently implemented architecture allow the following.

- **Design, and deployment of application-aware reliability techniques.** The RMK wraps (or abstracts out) the functionalities of the OS, and the underlying hardware into RMK pins. Using the interfaces, and services exported by the pins, designers of reliability modules can focus on developing application-specific techniques/algorithms without detailed knowledge on the specifics of the OS, or the underlying hardware. Several techniques have been implemented to demonstrate how OS knowledge on application execution can be used to
provide error detection, and recovery. Table 1 briefly summarizes the techniques currently available in the RMK.

**Table 1: Reliability Mechanisms in the RMK**

<table>
<thead>
<tr>
<th>Reliability Service</th>
<th>Application Execution Pattern</th>
<th>Technique Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Hang Detection</td>
<td>Number of instructions executed within a well-defined code block, e.g., a loop</td>
<td>Number of instructions executed within the code block is counted. If the count goes beyond a preset scope, a hang is flagged.</td>
</tr>
<tr>
<td>Application Crash Recovery</td>
<td>OS signal delivered to application</td>
<td>Delivery of the terminating signal to a process is intercepted. If the signal is not handled, the process is recovered from its checkpoint.</td>
</tr>
<tr>
<td>Transparent Application Checkpoint</td>
<td>Memory access patterns</td>
<td>Original pages written during the checkpoint interval are backed up. If the application fails while the system is still operational, the original pages are restored.</td>
</tr>
<tr>
<td>System Hang Detection</td>
<td>Number of instructions executed between two consecutive context switches</td>
<td>The count of instructions executed between two consecutive context switches is a finite number. If the system hangs, it does not schedule processes, and the instruction count goes beyond the preset scope.</td>
</tr>
</tbody>
</table>

- **On-demand configuration, and customization of reliability techniques.** The RMK enables on-demand configuration of reliability support provided to applications. RMK pins, and RMK modules can be installed or removed on demand.

- **Platform independence of the RMK architecture and modules.** The architecture of the RMK is independent of which platform (hardware, and operating system) the RMK is on. Moreover, the set of RMK modules implemented on an operating system (e.g. Linux) can be deployed onto other systems (e.g. Solaris, FreeBSD, or Windows systems) with minimal or no code changes, as long as corresponding RMK pins (the platform-dependent components) are implemented, and compiled for the target operating system. Currently, the RMK has been implemented on both Linux and Windows systems.
Fault/error injection experiments conducted to evaluate the efficiency of the RMK implementation show that the OS-level mechanisms detect all application, and system hangs (due to injected errors) with a very low false positive rate (1 out of 2000 experiments for application hang detection, and no false positives for system hang detection).\(^2\) Additionally, the RMK-based mechanisms provide low-latency detection, and transparent checkpointing with little impact on system performance (0.6% performance overhead for the application hang detection, and 0.1% overhead for the transparent application checkpointing).

2.2. RMK Framework

The Reliability MicroKernel (RMK) framework is developed as a loadable kernel module (or a device driver) in the operating system. The RMK has a two-level hierarchy, shown in Figure 1. The lower level (RMK pins) interfaces with the system, and hardware, while the upper level (RMK modules) hosts application-specific detection, and recovery techniques. RMK pins encapsulate low-level services of the system, and of the underlying hardware. Well-defined pin interfaces (available to users) can be selectively used to build RMK modules. Each RMK module implements a specific error detection or recovery mechanism, e.g., crash detection, and recovery of applications; hang detection of applications; hang detection of the operating system; or transparent application checkpoint, and recovery. The RMK core between the two levels manages the installation, and de-installation of pins, and modules; it also invokes pin functionalities on behalf of RMK modules. A set of RMK API is provided by the RMK core for the management purpose.

\(^2\) The 100% coverage and the false positive rate of 1 out of 2000 are the observed results of the conducted experiments.
The abstraction introduced by the two-level RMK structure enables: (i) the use of standard operating system functions to create service essentials in designing, and implementing reliability mechanisms (encapsulated as RMK modules); (ii) the portability of RMK modules across operating systems, because pins implemented on different platforms export the same interfaces (no need to modify the module code); and (iii) transparent coordination between multiple modules, and pins to avoid potential conflicts. For example, two RMK modules, system hang detection and application hang detection, both intercept process context switches to detect hangs. The RMK coordination mechanism handles scenarios such as the installation of application hang detection while system hang detection is already in place, and allows the two modules to function without conflicts.

2.2.1. System-Level RMK Interface

The RMK interfaces with the operating system (or more generically with the runtime environment) via RMK pins. A pin uses a collection of OS functions to construct a specific service that is essential in providing reliability mechanisms. In this sense, a pin implementation is platform-specific. But while the pin implementation on different platforms may differ, the pin
functionality and exported interface are intended to be the same, regardless of the platform.\textsuperscript{3} For example, Figure 2 depicts the architecture (Figure 2a) and implementation (pseudocode in Figure 2b) of the $P_{SCHL}$ pin in Linux to intercept the process context switch, which is an essential service in enabling application, and/or system hang detection.\textsuperscript{4}

![Diagram](image.png)

**Figure 2: The Implementation of an Example Pin, $P_{SCHL}$ in Linux**

In modern operating systems, context switches are transparent to applications; i.e., an application is not notified when a context switch occurs. A common method of intercepting context switches would be to instrument the scheduler. The $P_{SCHL}$ pin, however, takes advantage of the debug exception mechanism to intercept context switches without instrumentation of the operating system source code.

Generically, the interface exported by a pin consists of the following two sets.

- A set of operations the pin can perform. In the case of $P_{SCHL}$ the operations include INTERCEPT, and RESTORE (see Figure 2).

---

\textsuperscript{3} It is assumed that the basic functionalities (e.g., scheduler) are available across the operating systems considered here.

\textsuperscript{4} In the Windows operating system, the pin interface for $P_{SCHL}$ is the same, but the implementation is slightly different.
A set of events the pin can produce given a trigger condition (a process context switch in the case of P_SCHL pin). In the case of P_SCHL, the event set includes EVT_CONTSWITCH (see Figure 2).

The INTERCEPT operation within the P_SCHL pin performs two primary tasks: (i) it writes the scheduler entry point (the address of the first instruction of the\textit{schedule()}\textit{) into one of the processor breakpoint registers (DR3 in Pentium 4), thus forcing the processor to raise a debug exception whenever \textit{schedule()} is invoked,\(^5\) and ii) it modifies the system’s interrupt vector table so that the \textit{debug exception interrupt vector} points to the custom exception handler, \textit{my_DB_exp()}, which (in addition to the default debug exception handler) generates an event, EVT_CONTSWITCH, to indicate the context switch.

A small set of RMK API (\textit{use_pin ()}, \textit{release_pin ()}, \textit{issue_cmd ()}, \textit{subscr_event ()}, and \textit{unsubscr_event ()}) is implemented to enable transparent subscription to events, and the invocation of pin operations by RMK modules. In our example, the OS, and/or application hang detection module subscribes to the event exported by the P_SCHL pin. The subscription table (i.e., the mapping between an event, and a module or modules) is maintained by the dispatcher (see Figure 2a) and populated at the time of module initialization.

Although RMK pins deal with system specifics, no kernel source patch or recompilation is needed for pin implementation or deployment. This is because the interfaces exported by operating systems, and the debugging and monitoring facilities available in modern processors, can be exploited for this purpose. For example, Linux exports a large number of kernel functions, and variables in a symbol list stored in the file /boot/System.map (23,306 symbols in Linux 2.6.11.3).

\(^5\) In Linux, \textit{schedule()} is always invoked when a context switch occurs.
Table 2 lists the RMK pins currently implemented in the RMK prototype.

Table 2: RMK Pins in the RMK Prototype

<table>
<thead>
<tr>
<th>RMK Pin</th>
<th>Hardware/OS Features</th>
<th>Pin Functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_PMC</td>
<td>Hardware counters available in modern processors; permit monitoring and measuring</td>
<td>Configures the counters to measure specific parameters; starts/stops counting;</td>
</tr>
<tr>
<td></td>
<td>processor performance parameters, including instruction count, TLB access, and cache</td>
<td>reads/writes counters; generates a PMI interrupt to APIC when a counter reaches zero.</td>
</tr>
<tr>
<td></td>
<td>references.</td>
<td></td>
</tr>
<tr>
<td>P_APIC</td>
<td>Advanced Programmable Interrupt Controller (APIC) provided in modern processors.</td>
<td>Configures APIC for custom interrupt generation, e.g., generating an NMI interrupt</td>
</tr>
<tr>
<td></td>
<td>APIC receives interrupts from processor pins or external interrupt controller and</td>
<td>when a performance monitor counter raises a PMI to APIC.</td>
</tr>
<tr>
<td></td>
<td>delivers interrupts to the processor core.</td>
<td></td>
</tr>
<tr>
<td>P_DBR</td>
<td>Debug facilities available in hardware, including debug exception and debug registers.</td>
<td>Sets up a debug exception at a particular location; installs debug exception handler;</td>
</tr>
<tr>
<td>P_INTR</td>
<td>OS-level interrupt handling.</td>
<td>generates an event upon debug exception.</td>
</tr>
<tr>
<td>P_SIG</td>
<td>Signals delivered by the operating system to processes.</td>
<td>Intercepts particular signals to processes; generates an event upon a signal delivery.</td>
</tr>
<tr>
<td>P_MEM</td>
<td>Memory management provided in the OS.</td>
<td>Copies memory pages; sets page properties.</td>
</tr>
<tr>
<td>P_SCHL</td>
<td>Process context switch supported by OS scheduler.</td>
<td>Intercepts context switches; generates an event upon a context switch.</td>
</tr>
<tr>
<td>P_PERI</td>
<td>System-level periodic jobs performed by the work queues of the OS.</td>
<td>Sets up a system-level periodic job at a specified interval; generates an event upon</td>
</tr>
<tr>
<td></td>
<td></td>
<td>interval expiration.</td>
</tr>
<tr>
<td>P_NET</td>
<td>Network communications performed by the system.</td>
<td>Intercepts network activities, including message sending and arrival; generates</td>
</tr>
<tr>
<td></td>
<td></td>
<td>events upon these activities.</td>
</tr>
</tbody>
</table>

6 The P_DBR pin handles use of debug registers for debugging. When the debugged application is switched off the CPU, the pin saves the current value of the registers and sets new values. When the debugged application is scheduled onto processor, the pin restores the saved register value. In cases when debug registers are used to intercept process context switch (like the system hang detection module on Linux), a binary rewriting technique can be applied to implement context switch interception, and the debug registers are not used (like the system hang detection module on Windows).
Table 2: Continued

<table>
<thead>
<tr>
<th>P_SYSC</th>
<th>A variety of system calls executed by the system on behalf of applications.</th>
<th>Intercepts specific system calls; generates an event upon a system call.</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_FILE</td>
<td>Functions to manage the table of open files and read/write operations.</td>
<td>Wraps the manipulation of the file table; generates an event when a monitored file read/write completes.</td>
</tr>
<tr>
<td>P_IPC</td>
<td>IPC mechanisms, such as pipes, message queues, and semaphore.</td>
<td>Wraps the manipulation of IPCs; generates an event when a monitored IPC occurs.</td>
</tr>
</tbody>
</table>

2.2.2. Application-Level RMK Interface

Applications are monitored by RMK modules, which implement application-specific detection, and recovery techniques. While most modules are application-transparent, for some RMK modules the application may need to be instrumented, for example by using an enhanced compiler, to insert a system call in the application code to enable invoking a given RMK module.

An RMK module can protect a set of applications on the system. The RMK allows users to associate a set of applications with RMK module(s). This information is kept in the dispatcher of the RMK core. Moreover, RMK modules can be installed and removed on demand by users. A specially designed RMK module, Configuration Manager, enables RMK reconfiguration.

2.2.3. RMK Core

The RMK core is responsible for: (i) maintaining mapping between the events generated by pins and the modules subscribed to these events; (ii) delivering the events to the modules; (iii) dispatching operation requests to the pins; and (iv) managing and configuring the pins and modules (e.g., installation/removal). The RMK core consists of four components, illustrated in Figure 3, and listed here.

- **Pin manager** is responsible for registration, and loading/unloading of RMK pins.
- **Module manager** handles installation, and removal of RMK modules.

- **Dispatcher** maintains subscription information for events generated by pins (i.e., mapping between events and RMK modules), and delivers the events to modules according to the subscription list.

- **RMK communication channel** facilitates remote invocation of pins in a networked environment, i.e., enables requesting a pin operation, or subscribing to an event generated by a pin on a remote node. This remote pin invocation facilitates design of distributed reliability mechanisms in RMK.

---

**Figure 3: RMK Architecture**

**On-demand configuration of the RMK.** RMK modules and pins can be deployed and removed on demand. When a module is installed, it acquires services from a set of pins. If the required pins are not available in the RMK (e.g., specific pins for a new module), they are automatically loaded into memory from permanent storage, e.g., a disk. The on-demand RMK configuration is initiated by the dispatcher, and carried out by the pin/module managers (see Figure 3).
We now provide an illustration of how the RMK is configured to meet the needs of the specific error-detection mechanisms. Assume that the RMK has installed only one module, the system hang detection module, which loads five pins (P_SCHL, P_PMC, P_APIC, P_DBR, P_INTR) into memory. A new module, the application hang detection module (AHD), is to be deployed to protect an application. The AHD module uses six pins: P_SCHL, P_PMC, P_DBR, P_INTR, P_PERI, P_SYSC. Now, during initialization of the AHD module, services from these six pins are requested via the RMK API. On receiving the service requests, the dispatcher forwards them to the pin manager, which finds that P_PERI, and P_SYSC are not loaded into memory. Recall that the pin manager maintains the state of all the pins in the local host indexed by the pin ID. The pin manager then loads the two additional pins from the disk, and sends a success response to the module via the dispatcher. If the required pins are not found, an error response is sent. After successful module initialization, the dispatcher configures the event-subscription table to establish corrected mapping between the newly deployed AHD module and the events generated by the pins.

**RMK self-checking.** Errors in the RMK may cause system failures. The dispatcher, pin/module managers, and communication channels are well implemented, and thoroughly tested. Due to the simplicity of RMK pins, errors in pin implementations can be considered negligible. Errors in RMK modules are confined using the following method. Whenever an operation of a module is invoked, the RMK records the module ID. The record is cleared after the operation is finished. As there is no recursive invocation of module operations, the recorded ID always indicates the currently executing module. When an error in a module triggers a kernel exception, the RMK intercepts the exception through the P_INTR pin, checks the recorded module ID, and unloads the module (the module can be reloaded immediately after the unloading if that is preferred).
2.3. RMK Implementation on Linux and Windows

The decoupling of RMK modules (platform-independent) and RMK pins (platform-dependent) allows for implementing the RMK architecture and RMK modules only once (except for the RMK modules with platform-specific semantics), and re-implementing the RMK pins accordingly on different platforms. RMK pins are small in terms of the code size (around a few hundred lines for each pin). Figure 4 and Figure 5 depict the implementation and deployment of the RMK on Linux 2.6.11, and Windows XP Professional (SP2), respectively. In both figures, the RMK is deployed as a loadable kernel module (or a device driver). Construction of a set of sample RMK pins is also shown in the figures.

The RMK on Linux and Windows (RMK.ko, and RMK.sys, respectively) have the same architecture, and RMK modules have the same implementation without major code changes. However the implementation of RMK can be: (i) OS dependent if a pin uses OS specific services, or (ii) OS independent if a pin interacts only with the hardware rather than with the OS. The rest of this section discusses examples of pins from the two categories.
Figure 4: The RMK Implementation, and Deployment on Linux 2.6.11

Figure 5: The RMK Implementation and Deployment on Windows XP Professional
For example, the implementation of the P_SCHL pin, which intercepts process context switches, and issues the EVT_CONTSWITCH event, is OS-dependent. The context switch routine on Linux, `schedule()`, is exported by the Linux kernel, and can be intercepted by setting a breakpoint at the entry of `schedule()`. Whenever `schedule()` is invoked, a breakpoint handler is invoked, which generates EVT_CONTSWITCH, and delivers it to the RMK core. On Windows, the context switch routine, `SwapContext()`, is not exported and can be intercepted by applying the *kernel inline hooking* technique [4], which allows searching for predefined instruction patterns in the kernel code space (*ntoskrnl.exe*). The technique is used to search for a sequence of instructions found at the beginning of the `SwapContext()`. These instructions are the same for all Windows XP systems and can be obtained by analyzing the *ntoskrnl.exe* file offline. After locating the `SwapContext()`, the first instruction of `SwapContext()` is replaced with an unconditional jump to the hook function,\(^7\) and at the end of the hook function the replaced instruction is executed, and the control flow jumps back to the instruction in `SwapContext()` that follows the replaced instruction.

Another pin with different implementations on Linux versus Windows is P_SYSC, which intercepts system calls. Although the system call tables on both of the platforms (*sys_call_table* in Linux, and *KeServiceDescriptorTable* in Windows) are exported by the kernel, they each have specific data structures which require different handlings.

The P_APIC, and P_PMC pins can be reused without a reimplementation because they interact with the hardware instead of the operating systems. Similarly, the P_SCHL pin for Windows can be reused in any Windows system on different hardware. Also note that RMK pins do not need to be reimplemented across variants of a platform. So the implementation of the P_SCHL pin on

\(^7\) If the size of the first instruction is less than the size of the jump instruction, two or more instructions are replaced.
Linux 2.6 works on Linux 2.4, and the implementation of the P_SCHL pin on Windows XP works on the Windows 2003 Server as well.

**RMK use scenario.** RMK-supported services are portable across different platforms (hardware, and operating systems). As a result, a broad range of applications can potentially benefit from RMK. Consider a cluster of web servers on a number of heterogeneous nodes (Linux, and Windows). These web servers may crash or hang due to many causes including corrupted (or invalid) input, or bugs in poorly written device drivers. With the RMK installed on every node, the server downtime and the system downtime can be greatly reduced, and the availability of web services can be largely enhanced. With the platform-independence of RMK modules, the RMK can scale to embedded devices, e.g., smart phones, and enhance reliability without major change to module design and implementation. Furthermore, RMK-based services can be extended to provide the protection against malicious tampering with the system, e.g. monitoring access to critical application data, or system resources.
CHAPTER 3

RMK MODULES

This chapter discusses research and implementation challenges faced in developing several RMK modules.

3.1. System Hang Detection Module (SHD)

The operating system is subject to hangs due to, e.g., poorly written drivers with blocking operations, and unreleased locks. Chou et al. [5] claim that 34% of kernel bugs in Linux 2.4.1 potentially lead to system hangs. The processor may be executing non-HLT instructions, or be halted by executing a HLT instruction during a system hang. The System Hang Detection module (SHD) provides a low-latency detection of and recovery from system hangs.

An external hardware watchdog can be used to force a system reboot whenever the watchdog is not reset within a predefined timeout interval. In this case, however, one cannot determine whether the OS has crashed/hung, or the heartbeat process, designated to reset the watchdog periodically, has crashed/hung. In either situation, the system reboot is initiated, although in the latter case, a system reboot might be not necessary. Also, the detection latency might be significant because the timeout interval for resetting the watchdog timer is usually a matter of seconds (to reduce false alarms and the time overhead of the heartbeat).

OS kernel instrumentation and watchdog timer modifications are required to reduce latency in

---

8 For example, a device driver, after interrupt is disabled, performs a blocking operation to acquire a lock that is held by another process. As the blocking mutex-acquisition operation is performed in the kernel, the system hangs.
detecting system hangs and to avoid false positives (i.e., unnecessary OS reboot in the case of a heartbeat process crash/hang). For example, one could instrument the OS scheduler to send a heartbeat message to the watchdog on every context switch. If no heartbeat arrives within a certain time interval (i.e., the operating system does not schedule processes), a system hang is declared.

The approach in this thesis enables a low-latency, transparent OS-hang/crash detection. The detection mechanism is designed and implemented as a light-weight kernel driver; and hence, it does not require instrumentation of the kernel code and is fully transparent to the system.

The underlying fault model for a system hang is that an operating system in a hang state does not relinquish the processor and does not schedule any processes. Based on this fault model, system hang detection can be constructed as follows.

1. Use OS-level counters to separately track instructions executed by the application processes and the operating system.

2. Observe periodically (e.g., on each context switch) whether the instruction count in each counter changes between the consecutive readings.

3. If the contents of the OS counter continue to change and the counters associated with application processes are frozen, this is a clear indication of a system hang.

Two important issues are: (i) to determine a time period during which to measure the dedicated OS- and application-level instruction counters; and (ii) to have a mechanism to continuously and accurately measure the number of instructions being executed in the selected time period.
In a system executing a set of same-priority tasks, the count of instructions executed between two consecutive context switches (continuous-execution-instruction-count) is a finite number. If the system hangs, it may not schedule processes. As a result, the instruction count grows beyond a limit. In principle, this is determined by (i) the time-slice allocated by the OS for a process between two consecutive context switches (100 ms for typical processes in Linux 2.6, and 20/40/60 ms in Windows XP), and (ii) the fact that the count of executed instructions within the time slice has a bound (calculated as a product of the CPU speed, and the time slice). However, in most cases, this bound is a large number due to the fact that a process voluntarily yields the CPU when it is waiting on resources (e.g., asynchronous I/O input), or idling for a certain time period (e.g., sleep functions or blocking synchronous I/O operations). To provide low-latency system hang detection, a much lower bound, based on profiling of system execution, can be applied for the continuous-execution-instruction-count. A history of instruction counts collected over several time slices is used to guide the estimation of the low bound.

The SHD module implements system hang detection using two counters:

- **Profiling counter**, which keeps track of the number of instructions executed in the current time slice by a process; and

- **Checking counter**, which maintains a “check value” that is the running count of the maximum number of instructions executed in a time-slice, and obtained over a sliding window of about 50 time-slices.

Figure 6 illustrates the principle the SHD applies for system hang detection. The horizontal axis represents instruction-stream execution on the processor. The dashed and solid lines in the figure illustrate the evolution of the values in the two counters as a function of instruction execution. The
profiling counter (the dashed line) increments as instructions execute for each process during a time-slice. When a context switch occurs, the counter value is recorded by the SHD, and the counter is reset to zero to prepare for counting the instructions for the next scheduled process.

![Diagram showing system hang detection using instruction counting]

**Figure 6: System Hang Detection Using Instruction Counting**

The checking counter (the solid line) detects system hangs. It is set to an initial value, the *check value*, after each context switch, and it decrements with instruction execution. The check value is estimated according to the profiled count history, as discussed above. One can see from Figure 6 that, when a higher value is recorded by the profiling counter, the check value increases. The instruction counts recorded over a sliding window of 50 recent time-slices are used to compute the check value that is to be loaded into the checking counter.

Recall that the fault model for system hang detection assumes that the OS continues to execute instructions without relinquishing control to application processes, i.e., the context switch is no longer invoked. Thus, the checking counter is not reset, and finally reaches zero (indicated as a bell alarm in Figure 6). At that time, the counter raises a performance counter overflow interrupt.
(also known as performance monitor interrupt, or PMI) to the APIC, which then issues an NMI (non-maskable interrupt) to the processor. The NMI handler initiates system reboot.\footnote{More advanced, and sophisticated system recovery can be designed. For example, the erroneous device driver can be localized, uninstalled, and reinstalled. The involved application processes are recovered accordingly.} The profiling-based checking makes the detection low-latency while adapting to changes in the system execution.

3.1.1. Issues

**Hangs due to a halted processor.** Recall that, according to our earlier definition, the OS in a hang state indefinitely executes instructions without relinquishing the processor. In addition to this fault model, we consider another case of system hang in which the OS halts the processor while waiting for a response from a blocking I/O operation\footnote{One may expect to exploit the timestamp counter to detect indefinite execution. But the large timeout value results in long-latency hang detection.} that fails for some reason; or the OS executes a halt instruction due, for example, to an error in a control flow of a program.

When the processor is in the halt state, performance monitor counters do not work. However, a timestamp counter still counts clock cycles\footnote{More advanced, and sophisticated system recovery can be designed. For example, the erroneous device driver can be localized, uninstalled, and reinstalled. The involved application processes are recovered accordingly.} and can be used to detect system hangs due to a halted processor. Upon a context switch, the timestamp counter is set to a fixed timeout value (rather than to an instruction count). The counter decrements at each clock cycle and flags a system hang when it reaches zero (forcing the system reboot). The fixed timeout value for hang detection should be set sufficiently large, e.g. 5 seconds, to avoid false alarms, as blocking I/O operations (even when not in error) may take a long time.

**Check Value Determination.** The check value is naturally bounded as the product of the processor speed and the time-slice. A tighter bound is obtained as the running count of the
maximum number of instructions executed in a time-slice within a sliding window of a predefined
number of time-slices. However, if the system transitions from an idle state to a busy one, the
determined check value may be small, causing a false alarm. In this case, a default check value is
applied (the product of the processor speed, and half the time-slice) in the current prototype. Note
that a larger default value reduces false alarms, but increases hang-detection latency.

**Priority-based scheduling.** We assume that in a balanced system all tasks have the same priority.
Although Linux and Windows support priority-based scheduling, many practical systems usually
deal with equal-priority tasks. In these systems, kernel operations (e.g., system calls, and
interrupt/exception handlers) are completed within a time far less than the process time-slice
appropriately selected during the system design. Consequently, SHD is able to detect system
hangs successfully. For systems with prioritized tasks, the check values can be set by profiling
instruction counts executed in the time-slices of the high-priority tasks.

### 3.1.2. SHD Implementation

Three pins, P_SCHL (scheduler interceptor), P_APIC (advanced programmable interrupt
controller), and P_PMC (performance monitor counter), support the SHD module. To illustrate
how the SHD works, the behavior of the module in Linux, as an example, is depicted in Figure 7
(with implementation of the pins shown). The solid and dotted lines illustrate the control flow of
the processor execution. Circled numbers indicate the step order.

An application process, *app1*, enters the kernel mode to execute a system call, or to handle an
interrupt (step 1 in Figure 7). After the kernel mode processing completes, the Linux system
checks whether the time-slice assigned to *app1* is used up. If the time-slice is used up, the system
invokes the scheduler (step 2). In a typical case, when the RMK is not deployed, the scheduler
completes the context switch, and transfers the control to the next application process, *app2* (step 9, the dotted line).

Figure 7: Implementation of System Hang Detection in Linux 2.6.11 on a Pentium 4

When the RMK is deployed, the scheduler entry point (the address of the first instruction of the scheduler) is written into a breakpoint register, and the debug control register is properly configured so that whenever the instruction at the breakpoint is to be executed, a debug exception fault is raised by the processor (steps 3, 4). The DB exception interrupt handler registered in the system’s interrupt vector table points to the custom exception handler, my_DB_exp (step 5). This exception handler performs three tasks: (i) it reads, and sets hardware counters in the processor, and executes the detection algorithm (i.e., os_hang_check()); (ii) it calls the original debug exception handler (step 6); and (iii) it writes the scheduler entry point into the breakpoint register, configures the debug control register to enable repeating the same processing on the next context switch, and then returns to the scheduler\(^\text{11}\) (step 8). The third task is required because the original debug exception handler by default disables the debug exception. After returning to the scheduler,

\(^{11}\) A *do-not-retrigger* bit in a hardware register should be set here to successfully return to the scheduler. Otherwise, the debug exception fault is triggered again upon the return, resulting in infinite recursive fault triggerings.
the typical processing resumes (the next process is scheduled) (steps 9, 10). Steps 3-8 are provided in the P_SCHL pin with the support of P_DBR, and P_INTR.

Two performance counters, MSR_IQ_Counter0 and MSR_IQ_Counter2, are used for profiling and checking, respectively. The Time Stamp Counter is used to handle processor-halt cases. When a system hang occurs, MSR_IQ_Counter2 decrements to zero, and triggers a performance monitor interrupt (PMI) into the NMI generator in the local APIC. The NMI generator raises the NMI to the processor, where the interrupt is converted to the EVT_NMI event, and the registered callback function, os_hang_reboot(), reboots the system. The corresponding control flow is depicted by dashed lines in Figure 7.

3.2. Application Hang Detection Module (AHD)

Applications (or individual application threads) may hang due to incorrect function parameters, hardware faults, deadlock, or failed I/O. For example, it is reported in [7] that 6 out of 11986 experiments with bad parameters lead to hangs of POSIX functions in Linux 2.0.18. The failed thread is either continuously scheduled on the processor (as in an infinite loop), or not scheduled at all (e.g., waiting for wakeup on completion of asynchronous I/O operations). The Application Hang Detection module (AHD) transparently detects application hangs. The AHD exploits the fact that the count of instructions executed in a well-defined code block or code section can be statistically bounded [8]. If the instruction count goes outside the bound, the application hang is flagged.

A code section is a block of code with a single entry. A code section is monitored as a unit by the

---

12 Note that the problem of determining the worst case execution time or instruction count of a program or code block is, in general, not decidable, and is equivalent to the halting problem.
AHD. After a thread enters a code section, it must leave the section before entering the section again. Examples of code sections include a loop, a loop body (i.e., a single iteration of a loop), a function, and a mutex block (a code block between mutex acquisition, and release). A recursively called function is not monitored as a code section. A code section may include multiple nested sections (e.g. in a form of nested loops) and code sections may overlap. Each code section has a unique ID within a process. Multiple threads may execute the same code section simultaneously.

The AHD keeps a logical counter for every thread running in a monitored code section. Consequently, an array of counters monitor a multithreaded application, as shown in Figure 8. When a thread enters a code section, the corresponding counter starts counting instructions from zero, and the counter is called *active*; when it leaves the section, the counter stops counting, becoming *inactive*. Because the thread may be executing an instruction within multiple code sections, multiple counters for the thread may be active at the same time. The granularity of code sections can be selected to limit the number of sections, and thus avoid large overhead. Typically, 3-6 code sections are monitored for an application.

Though the AHD uses an array of logical counters for application hang detection, only one hardware performance monitor counter is used for counting. By using the hardware counter, and runtime system monitoring, the AHD detects application hangs with low overhead and low latency.
3.2.1. Issues

**Detecting an Unscheduled Thread.** A timestamp is introduced for each thread in order to record the time at which a thread has entered a given code section (*active code section*). The AHD performs a periodic check of these timestamps at a fixed interval. If a timestamp expires, the corresponding thread is flagged as hung. Similar to system hang detection, the fixed interval must be sufficiently large to avoid false alarms, e.g., 5 seconds.\(^\text{13}\) Note that a typical thread may not be scheduled when it executes a function with indefinite waiting time, such as `accept()`, `select()`, or `recv()` in TCP. To avoid false alarms, these functions should not be included in a monitored code section.

**Determining check value.** The check value for a code section is derived from the execution history of the section, similarly to the check-value determination in SHD. However, a code section may have multiple exits, and the counts of instructions for differently exiting executions may vary widely. An example is a switch-case block with a different computation task, and a different exit for each case. AHD maintains a history of instruction counts for executions

\(^\text{13}\) A thread scheduling may be delayed due to a heavy system load.
associated with each exit, and the check value for the code section is derived based on the history with the largest instruction counts. A threshold based on heuristics obtained from application profiling is applied to avoid selecting small check values derived from insignificant cases of code section execution (e.g., a web server sends a response to a client without sending web pages, because the client has an unexpired copy of the pages).

### 3.2.2. AHD Implementation

The AHD module employs services provided by four RMK pins: P_SCHL, P_SYSC (system call), P_PERI (periodical task), and P_PMC, in a way similar to the SHD implementation. AHD exploits compiler-provided information on code sections to detect application hangs. When generating binary code for a program, the compiler adds two function calls, section_enter(section id) and section_leave(section id, exit id), at the entries and exits of the monitored code sections, respectively. (The compiler may select only coarse-granularity code sections for monitoring, like the outer loops or functions.) A new system call, sys_section_action(), is added to service these two functions.

Only one hardware counter is used in implementing the counter array in Figure 8. For each monitored process, the AHD maintains three kinds of tables: a section table recording all the monitored code sections, a thread-execution table for each thread column in Figure 8, and an instruction-count history table associated with each exit of a code section. The AHD algorithm is outlined as follows:

- **Upon a context switch:**
  - Read the hardware counter for the number of instructions executed during the last time-slice, and reset the counter to zero.
  - Add the read number to the current count values for all the active sections in the thread. If the sum for a section is larger than the section’s check value, indicate a hang.
- Set the timestamp for the thread to the current system time.
- Upon an instruction retirement:
  - The hardware counter increments by 1.
- On section_enter(section id) invocation:
  - Set the active bit for the section and create related table entries if they are not present.
  - Read the hardware counter’s value \( c \), and set the current count value of the section within the thread to \(-c\) (so that the adding operation performed in the next context switch generates the correct value).
- On section_leave(section id, exit id) invocation:
  - Read the hardware counter’s value, and adds it to the current count value of the section within the thread.
  - Append the sum as a piece of count history in the path table for the section and the exit. Then compute the new check value\(^{14}\) according to the count history for the exit with the largest average instruction count.
  - Clear the active bit, and remove unused table entries.
- Periodically perform the following check at a fixed interval:
  - Check the timestamps for all the threads of the process associating with at least one active section. If a timestamp expired, indicate a hang.

3.3. Transparent Application Checkpoint Module (TAC)

Existing checkpointing schemes for individual applications usually take a snapshot of the application process image. However, as an application may have issued I/O operations that are pending in the system, and not finished by the checkpointing time, the snapshot of the process image, if restored upon a failure, may be inconsistent with the I/O status in the system. File reading/writing and network communication are examples of the I/O operations. Previous application checkpointing schemes, e.g. Plank’s libckpt [9], rely on user-level knowledge to decide when to take checkpoint for addressing consistency with I/O state, OS-issued signals, and message queues. This method requires the program developer to have enough system-level expertise to handle the consistency issue.

The TAC applies an approach similar to libckpt to incrementally checkpoint the dirty memory pages. However, in doing so, TAC: (i) exploits the OS-level knowledge (collected by RMK pins) \(^{14}\) As checking is not performed here, the sum may be larger than the current check value with the hang notification not raised.
on the application execution; and (ii) decides (transparently to the application) when and how to checkpoint so to avoid any inconsistency between the application image and the system state. In addition, synchronization between threads in a multi-thread process and consistency of process checkpoints in a distributed application can also be addressed transparently with OS-level knowledge (e.g., messages can be logged by invoking appropriate operations of the P_NET pin).

In addition to the generic approach to deciding when and how to checkpoint, checkpointing mechanisms can be customized to achieve even higher performance if the application execution characteristics are known. This section discusses how TAC addresses the consistency issue in a generic checkpointing method, and how the same approach can be employed to provide a custom application checkpointing.

### 3.3.1. Generic Checkpointing

Each application process has its kernel state for correct execution of the process; and a transparent application checkpoint must address the consistency between the recovered application process and the kernel state. For example, Figure 9 illustrates a breakdown of the address space of a process execution in Linux. The kernel state of the process includes the process info, kernel stack, memory tables, signal/IPC status, wait queues, and tables for opened files. The process info (in Figure 9) contains process attributes (process ID, schedule priority), process relations (parent, child), and user information (uid, gid). The wait queues deal with the pending asynchronous I/O operations, and process/thread synchronization. The user state of the process consists of memory pages constituting the segments of code, data, heap, stack, and dynamically loaded libraries.
The checkpoint taken by the TAC module includes the modified user memory pages and the processor state. The TAC module eliminates the need to checkpoint all kernel states of the process by taking a checkpoint only when the kernel state is in a safe point, i.e., no pending I/O, signals, or IPC messages in the context of the application process. Only the opened files table (names, and current seek offsets of all open files) and the memory table in the kernel state of the process are checkpointed. When the process is recovered upon a failure, the file table and memory table are restored in the process kernel state with no pending I/O, signals, or IPC messages; and with an empty kernel stack. Consequently, after restoring the checkpointed user memory pages, the recovered process is in a consistent state. The steps given below outline steps in the TAC algorithm for application checkpointing.

1. Periodically initiate a checkpoint by setting a chkpt_started flag to 1.
2. Check whether there is a pending I/O operation, signal, or IPC message in the target process. If TRUE, go to sleep; otherwise, go to step 6.
3. Upon completion of an I/O operation (or signal processing, or IPC message processing), check the chkpt_started flag (this is done using functionality provided by P_FILE/P_NET pins). If the flag is set, the pin raises an event EVT_IO_COMP/EVT_NET_COMP to notify the TAC module.
4. Upon initiation of an I/O operation by the process (the \texttt{P\_FILE/P\_NET} pins), check the \texttt{chkpt\_started} flag. If the flag is set, postpone the I/O operation. New IPC messages (check by the \texttt{P\_IPC} pin) are handled in similar fashion as I/O operations. New arriving signals (check by the \texttt{P\_SIG} pin) are allowed to be immediately processed.

5. Upon notification of new event \texttt{EVT\_*_COMP}, go to step 2.

6. Checkpoint the application process (there are no pending I/O, signals, or IPCs): (i) set all the user memory pages of the process as read-only (using the \texttt{P\_MEM} pin), (ii) save the names and current seek offsets of all open files, and (iii) preserve memory table and CPU state.

7. Clear the \texttt{chkpt\_started} flag and wait for the duration of the checkpoint interval to initiate the next checkpoint.

8. Upon a write to a read-only user page in the process, handle a segmentation fault signal raised by the system, and captured by the \texttt{P\_SIG} pin, which raises an event \texttt{EVT\_SIG\_SEGV} to notify the TAC; duplicate the page in backup memory, hosted by a user-level dummy process, and enable writes to the page using the \texttt{P\_MEM} pin.

**Comparison with shadow-process checkpointing.** A shadow process [10] may be forked for checkpointing purposes. The shadow process is inactive and only stores the process image. Memory pages are shared between the two processes with the copy-on-write mechanism applied. When the original process fails, the shadow process is started. The TAC has two advantages over the shadow-process method. (i) TAC preserves process states such as process ID and process parenthood. Preserving process ID is crucial for PID-aware applications like the Apache web server. (ii) The shadow-process checkpoint kills the old shadow process and forks a new one during each checkpoint. This incurs a fairly large overhead, including allocation, and deallocation of process resources, and memory (several milliseconds).

**3.3.2. Custom Checkpointing for the Apache Web Server**

When the application characteristics are taken into account, the generic checkpointing scheme can be customized to achieve better performance. We demonstrate this by deploying a custom checkpointing scheme for the Apache web server, a request-transaction-based application. Apache consists of a parent server process and multiple child server processes, as depicted in Figure 10(a) [11].
The parent server manages the pool of child servers through their process ID, and the child server performs the web request processing. After a connection from a client is accepted, the child server receives web requests from the client, processes the requests, and sends back replies. If a connection expires when there is no activity during a pre-specified period, the server waits for the next connection. A state transition diagram in Figure 10(b) depicts the execution flow of an Apache child server process.

When a child server crashes, the connection and the ongoing data transfer are broken: and the web page failure is indicated to a user. If the user initiates the request again, data must be retransmitted. This incurs a waste of network bandwidth.

The Apache child server state does not depend on the processing history. This feature leads to the custom application checkpointing scheme, which takes only two checkpoints of Apache, once for

---

15 Here we only consider processing basic web requests (e.g. static webpage retrievals). More complicated web applications, e.g. e-commerce, may invoke application servers and databases, which are beyond this dissertation’s focus.
each checkpoint throughout the entire life of the application, and logs the web request and the amount of transmitted data of the incomplete reply. During a failure recovery, one of the two checkpoints is restored with the connection preserved (the checkpoint to restore depends on what state the application is in when the failure occurs), and the logged web request is replayed. Process ID is preserved during the recovery. Because each checkpoint is taken only once throughout the application’s life, the runtime overhead of the scheme is negligible.

The details of the transparent custom checkpointing scheme are illustrated in Figure 10(b). Checkpoint 0 is taken the first time the mutex is acquired in state 0; checkpoint 1 is taken the first time a request is received in state 2. When the application fails in state 0 or state 1, checkpoint 0 is restored, and the mutex is released if it is still held. When the application fails in states 2, 3, or 4, checkpoint 1 is restored, and the logged request, if there is one, is replayed. During a reply, the part of the reply already sent to the client is not retransmitted.

**Implementation.** Four pins are employed in the custom checkpointing mechanism for Apache: P_MEM (memory image dumping/restoration), P_FILE (recording the information of opened files), P_NET (intercepting network operations for taking checkpoint 1, logging request, replaying request, and preventing transmission of data already sent), and P_SYSC (system calls for mutex manipulation, including the hook for taking checkpoint 0). The checkpointing, and recovery are performed without any instrumentation to the application by employing the operations/events provided by these pins.

**3.4. Experimental Evaluation**

The RMK is implemented as a loadable kernel module in Linux 2.6.11, and as a device driver in Windows XP Professional SP2, on a Pentium 4 processor (1.5 GHz). The RMK pins are
implemented as part of the RMK driver by taking advantage of the kernel functions, and variables exported from the operating systems, as well as the monitoring/debugging capability of the modern processors. Experiments are conducted on the two RMK implementations to evaluate its effectiveness, and performance overhead. NFTAPE [12], a software toolset, is used to launch fault injections, and assess the effectiveness of individual RMK modules in terms of their error coverage. We mainly present the experiment results for the RMK on Linux here because the availability of Linux source code allows us to instrument the Linux kernel for in-depth observation of the system behavior during experiments.

3.4.1. Evaluation of System Hang Detection

**Experiment setup.** To assess effectiveness of the system hang detection, we need to create operational conditions that are likely to lead to system hangs. Toward this goal, a victim device driver (CRC-driver), which calculates cyclic redundancy code, is implemented, and attached to the operating system. At runtime, faults are injected into the CRC-driver to induce system hangs. A synthetic application invokes the driver to compute CRC on a selected data set, and to generate sufficient system load to enable activation of errors injected into the CRC-driver. The results from the experiments are collected for offline analysis.

**Outcome categories.** Outcomes from error injection experiments are classified according to the categories given in Table 3.
Table 3: Outcome Categories

<table>
<thead>
<tr>
<th>Outcome Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Output</td>
<td>The application produces correct output. The error may not be activated, or activated but not manifested. The OS performs normally during the experiment period (15 s).</td>
</tr>
<tr>
<td>Fail Silence Violation</td>
<td>The application produces incorrect output. The OS performs normally during the experiment period (15 s).</td>
</tr>
<tr>
<td>Kernel Exception</td>
<td>The application terminates abnormally. The OS raises an exception: invalid kernel paging request, kernel NULL dereference, general protection, invalid operand, or others.</td>
</tr>
<tr>
<td>Silent Hang</td>
<td>The application does not complete. The system hangs without reporting any exception.</td>
</tr>
</tbody>
</table>

**Results.** Table 4 gives the distribution of fault injection outcomes. Strictly speaking, as errors propagate within the system, a single fault may trigger multiple exceptions; the first observed exception or outcome is reported in Table 4.

Table 4 indicates that 9.0% of injected errors lead to system hangs, and that the SHD (System Hang Detection) module detects all hangs observed in our experiments. Table 5 gives five examples of error propagations leading to system hangs. We can see that a propagated error may cause different kinds of kernel exceptions (Kernel NULL Dereference, and Invalid Kernel Paging Request are caused in example 3). Furthermore, critical data structures in the OS can get corrupted due to the error propagation (*Global Descriptor Table (GDT)*, and *Task Stack Segment (TSS)* in the system are corrupted in examples 4, and 5). Surprisingly, kernel exceptions can be triggered repeatedly (e.g., spin_lock_unreleased is reported more than 30 times before GDT is corrupted in example 5).
Table 4: Error Injections in the Victim Driver (1346 Experiments) for Linux

<table>
<thead>
<tr>
<th>First Observed Failure Outcome</th>
<th>Number of Manifestations</th>
<th>Number of Hangs</th>
<th>Number of Detected Hangs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Output</td>
<td>391 (29.0%)</td>
<td>1*</td>
<td>1</td>
</tr>
<tr>
<td>Fail Silence Violation</td>
<td>380 (28.2%)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kernel Exception: Invalid Kernel Paging Request</td>
<td>249 (18.5%)</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Kernel Exception: Kernel NULL Dereference</td>
<td>93 (6.9%)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Kernel Exception: General Protection</td>
<td>66 (4.9%)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kernel Exception: Invalid Operand</td>
<td>62 (4.6%)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kernel Exception: Other</td>
<td>15 (1.1%)</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Silent Hang</td>
<td>90 (6.7%)</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>Total Failed</td>
<td>956** (71.0%)</td>
<td>121 (9.0%)</td>
<td>121</td>
</tr>
</tbody>
</table>

*In this experiment, the application produces correct results; however, the error propagates, and causes an invalid kernel paging request, which further leads to the system hang.

**This number includes the case marked with *.

Table 5: Examples of Error Propagations Leading to System Hangs

<table>
<thead>
<tr>
<th>Example</th>
<th>First Observed Failure Outcome</th>
<th>Failure Propagation Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Invalid Kernel Paging Request</td>
<td>Invalid kernel paging request -&gt; Invalid kernel paging request -&gt; Kernel panic -&gt; hang detected</td>
</tr>
<tr>
<td>2</td>
<td>Invalid Kernel Paging Request</td>
<td>Invalid kernel paging request -&gt; Kernel NULL dereference -&gt; hang detected</td>
</tr>
<tr>
<td>3</td>
<td>Kernel NULL Dereference</td>
<td>Kernel NULL dereference -&gt; Invalid kernel paging request -&gt; Invalid kernel paging request -&gt; Kernel panic -&gt; hang detected</td>
</tr>
<tr>
<td>4</td>
<td>Other</td>
<td>gdt double fault -&gt; tss double fault -&gt; hang detected</td>
</tr>
<tr>
<td>5</td>
<td>Other</td>
<td>spin_lock unreleased -&gt; spin_lock unreleased -&gt;...-&gt; spin_lock_unreleased -&gt; gdt double fault -&gt; tss double fault -&gt; hang detected</td>
</tr>
</tbody>
</table>

SHD detects a hang after a predetermined number of instructions executed without a context switch, then reboots the system. Therefore, it is not known how long after the hang detection the system remains in a non-operational state. Six experiments randomly selected from the 90 silent hangs are analyzed to determine what happened to the system after the hang detection. In two of the six experiments, the system hangs for 2 seconds, and then reports an invalid kernel paging
request; in another two, the system hangs for 10+ seconds before another failure is reported; and in the remaining two experiments, the system hangs for a time longer than the observation period (4 minutes). The analysis results show that, without SHD, the system undergoes a long period of invalid execution. It may finally trigger a kernel exception, or not trigger an exception but continue hanging. This study also indicates that, although the OS provides checking for erroneous instruction executions (in 455 of 1346, or 33.8% of the experiments, the injected errors are detected by the system itself), the SHD of RMK provides a complementary solution for detecting erroneous instruction executions with a bounded latency.

**Experiment results on windows.** Experiments with the same setup described above are conducted for the RMK implementation on Windows XP, and the results are reported in Table 6. Note that there is a difference between the terminology used in Table 3 and Table 6 to define outcome categories. As we have access to Linux source code, we instrument the Linux system to study error propagation (see Table 5), and the terminology in Table 3 is based on inside knowledge of kernel behavior (e.g. *kernel exception*). For the Windows system, analysis of error propagation is not possible due to the lack of the access to the Windows source code. Therefore, failure outcome categories are defined slightly different from those shown in Table 3 (there is no *kernel exception*, and *system hang* replaces the *silent hang*).

**Table 6: Error Injections in the Victim Driver (802 Experiments) for Windows XP**

<table>
<thead>
<tr>
<th>Outcome Category</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Output</td>
<td>218 (27.2%)</td>
</tr>
<tr>
<td>Fail Silence Violation</td>
<td>94 (11.7%)</td>
</tr>
<tr>
<td>System Crash</td>
<td>476 (59.4%)</td>
</tr>
<tr>
<td>System Hang</td>
<td>14 (1.7%)</td>
</tr>
</tbody>
</table>

Most of the experiments (59.4%) result in system crash (i.e. the Windows system reboots), 11.7% of the experiments result in fail silence violation, and system hangs are resulted in only 1.7% of
the experiments. Compared with the experiment result in Linux (Table 4), there are many more system crashes while less system hangs. This difference is because the default behavior of kernel exception handlers in Windows reboots the system, while the default behavior of kernel exception handlers in Linux allows the system to continue, which can lead to a system hang.

3.4.2. Evaluation of Application Hang Detection

Experiment setup. The web server Apache 2.0.55 on Linux is the target application for fault injection to evaluate the AHD (Application Hang Detection). A website consisting of hundreds of pages and files is set up on the target machine. A web client launches multiple requests from another machine. Because the application consists of tens of thousands of lines of code, fault injection is conducted into the main loop of the server (target code), which accepts web requests, and sends back replies (illustrated in Figure 10(b)). Three code sections are identified in the target code: initialization (state 0, and transition to state 1 in Figure 10(b)), connection acceptance (state 1, and transition to state 2), and request processing (states 2, 3, 4). The instruction counts of 50 recent executions are used to determine the check value for each code section.

Results. Three fault injection campaigns, listed in Table 7, are launched to evaluate the AHD. In each experiment of the three campaigns, the AHD is trained with 50 web requests to learn the initial check value. The threshold for determining the check value in all the three campaigns is selected as 1.5 million instructions (equivalent to a 1ms-duration execution of the processor) to avoid small check values derived from trivial cases of code-section execution.
Bit flips are injected into the target code in Campaign 1. Table 7 shows that the application is very likely to crash (99.6%) in the presence of bit errors. Only seven of the injections lead to hangs. As seven hangs are not sufficient to evaluate the detection coverage of the AHD, hangs are explicitly injected into the target code in the second campaign. A hang is injected by activating an infinite loop embedded in the application. The AHD module detects all hangs injected in our experiments.

Campaign 3 was launched to measure false positives, and hence no faults are injected in this campaign. In the 2000 experiments, only one raises a false positive. The false positive was raised because a very large file was requested following a number of requests for small pages/files. The transmission of this large file requires many more instructions be executed within a code section (2726433 instructions are executed for the request for processing, while the largest instruction count for the previous 50 requests for processing is 449493). The false positive may be avoided with a longer monitoring history (e.g. 100 requests), or a higher threshold.16

3.4.3. Performance Overhead of RMK, SHD, and AHD

**Experiment setup.** Experiments are conducted to study the performance overhead of the RMK. Two machines are used in the experiments: one as the web server, and the other as the web client. The Apache 2.0.55 runs on the server machine. The web clients on the client machine are launched by WebStone [13], a benchmark program for web servers. WebStone creates a load on a web server by simulating the activity of multiple web clients on one or more machines. In our

---

16 Prior knowledge of the size of the requested file/page can be exploited to reduce false positives.
experiments, WebStone starts 20-100 simulated web clients on the client machine. Three sets of experiments are carried out: baseline (RMK not deployed), RMK+SHD (RMK with the SHD module deployed on the server machine), and RMK+AHD (RMK with the AHD module deployed to detect hangs of the Apache server). No fault injection is performed in the experiments.

**Results.** Figure 11 gives the server throughput (amount of data transmitted per second) and average response time for a request as a function of number of clients. One can see that the throughput does not change with the number of clients, and the average response time increases linearly with the number of clients. This is because, the web clients launched by WebStone send requests to the server continuously, and the server processes the requests at its full throughput, which is a fixed value. Because the server’s processing capability is fixed, the request response time gets larger when more clients send requests.

Figure 11 shows that the performance overheads of RMK, SHD, and AHD are very small. The actual throughput (with the 95% confidence interval) measured for the three scenarios in Figure 11(a) is 6.51±0.04 ms for the baseline, 6.51±0.07 for RMK+SHD, and 6.47±0.02 for RMK+AHD.

![Graph](image1)

(a) Throughputs of the web server

(b) Average response time for a request

**Figure 11: Performance Overheads of RMK and SHD/AHD Modules**
The overhead of SHD is negligible because instruction counting is performed in hardware, and the context switch frequency is not high. AHD incurs an overhead of about 0.6% of the baseline performance. The AHD overhead is due to (i) system call invocations at the code section entry/exit points, (ii) updates of count values at context switches, and (iii) periodic timestamp checks. Concerning the low frequencies of context switches and timestamp checks, (ii) and (iii) incur negligible overheads. If code sections are entered and exited frequently, the overhead in (i) is noticeable, but still small.

3.4.4. Performance Overhead of TAC

Three applications are used to evaluate the performance overhead of the generic checkpointing in TAC (Transparent Application Checkpoint) module: (i) gzip, a SPEC2000 benchmark, (ii) ns2, a popular network simulator, and (iii) Apache, a web server. Because the TAC module aims at providing high-efficiency checkpointing for computation-intensive applications, gzip and ns2 are selected as target applications in the experiments. Apache is included to contrast the generic checkpointing scheme with a custom checkpointing approach. Results show that TAC incremental checkpointing has a 0.1% performance overhead, and gains a 91% performance improvement against the memory-dump approach (dumping all process pages to backup memory) for a large-memory application. These results are consistent with the measurements in libckpt [9] (85% of overhead reduction), and libFT [14] (0.1% performance overhead).

3.4.4.1. Results for Generic Checkpointing

As an in-memory checkpointing scheme, the TAC has a relatively small performance overhead, and more frequent checkpointing can be applied to avoid a large loss of work due to application failures (the checkpoint interval for the experiments is 2 seconds). Table 8 summarizes the TAC
performance overheads (the 95% confidence interval) and compares the results with the performance of the memory-dump approach.

Table 8: TAC Performance Overhead During a Checkpoint Interval (2 Seconds)

<table>
<thead>
<tr>
<th>Application</th>
<th>Data Pages, $n$</th>
<th>Written Pages, $p$ *</th>
<th>TAC Checkpoint Overhead (ms)</th>
<th>Memory-Dump Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Setting pages as read-only, $c$</td>
<td>Handling a page fault, $s+r$</td>
<td>Overall, $TAC_{ov}hd$</td>
</tr>
<tr>
<td>gzip</td>
<td>111 (data: 2, bss: 83, heap: 0, e.v.+arg+stack: 21, lib data: 5)</td>
<td>29±6 (data: 1, bss: 27, heap: 0, stack: 1, lib data: 0)</td>
<td>0.0083±0.00/43</td>
<td>0.0344±0.00/25</td>
</tr>
<tr>
<td>Apache</td>
<td>321 (data: 7, bss: 3, heap: 203, e.v.+arg+stack: 21, lib data: 87)</td>
<td>32±8 (data: 5, bss: 1, heap: 17, stack: 3, lib data: 6)</td>
<td>0.0191±0.00/36</td>
<td>0.0344±0.00/25</td>
</tr>
<tr>
<td>ns2</td>
<td>1426 (data: 158, bss: 8, heap: 1213, e.v.+arg+stack: 21, lib data: 26)</td>
<td>34±6 (data: 1, bss: 1, heap: 31, stack: 1, lib data: 0)</td>
<td>0.1182±0.00/34</td>
<td>0.0344±0.00/25</td>
</tr>
</tbody>
</table>

*The breakdown information of written pages listed in the column is collected from a sample page-access trail in a checkpoint interval.

Table 8 lists the numbers of data pages, and pages written during a checkpoint interval for the three applications. The breakdown information (in parenthesis) of pages accessed during a checkpoint interval provides insights into the executions of the applications. For example, gzip accesses a total of 111 user-space data pages, including initialized static data (data), uninitialized static data (bss), dynamic data (heap), stack, and data of shared libraries. Apache uses 321 pages, and ns2 uses 1426 pages, the largest number among the three benchmarks. This is because gzip uses a streaming mode in data compression (read a block – compress – write result – read next block – …), and a small amount of fixed-size memory is sufficient (gzip has no heap). In the
conducted experiments, ns2 simulates a network transmission protocol in a complex network configuration and explores a large state space in the simulation, so the memory requirement is high.

Due to space locality, the number of memory pages written by an application within a short period is often far less than the total page number. In our experiments, around 30 pages are written in the three applications during a checkpoint interval, though the total page numbers range from several hundred to several thousand.

TAC checkpointing involves two activities: (i) periodically setting the entire process memory to read-only (the associated overhead is denoted as $c$ in Table 8); and (ii) upon a page fault, raising the page fault exception (the overhead denoted as $s$), granting write access to the targeted page, and replicating the page (denoted as $r$). The TAC performance overhead is computed by adding the overheads of the two activities. Let $p$ denote the number of page faults in a checkpoint interval. Then the TAC checkpoint overhead during a checkpoint interval is

$$TAC\_ovhd = c + (s+r)*p.$$ 

The measurement in the experiments shows $s = 24.55 \pm 2.02 \, \mu s$, and $r = 9.86 \pm 0.46 \, \mu s$ (with the 95% confidence level). Because $s$ measures triggering a page fault as well as delivering a signal, and $r$ measures copying one page and granting write permission for the page, $s$ and $r$ have the same values for the three benchmark applications. The values of $c$ are different for the three applications because different numbers of memory pages (denoted as $n$) are set as read-only. The TAC\_ovhd values listed in Table 9 are computed using the corresponding average values of $c$, $s$, $r$, and $p$. 

48
Comparison with shadow-process checkpointing. Both TAC checkpointing and shadow-process checkpointing have overheads computed by \( c+(s+r)p \), but the \( c \) in shadow-process checkpointing is much larger, as a process forking and a process termination are included (for the Apache application, the overhead associated with the process forking and termination is 1.75 ± 0.05ms, much larger than the 0.019 ms in TAC).

Comparison with Memory-Dump Checkpointing. The overall overheads for the three benchmark applications, 1.01 ms, 1.12 ms and 1.29 ms, are less than 0.1% of the checkpoint interval (2 seconds), and justify the selection of checkpoint intervals of several seconds. Table 8 (the rightmost column) provides comparison of the TAC checkpoint overhead with the overhead of the memory-dump method. One can see from the table that TAC checkpointing has a greater performance improvement than memory-dump checkpointing, especially for applications with large memory (1.29 ms versus 15.08 ms, a 91% improvement for ns2). This is because TAC effectively exploits the space locality of memory accesses performed by applications.

3.4.4.2. Experiment Results for Custom Checkpointing for Apache

Experiments are conducted to evaluate the custom checkpointing for Apache with the support of TAC. The performances of three checkpoint/recovery approaches are compared in the experiments: (i) a simple restart solution, (ii) memory-dump checkpointing, and (iii) custom checkpointing for Apache. The comparison in terms of the runtime checkpoint overhead and recovery time is summarized in Table 9 (with a 95% confidence level).
**Table 9: Comparison of Three Checkpoint/Recovery Approaches**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
<th>Checkpoint/Logging Overhead (ms)</th>
<th>Recovery Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Restart</td>
<td>The web server is restarted upon failure. No checkpointing.</td>
<td>N/A</td>
<td>42.02 ± 0.13</td>
</tr>
<tr>
<td>Memory-Dump</td>
<td>The process image is dumped to backup memory.</td>
<td>3.10 ± 0.15</td>
<td>4.05 ± 0.08</td>
</tr>
<tr>
<td>Custom Checkpointing for Apache in TAC</td>
<td>Two checkpoints are taken only once during execution. Information logging is performed at runtime.</td>
<td>Not measurable*</td>
<td>4.07 ± 0.07</td>
</tr>
</tbody>
</table>

* The time-measuring tool provided by the system is not able to measure so small an overhead, and it reports 0.

From Table 9, one can see that if there is no checkpointing provided for Apache, the recovery time is large (42.02 ms), which greatly degrades web service availability. The recovery times for memory-dump checkpointing and custom Apache checkpointing are similar (around 4 ms), as both recover the application from in-memory checkpoints. But the runtime overhead of the custom Apache checkpointing is very small (not measurable in the system).
CHAPTER 4

FORMALIZING SYSTEM BEHAVIOR FOR EVALUATING SHD

4.1. Introduction

Operating systems are subject to hangs due to, e.g., poorly written drivers with blocking operations, and unreleased locks. A number of techniques have been proposed for low-latency and low-cost hang detection, e.g., SHD in RMK [15], KHM [16], hardware watchdog [17], and Linux NMI watchdog [18]. These techniques are either not systematically evaluated ([16], [17], [18]) or are evaluated through injection of memory or processor errors ([15]) in the program execution. Random fault injection enables assessing coverage and latency figures for the built-in hang detection mechanisms. However, due to its statistical nature, even extensive error injection campaigns cannot guarantee uncovering all hang cases that escape detection.

This chapter proposes a formal approach to verify the coverage of a system hang detector, SHD, in particular to expose the system hang scenarios that escape detection. The Linux system with the integrated SHD is abstracted using a formal model developed for the purpose of evaluating the SHD. The state of the model is described in terms of the computation performed by the system (i.e. execution of relevant hardware/software components such as hardware timer, user threads, system calls, and interrupts), rather than in terms of its structural components (e.g. the cache or the register values). As a result, the system behavior in the model is represented as an execution flow of interacting and interleaving execution of the software components. The execution flow is modeled at a granularity of a basic computation unit (BCU), i.e., uninterrupted execution of code
in a specific computation context. In this sense, the user-mode execution of the program between any consecutive interruptions (either due to a system call or an interrupt) is regarded as a BCU. Note that a BCU is not determined by the source code, but rather by the dynamic execution. For example, the user code execution between two consecutive system calls is regarded as one BCU if the execution is not interrupted by an interrupt; and the same user code execution, if interrupted by an interrupt in the middle, is regarded as including two BCUs. The unique part of this work is that, at this stated level of abstraction, we are able to enumerate all interleaved execution corresponding to actual system behavior in so far as system hangs are concerned.

Specifically, our modeling approach consists of several steps:

- Abstraction of the SHD and the hardware/software components in the target real-world system that directly interface with or are invoked by the SHD. A set of hierarchical state machines (derived based on the analysis of the source code semantics of the target system) is used to model the behavior of a software component at the granularity of a BCU.

- Transformation of the model into a formal language representation to enable reasoning about the system behavior.

- Verification of the system hang detector and identification of corner cases in system hangs that escape detection. The system hang is interpreted as violation of a liveness property of the model. Then explicit-state model checking exhaustively explores execution scenarios, including cases that lead to system hangs.

This chapter contributes with: (i) an abstraction and a formal model of a Linux system with an integrated system hang detector, (ii) a proof that the abstraction neither misses system hang
scenarios in the real-world system, nor incurs false positives, (iii) an algorithm for deriving all possible system execution flows by combining the behaviors of the system components modeled in hierarchical state machines, (iv) implementation of the model using the Maude formal language and the model checker [19], and (v) assessment of the coverage of the target system hang detector and identification (characterization) of execution scenarios that lead to hangs and escape detection.

4.2. Related Work

A number of models formulate system behavior as state machines in the literature and industry. An abstract state machine (ASM) is often applied for this purpose.\(^\text{17}\) An ASM state is a step in the execution of a program/system, rather than a point in the state space. Every algorithm (the verification target) is emulated step-for-step by an appropriate ASM. Because ASMs model algorithms at arbitrary levels of abstraction, this approach has been frequently applied to verify software and hardware systems [20], [21], [22]. Besides ASM, SPIN models processes in distributed systems as individual state machines to detect design errors in distributed protocols [23].

There are also approaches which verify system properties using functional language. For example, in [24], OS operations are represented in a functional language suited for manual proof of security properties. [25] represents an abstract OS using a functional language to study properties of system partitioning (i.e., address space, time-sharing) for providing error isolation.

In addition to modeling system behavior based on semantics, modeling and verification are also conducted on the actual code of programs, libraries, and APIs. Symbolic execution falls into this

\(^{17}\) More information about abstract state machines can be found at http://www.eecs.umich.edu/gasm/
category. SLAM [26] verifies the source code of device drivers for Windows by first transforming driver code into Boolean programs (programs manipulating only Boolean variables) and then applying model checking for the Boolean programs. In [27], [28], the L4 microkernel API is verified to ensure consistency among the API functions and the correctness of the API implementation. MOP [29] instruments programs with formal verification code for runtime program verification. A formal model is constructed for the target program, state transition in the formal model is embedded in the program source code, and properties of the formal model are checked upon state transition. JavaFAN [30] symbolically executes Java programs while verifying behavior properties. Pointer taintedness analysis [31] finds security vulnerabilities in library source code by propagating an attribute of pointer taintedness along data flows of the library functions.

4.3. System Hang Detector

The target technique, System Hang Detector (SHD), which is evaluated through formal methods in this chapter, enables low-latency, transparent OS-hang detection; it has been introduced in [15]. SHD is designed and implemented as a light-weight kernel driver. Compared with other system hang detection techniques, SHD requires neither extra hardware devices, nor instrumentation of the kernel source code.

The underlying fault model for a system hang assumed in SHD is that an operating system in a hang state does not relinquish the processor and does not schedule any processes. The detection principle underlying the SHD is based on a generic observation that in a system executing a set of equal-priority tasks, the number of instructions executed between two consecutive context switches is finite (or bounded). A system hang causes the instruction count to grow beyond the
predetermined (or measured) bound.

The SHD uses a hardware performance counter (a register available in most current generation processors) to count the number of instructions executed by the processor. The SHD implementation intercepts the scheduler by replacing the first instruction of the scheduler with a jump instruction to redirect the execution to the SHD code. Upon a context switch, the SHD code resets the hardware performance counter to a preset value (in-depth discussion on determining the value can be found in [15]). The counter decrements as an instruction executes.

During normal behavior of the system, the counter is regularly reset. Upon a system hang, the OS continues to execute instructions without relinquishing control to application processes, i.e., the context switch is no longer invoked. Thus, the counter is not reset, and it finally reaches zero. At that moment, the counter raises a Non-Maskable Interrupt (NMI) to the processor, which indicates a system hang.

4.4. System Abstraction and Modeling

To evaluate the SHD, an abstract model of a typical Linux system is created to thoroughly exercise all execution scenarios that may lead to hangs. The goal is to expose cases (i.e., hang scenarios) that escape detection by the detector. Our system model abstracts the basic hardware (e.g., timer, hardware counter) and software (e.g., processes/threads) components of a typical uniprocessor system (with the SHD integrated) and enables: (i) capturing behavior of these components to model possible execution scenarios that may lead to system hangs, and (ii) evaluating the hang detection coverage of SHD.
**Modeled components.** A typical system may concurrently run hundreds of processes/threads that perform a variety of operations (e.g., system calls, kernel-level tasklets, and invocation of device drivers) while potentially accessing multiple hardware devices (e.g., hard drives, or network adapters).

To reduce the complexity of our model while still achieving the objective of formally evaluating the SHD technique, we model only the behavior of the components that directly interact with or are invoked by the SHD. Here “directly interact with/invoke” means that in an execution flow of the system behavior, the BCUs of these components execute either immediately before (i.e., the binary code immediately preceding the SHD code) or immediately after (i.e., the binary code immediately following the SHD code) the invocation of the SHD. These components constitute the system interface to the SHD.

By modeling only the behavior of the components interfacing with the SHD, we essentially assume that any system hang manifests at these interfaces and leads to detectable (or observable) changes in the behavior of the modeled component(s). Consequently, observing and analyzing the behavior of these components is sufficient to expose hang cases that escape detection. The assumption is proved in Section 4.4.2.

### 4.4.1. Components in the Modeled System

Source code is needed for obtaining an accurate model of the system behavior semantics. Analysis of the Linux source code and the code for the SHD implementation identifies the software and hardware components interfacing with SHD (note that the kernel source code is not needed to implement and deploy the SHD in the real system).
The scheduler and the hardware counter. The SHD implementation intercepts the scheduler. Upon a context switch, the SHD resets a hardware performance counter to a preset value. The hardware counter decrements as instructions execute and reaches zero after a system hang occurs.

Return from kernel to user. To model the behavior of the scheduler and the hardware counter, we need to find the components that interface with the scheduler. The scheduler (in Linux 2.6) is invoked only when the system execution flow leaves kernel mode for user mode. In Linux implementation, a snippet of kernel code performs the return to user mode. This code snippet processes pending signals, IPCs, and I/O responses from device drivers for the current user thread and checks the time slice for the current user thread to determine whether a context switch should be scheduled. We only consider the time slice checking in the code snippet and ignore the other operations, as they are not directly involved.

Interrupt handling and system calls. As we model the execution flow which transitions from kernel to user mode, we also need to model the execution flow which transitions from user to kernel mode. As a result, interrupt handling, system calls, and exception handling should be modeled (in our model, exception handling is consolidated with interrupt handling because an exception can be regarded as a special type of interrupt).

Timer. We need to model instruction counting by the hardware performance counter. As hundreds of millions of instructions are executed during a time slice in modern processors, it is not feasible to model counting of every single instruction. Instead, we model the time corresponding to instructions’ execution at the granularity of the timer interrupt interval. Therefore, the timer is also included in our model.
User threads. During system hang scenarios none of the user threads makes further progress. We model execution of user threads for observing such progress to determine whether the system is hung or not.

In summary, our abstract system model includes the following components: scheduler, hardware counter, return from kernel to user mode, system calls, interrupt handlers, timer, and user threads. Among these components, the hardware counter and timer are hardware components, and the others are software components executed on the processor.

4.4.2. Assumption Proof

A system hang manifests as lack of progress in executing the user code. In our model, this corresponds to computation being trapped in a BCU or a set of BCUs not part of the user code, e.g., a system call or an interrupt. A livelock is an example of computation being trapped in a set of BCUs. The two lemmas below reflect the assumptions made in Section 4.4. Proof sketches are given here.

Lemma I. Any system hang in the target real-world system manifests as a hang in the modeled system.

Proof Sketch. Three generic types of system hang scenarios are possible in the target real-world system.

Type I. The execution flow is trapped in a BCU (or BCUs) of a component (or components) which are also in the modeled system. In this case, the system hang manifests in the modeled system.

Type II. The execution flow is trapped in a BCU (or BCUs) of a component (or components) which are not in the modeled system. Let c be the component in the modeled system whose BCU is the last one executed before the system execution flow gets trapped in a component (or components) not in the modeled system. In this scenario, the hang is observable as computation being trapped in component c and hence manifests as a hang in the modeled system.

Type III. The execution flow is trapped in executing BCUs of components of which some are also in the modeled system while others are not. By excluding from the execution flow the BCUs of
the components not modeled, the resulted execution flow is of Type I, and hence the system hang manifests in the modeled system.

Lemma II. Any system hang in the modeled system manifests as a system hang in the target real-world system.

Proof Sketch. Two generic types of system hang scenarios are possible in the modeled system.

Type I. The execution flow is trapped in a BCU of a component in the modeled system. As the BCU of the component is also in the target real-world system, the system hang is observable in the real system.

Type II. The execution flow is trapped in multiple BCUs of a component (or components) in the modeled system. Then the trapped part of the execution flow corresponds to a number of execution flows in the real system, including BCUs of components not modeled. For each of these execution flows, user code is never executed (otherwise the BCUs as part of user code are also executed in the modeled system, and this is not a system hang). So the system hang is observable in the real system.

4.5. Modeling System Behavior

We employ finite state machines (FSMs) to model the system behavior and enable studying the SHD’s capability of detecting system hangs. Before we dive into the details of the model, the key features of our model are summarized here: (i) the hardware components – the hardware counter, the timer, and the processor – are modeled in three separate FSMs to accommodate inherent parallelism among hardware components. Modeling the parallelism among physical processing units makes our approach applicable to multi-core, multiprocessor, and virtual machine based computing systems; (ii) the behavior of software components on the processor is modeled as a set of hierarchical FSMs based on analysis of the source code semantics; and (iii) individual components of the same type are not differentiated from the perspective of studying system hang behavior (e.g. thread represents all threads in the system). Note that differentiation of individual components may be needed to study behavior other than system hang.
4.5.1. Modeling the Hardware Components

This subsection describes the model for the timer and the hardware counter, as illustrated in Figure 12 (dashed lines indicate events across different FSMs). The timer has only one (dummy) state, *sleep*. A transition from *sleep* to *sleep* corresponds to a subsequent timer interrupt.

The **hardware counter** is modeled to get decremented as instructions execute. Recall that we model the time corresponding to instructions’ execution at the granularity of the timer interrupt interval; i.e., the counter is decremented on an arrival of the timer interrupt. In Figure 12 (a), this is illustrated by transitioning from *state i* to *state i+1*. Without the SHD integrated into the system, the hardware counter is modeled as a state machine with a large (but finite) number of states because: (i) the counting continues all the way as instructions execute, and (ii) on an underflow (32-bit register), the counter returns back to the *state 0*.

With the SHD integrated into the system (and our model), the hardware counter is reset by SHD upon a context switch (Figure 12 (b)). This behavior is represented in the model as a transition from a given counter state, *state i*, to the initial state, *state 0*. Since the context switch may occur whenever a user thread yields the CPU voluntarily, any *state i* may transition to *state 0*. In the current version of Linux (kernel 2.6), the default time slice is 100 ms, and a timer interval is 18.7 ms. So in the normal behavior of the system, the hardware counter state is reset to *state 0* after at most 6 transitions (*ceiling*(100/18.7)).
After a preset threshold number of instructions are counted, i.e., after arrivals of a preset number of timer interrupts, the hardware counter underflows (reaches zero) and declares the system hang. This behavior is captured as the transition from a state (state \( n \) in Figure 12 (b)) to the next state \( \text{hangdetected} \), which is a final state. The preset period of time for hang detection is a parameter in the SHD, and it is usually set larger (e.g., 1.5 times of the time slice) than the time slice to guarantee that the context switch occurs at least once before the preset hang detection period finishes.

**Further model refinement.** The \( n+1 \) states (see Figure 12 (b)) in the hardware counter model can be collapsed into three states: (i) \( \text{reset} \), which corresponds to the initial state \( 0 \), (ii)
timerinterval, which depicts the state after arrival of the first timer interrupt since reset, and (iii) timeslice, which corresponds to the ending of a time slice. It is relatively straightforward to show formally that, with support of formal model checking tools, the behavior modeled using complete state machine (see Figure 12 (b)) and the behavior depicted by the refined model (see Figure 12 (c)) are equivalent in terms of their abilities to expose execution scenarios that lead to system hangs. We use the refined model in Figure 12 (c) for evaluating the SHD.

4.5.2. Modeling the Software Components

A set of hierarchical finite state machines (see Figure 13; dashed lines indicate transitions to higher-level states; dash-dot lines indicate events) is used to model the software components. These FSMs are derived by analyzing the source code semantics of the software components in the modeled system (as described in Section 4.4.1). The user thread and the scheduler are at the top of the FSM hierarchy (Figure 13 (a)). The behavior of threads is further decomposed using low-level state machines. This decomposition process continues until the granularity of a BCU is reached. For example, a user thread, thread, is modeled as interleaving execution of user code, usercode, and system call, syscall (Figure 13 (b)). The system call is further decomposed into execution of kernel code (system call service), kernelcode, and the post-service code for return from the kernel to the user mode, returntouser (Figure 13 (c)). This decomposition process continues until a BCU (uninterrupted execution of a bulk of instructions), instreexec, is reached.
Figure 13: Hierarchical Modeling for the System Behavior

A dummy state, $\epsilon$, is added in the FSMs to indicate the termination of the decomposition process of a computation scenario. For example, the $\epsilon$ state after the `returntouser` in Figure
13 (c) means that, after the execution flow returns to the user mode, the system call invocation terminates.

All of the FSMs for the software components in our model are illustrated in Figure 13. The topmost FSM is depicted in Figure 13 (a). At any time, the system is in a thread execution context or the scheduler is performing a context switch. Each thread executes for a period of a time slice unless the thread volunteers to yield the processor.

Due to space limitations here we do not go over all of the FSMs in Figure 13, but only briefly describe the FSMs in Figure 13 (d) and Figure 13 (i). The returntouser executes in the kernel mode to check whether a thread volunteers to yield the CPU, whether a time slice is used up, and invokes the scheduler accordingly. The dashed lines in Figure 13 (d) (transitions to the scheduler state) indicate that the execution flow transitions to a state in a higher-level FSM (the scheduler state is the state in Figure 13 (a)). Interrupt behavior is modeled in the FSM in Figure 13 (i). In real-world systems, an interrupt may arrive after any instruction. We apply the coarser granularity of the BCU in modeling interrupts because it is sufficient for exploring the state space of BCUs interleaving. Note that non-deterministic behavior semantics are modeled in both Figure 13 (d) and Figure 13 (i).

**Processor attributes.** Using the interrupt FSM (Figure 13 (i)), we model execution semantics where an interrupt can preempt system calls, the scheduler, or interrupt handlings. However, when the system disables an interrupt (or interrupts) some (or all) incoming interrupts are not serviced. A processor attribute, *interrupt_flag*, is defined in our model to capture this behavior semantic. Usually when an interrupt is being serviced, the interrupt routine has an option to enable
or disable interrupts. The attribute checking or setting is not shown in Figure 13, to avoid cluttering the graphical representation of FSM models.

4.5.3. Execution Flow Representation

The execution flow of the system is depicted by a flow of traversing the defined (identifiable in the model) system states. A system state is defined in the context of the BCU currently running on the processor (processor attribute values are also included in a system state, but we ignore them in the following discussion). The context associated with the BCU consists of the states of the computational components from the topmost level of FSM hierarchy to the BCU. The computational components on the path from the top FSM to the BCU form logical-expansion-to-BCU (see Figure 14), which identifies the BCU. For example, the BCU thread.usercode.instrexec indicates that the processor is executing the user-level code of a thread that is not interrupted by an interrupt or system call.

We should clarify two notions of state: (i) the system state, which is defined by the state of computation components in the logical-expansion-to-BCU, and (ii) the FSM state, which indicates a state in an FSM (such as the syscall in Figure 13 (b)). Figure 14 gives an example of computation execution flow in the case of hang-free execution scenario.

In Figure 14, each line corresponds to a system state, and the sequence of transitions from state 1 to state 15 represents an actual system execution flow, which consists of a sequence of executing BCUs. The state transitions are explained on the left of the execution flow. Both the logic expansion of software components into BCUs and the system state transitions are performed according to the FSMs in Figure 13.
In the following, the example in Figure 14 is briefly explained. In the initial state, the system is executing user-level code of a thread, say, \textit{thread1}. Then \textit{thread1} invokes a system call and resumes execution of user-level code (line 4). Further, \textit{thread1} invokes another system call, and when the system returns to the user mode (after servicing the system call), the scheduler is invoked to perform the context switch. The SHD intercepts the scheduler (to reset the hardware counter) and then returns to the scheduler to complete the context switch (lines 7 and 8). Then another thread (say, \textit{thread2}) starts to execute on the processor. During \textit{thread2}’s execution, an interrupt arrives (line 10), which is handled by the system. Soon after the return from the interrupt, the time slice for \textit{thread2} is used up, and a context switch occurs. When the scheduler is executing on the processor, an interrupt arrives (line 14) and preempts the scheduler. After the interrupt is handled and the scheduler completes the context switch, the next thread (say, \textit{thread3}) begins execution (line 16).
4.5.4. System Hang Representation

As a system hang is time-related (the system does not schedule processes for an infinitely long time), it cannot be represented by the system state transitions alone. The time information in the state machine for the hardware counter is used to represent system hangs. Figure 15 depicts an example execution flow for a system hang scenario. Both the system state and the hardware counter state are combined in the execution flow. After a thread invokes a system call, an error in the system call routine causes a system hang (line 2). When a timer interrupt arrives, it is handled by the processor (line 3). Since we use the refined model for the hardware counter (see Figure 15 (c)), the second timer interrupt transitions the hardware counter to the state \textit{timeslice}. After handling the second timer interrupt (line 5), the system returns to the system call and hangs here (line 6). The next timer interrupt transitions the hardware counter to the \textit{hangdetected} state, which means that the hardware counter reaches zero, and the SHD detects the system hang. Before we proceed with a more generic description of the derivation of the execution flow from FSMs, we will discuss few additional considerations in modeling the system.

<table>
<thead>
<tr>
<th>System state</th>
<th>HW counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. thread.usercode.instrexecl</td>
<td>reset</td>
</tr>
<tr>
<td>2. thread.syscall.kernelcode.instrexecl</td>
<td>reset</td>
</tr>
<tr>
<td>3. thread.syscall.kernelcode.instrexecl.interrupthdl.kernelcode.instrexecl</td>
<td>timerinterval</td>
</tr>
<tr>
<td>4. thread.syscall.kernelcode.instrexecl.interrupthdl.kernelcode.instrexecl</td>
<td>timerinterval</td>
</tr>
<tr>
<td>5. thread.syscall.kernelcode.instrexecl.interrupthdl.kernelcode.instrexecl</td>
<td>timeslice</td>
</tr>
<tr>
<td>6. thread.syscall.kernelcode.instrexecl.interrupthdl.kernelcode.instrexecl</td>
<td>timeslice</td>
</tr>
<tr>
<td>7. thread.syscall.kernelcode.instrexecl.interrupthdl.kernelcode.instrexecl</td>
<td>hangdetected</td>
</tr>
</tbody>
</table>

\textbf{Figure 15: A System Hang Detected by the SHD}

\textit{Representing a system hang in a formal model.} A system hang in an execution flow corresponds to a lack of progress in executing the user threads and can be interpreted as violation of a \textit{liveness property} of the system.
Hangs due to design error. Some of the system hang scenarios can be due to a design error rather than a transient error. While our approach can still identify these scenarios, the model checker cannot distinguish them from system hangs due to transient errors; manual inspection of system hangs is required to make this distinction.

Application hangs. The model-checking procedure explores all possible system execution scenarios (at the level of our modeling abstraction), including scenarios corresponding to application hangs. Our current approach cannot distinguish application hangs from normal application execution because, in both situations, user threads are continuously scheduled onto the processor and executed.

4.5.5. Deriving System Execution Flows from FSMs

System behavior, or a system execution flow, is represented as transitions between system states.

The example given in Figure 14 also illustrates how such an execution flow is derived. Two types of operations are performed to derive the execution flow: (i) logical expansion to a BCU to represent the system state (horizontal flow in Figure 14) and (ii) system state transition along the execution flow (vertical flow). Any given state is first expanded to the granularity of a BCU, and then the system state transitions to the next system state (which needs to be expanded to the BCU granularity again). The expansion and the transition are performed according to the FSMs in Figure 13. The algorithm below specifies this derivation procedure:

```
derive_control_flow(s, fsms, L)
s: the current system state
fsms: the set of FSMs
L: the current sequence of transition target
begin
  while (true) do
    expand(s, fsms, L); // expand s to granularity of BCU while maintain L accordingly;
```
transition(s, fsms, L, s'); // transition the system state s to another system state s’ by using L properly;
    s = s’;
end while
end

A loop iteration in the algorithm transitions a system state to a next one, and the entire algorithm derives an execution flow of system behavior in terms of transitions of system states according to the input FSFs. We implement the algorithm in a model checking tool, which then exhaustively derives all the possible execution flows for the system behavior.

Here we use the transition from line 1 to line 2 in Figure 14 as the example to demonstrate how the expansion and transition operations in one loop iteration of the derive_control_flow() exactly work. Figure 16 illustrates the procedure for the operations.

![Figure 16: Deriving the Flow of System State Transition from FSMs](image)

Expansion to BCU granularity. Four FSFs are applied in expanding the starting state system into the system state at the granularity of a BCU. The System state column records the computation component in different FSF levels when identifying the BCU, and the Transition target column records the target FSM state for the next transition in different levels. The dash-dot
lines in Figure 16 depict the expansion procedure, and the double-dashed lines illustrate the system state transition.

The expansion of system starts from the topmost FSM level. During step 1 in Figure 16, the thread is the current computation component in the system. After the thread finishes, the scheduler is the next component to execute in the system according to the system FSM in Figure 16 (a) (the scheduler is marked with a dot in the figure to highlight the next selected component). To reflect this, write the thread in the System state column, and write the scheduler in the Transition target column.

During step 2, the thread FSM is used to expand thread. Following a procedure similar to that in step 1, the usercode is the current computation sub-component, and the syscall is the target state for the next transition in this level. Write usercode and syscall in the two columns on the right. The expansion procedure continues until the current computation component is epsilon during step 4. In this case, write epsilon in the Transition target column. The expansion procedure terminates when BCU granularity is reached. There are two options when selecting the current computation components during step 4: interrupt_hdl or epsilon. The two options indicate whether or not an interrupt arrived during execution of the BCU. The epsilon is selected in Figure 16. Exhaustive enumeration of all of the options is performed automatically by a model checker tool. After the four steps, the expansion result is obtained by collecting the information in the two columns and represents a system state thread.usercode.instrexec and a sequence of transition targets (epsilon epsilon syscall scheduler).

System state transition. The system state transition is depicted by the double-dashed lines in Figure 16. We start with the transition target in the lowest level. It is epsilon, which means there is
no transition target in this level, and the \textit{instreexec} component completes (step (a) in Figure 16). Then we proceed to the next upper level by removing the \textit{epsilon} from the transition target and \textit{instreexec} from the system state (i.e., the two symbols linked by the arrow \textit{a} in Figure 16).

Now the transition target is \textit{epsilon}, and the corresponding computation component is \textit{usercode} (the two symbols linked by the arrow \textit{b}). Again, the \textit{epsilon} means there is no transition target in this level, and we proceed to the next higher level. Then the system state is \textit{thread}, and the transition target sequence is (\textit{syscall scheduler}), step (c). As the current transition target \textit{syscall} is not \textit{epsilon}, the system state transitions to \textit{syscall} by moving \textit{syscall} from the transition target sequence to the system state. The new system state is \textit{thread.syscall}, and the new transition target sequence is (\textit{scheduler}). Following the same expansion procedure explained above, \textit{thread.syscall} expands to \textit{thread.syscall.kernelcode.instreexec}, finishing the transition from line 1 to line 2 in Figure 14.

\textbf{Transition target in a higher level.} In the example above, the transition target is a computation component in the same level. However, in Figure 13(d) the transition target for the \textit{kernelcode} is \textit{scheduler}, an FSM state in a higher FSM level (in the system FSM in Figure 13 (a)). We deal with the problem using the following approach: during the expansion, a special FSM state, \textit{higherlevel_scheduler}, is written in the transition target column in Figure 16 rather than the \textit{scheduler}. Then during state transition, we directly proceed to the level of the system FSM where \textit{scheduler} is found in the transition target sequence to match the \textit{higherlevel_scheduler}.

\textbf{4.6. Implementation}

A model checker is used to reason about the system behavior and explore all possible execution flows, including those that lead to system hangs. Section 4.5.5 introduced two types of rules to
govern the system state transitions: (i) expansion rules and (ii) transition rules. Since processing the rules involves frequent string operations, the Maude [19] model checker is selected to implement our formal model. Other model checker tools such as NuSMV [32] and PVS [33] excel at describing Boolean variables and operations.

Maude is a high-performance reflective language and system that supports both equation and rewriting-rule specification and programming for a wide range of applications. Equations are deterministic and cannot accommodate ambiguity or non-deterministic behavior, while rewriting rules can describe non-deterministic behavior.

Maude performs rewriting using equations much faster than is possible using rewriting rules. Therefore, our formal model is implemented using equations to achieve a fast model checking, and only expansions of nondeterministic behavior (corresponding to Figure 13 (d) and (i)) are implemented as rewriting rules.

**Model state.** The core in implementing our formal model is proper encoding of the modeled system state, which consists of following entries:

\[
<\text{SysState}; \text{TransTargets}; \text{CounterState}; \text{AttrValues}; \text{SavedAttrs}>
\]

*SysState* is a list of strings representing the current system state, and *TransTargets* is a list of strings indicating the sequence of the transition targets. During the expansion illustrated in Figure 16, *thread, thread usercode, thread usercode instruexec* are instances of *SysState*, and *scheduler, syscall scheduler, epsilon syscall scheduler, and epsilon epsilon syscall scheduler* are instances of *TransTargets*. *SysState* and *TransTargets* are manipulated according to the specifications of expansion rules and transition rules.
CounterState is the current state of the hardware counter (reset, timerinterval, timeslice, or hangdetected). AttrValues is the set of values of processor attributes, e.g., cpumode=Kernel means the system is currently in Kernel mode. SavedAttrs saves the old values of processor attributes. Upon an interrupt, the cpumode attribute is set to Kernel, and the original value of cpumode is saved for restore of the cpumode value after the interrupt is serviced.

Rule specifications. Maude rules are specified to reflect the change of the model state during system state expansions and transitions. The rules, except those for nondeterministic events, are implemented in an equation function deriveflow() which transforms an input model state into an output model state. Each step during system state expansions and transitions is specified as recursive invocation of deriveflow() with a specific input state for deterministic events, interleaved with execution of rewriting rules for non-deterministic events.

For example, in Figure 16 the expansion of system to the BCU granularity consists of four steps. Steps 1, 2, and 3 are implemented as invocations of the deriveflow() with corresponding input states. Step 4 is specified as a rewriting rule because the arrival of an interrupt is a non-deterministic event. The system state transitions, in the same example (steps (a), (b), (c) in Figure 16), are also specified as invocations of the deriveflow(). These rules are given below to illustrate specifications expressed in the Maude syntax (the change in the model state is highlighted):

(1) Expansion rules:

Step 1: \[ eq \ deriveflow((<system; ; CounterState; AttrValues; SavedAttrs>)) = deriveflow((<thread; scheduler; CounterState; AttrValues; SavedAttrs>)) \].
Step 2: \( eq \ deriveflow((<SysState \ thread; \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>)) = deriveflow((<SysState \ thread \ usercode; \ syscall \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>)) . \)

Step 3: \( eq \ deriveflow((<SysState \ usercode; \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>)) = (<SysState \ usercode \ instreexec; \ epsilon \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>) . \)

Step 4: \( rl \ [\ instreexeclipsepsilon]: \ <SysState \ instreexec; \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs> \Rightarrow deriveflow((<SysState \ instreexec \ epsilon; \ epsilon \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>)) . \)

(2) Transition rules:

Steps a & b: \( eq \ deriveflow((<SysState \ acomponent \ epsilon; \ epsilon \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>)) = deriveflow((<SysState \ epsilon; \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>)) . \)

Step c: \( eq \ deriveflow((<SysState \ epsilon; \ syscall \ TransTargets; \ CounterState; \ AttrValues; \ SavedAttrs>)) = deriveflow((<SysState \ syscall; \ usercode \ TransTargets; \ CounterState; \ setattr(cpumode, \ Kernel, \ AttrValues); \ SavedAttrs>)) . \)

Each rule transforms the input model state (the left-hand side of the rule specification) into the output model state (the right-hand side of the rule specification) by matching the input model state and performing appropriate replacement. For example, the \texttt{SysState thread} in the rule for the step 2 (see the specifications above) matches any system state representation ending with the \texttt{thread} component (the \texttt{SysState} in the input model state is a variable representing any list of component
names). Consequently in our example, *usercode* is appended to these system states, and *syscall* is added to *TransTargets*.

Any system state representation ending with *epsilon* means the BCU granularity is reached during expansion and system state transition should occur. The two transition rules in our example demonstrate this (i.e., *epsilon* at the end of *SysState*). In step (c), the system enters the kernel mode by invoking a system call. This is represented by using *setattr()*, which sets the processor attribute *cpumode* to *Kernel*. After step (c) the expansion resumes from the *syscall* component.

In addition to the rules for expansion and transition, there are a number of rules (not discussed here due to space limitation) specified for triggering timer interrupt, transitioning *CounterState*, and manipulating processor attributes (e.g., *setattr()*). *SysState* and *CounterState* may change simultaneously in some of these rules.

**Model Checking.** System hang in our model can be represented as violation of a system liveness property. Specifically, the property that a system executes without a hang is defined as “user code always and eventually executes.” Typically the system liveness property is best modeled using the Linear Temporal Logic (LTL) [34]. However, the LTL module supported by Maude model checker identifies only a single counterexample (i.e., a single case of a system hang that escapes detection).

In order to uncover all system hang scenarios that escape detection (and to overcome the drawback of the tool), we exploit explicit-state model checking supported by the *search* command in the Maude model checker. The *search* command explores the state space of the model in the breath-first way and checks if an invariant predicate is violated. Two Boolean variables, *hangisdetected* and *systemishung*, are defined, and a predicate, [(not systemishung) or
is composed such that violation of the predicate identifies a system hang which escaped detection. In our model, \textit{hangisdetected} is set to true if the hardware counter reaches the \textit{hangdetected} state (see Figure 12(c)). \textit{systemishung} becomes true if the user code has not executed for a sufficiently long time – approximated by the \textit{bound period} discussed below.

Bounded model checking is used to limit the search state space. In our approach we bound the time interval, \textit{bound period}, for the model to terminate rather than the depth in the breath-first search. Since it takes more than one time slice for SHD to detect a system hang, we determine the bound period as multiple of the time slice (e.g. 2 time slices). The bound period begins when the hardware counter is reset, which corresponds to a transition to the \textit{reset} state in the hardware counter FSM (see Figure 12(c)).

Another optimization made to minimize the state space explosion puts a limit on the number of interrupts during a bound period as system behavior repeats itself when processing a large number of interrupts. Moreover, when enumerating all possible execution flows using the explicit-state model checking, we consider only acyclic flows since they correspond to unique behavior.

\textbf{4.7. Experimental Results}

The model checking is conducted on a machine with a dual-core CPU (2.2 GHz each) and 2 GB of memory. The experiment results are reported as measurements in two categories in Table 10: detection capability and performance of model checking. Note that only acyclic flows within the bound period are counted in the analysis. The key observations from the experiments are summarized below:
Table 10: Experiment Results

<table>
<thead>
<tr>
<th>Bound Period</th>
<th>Detection Capability</th>
<th>Performance of the Model Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># flows</td>
<td># system hangs</td>
</tr>
<tr>
<td>2 time slices</td>
<td>11388</td>
<td>18632</td>
</tr>
<tr>
<td>3 time slices</td>
<td>10303</td>
<td>1363435</td>
</tr>
</tbody>
</table>

- The percentage of the detected hangs is 84.8% (15804/18632) for the 2-time-slice bound period, and is 99.5% (1356963/1363435) for the 3-time-slice bound period, as reported in Table 10. The percentage is not detection coverage in the usual meaning because the value does not reflect the frequency of runtime occurrences of different hang scenarios. This figure gives the percentile of all unique hang scenarios detected by the hang detector. This is an important result, which cannot be obtained by employing the traditional random fault injection approach. According to Table 10, the number of execution flows, as interleaved execution, scales exponentially with the bound period. The number of flows that lead to system hangs also scales exponentially. But the number of flows leading to system hangs not detected by the SHD does not demonstrate this exponential scalability. This is because this type of system hang scenario is a small set with specific nature (discussed below in Section 4.7.1).

- Most execution scenarios are in normal behavior, and only a small percentage of the execution scenarios lead to system hangs. With the 2-time-slice bound period, around 1.64% (18,632 out of 1,138,893) of execution scenarios lead to system hangs; while with the 3-time-slice bound period, around 1.32% (1,363,435 out of 103,031,856) of execution scenarios lead to system hangs, slightly smaller than the percentage for the 2-time-slice bound period.
• The model checking demonstrates very good performance: it takes 69,147 rewrites (171,125 rewrites), including equations and rewriting rules, and 122 ms (345 ms) to model check our system model when the bound period is 2 time slices (3 time slices). Note that during the model checking the entire state space of our system model (within the bound period) is traversed and the longest flow within the 3-time-slice bound period has only 72 steps (a step is execution of a rewriting rule). The small number of the steps in the longest flow, as well as the short amount of time for model checking, demonstrates the effectiveness of applying formal model checking to study behavior of complicated real-world systems regarding specific system properties.

**Example Hang Scenarios That Escape Detection**

In this section, we discuss execution scenarios that lead to system hangs and escaped detection.

**Scenario 1.** In this scenario, a user thread, executing on the processor, uses up the allocated time slice when an interrupt arrives. The handler invoked to serve the interrupt first disables any incoming interrupt and services the current one. After the interrupt service routine terminates, it enables interrupts and tries to return the control to the user mode. On the return from an interrupt, Linux checks whether a context switch should be scheduled, and in our example, the context switch is needed (the user thread used up the time slice). The SHD module intercepts the context switch and resets the hardware counter. However, before the context switch completes, another interrupt arrives and the same scenario repeats. Since SHD is always able to reset the hardware counter, it never detects the hangs.

**Scenario 2.** In this scenario, the user thread invokes a system call, and the OS enters the kernel mode to service the user request. On return from the system call service, the OS finds that the user thread must wait for a certain condition to be met before it can continue the execution. Although
the user thread does not use up the time slice, it voluntarily yields the processor, and hence, the OS schedules a context switch. The system hang detection intercepts the context switch, resets the hardware counter, and then is trapped for some unknown reason. The next time the timer interrupt arrives, the system handles the interrupt and tries to schedule the context switch (as the condition for context switch still holds). This invokes the system hang detection, which resets the hardware counter and then hangs again.

Most of the hangs that escape detection are of this nature. Our formal model enables uncovering these not-quite-intuitive scenarios and provides a feedback to the system/application developers on how to enhance detection capabilities. Note that such scenarios are rather unlikely in real systems; e.g., an interrupt coinciding with a context switch is a low-probability event.

4.8. Conclusions

This chapter proposes an approach to formally verify the detection capability of a system hang detector. An abstract formal model of a typical Linux system is created to thoroughly exercise all execution scenarios that may lead to hangs. Model checking is applied to reason about the system behavior and to uncover the hang scenarios that escape detection. The results indicate that the proposed framework allows identification of corner cases of hang scenarios that escape detection and provides valuable insight to the developers for enhancing the detection mechanisms. Although a single-core standalone system is studied in this chapter, the proposed approach is applicable to study detection capabilities of system hang detectors in more complicated systems, e.g. multi-core, multi-processor, or virtualized systems. (The number of states may still be within capability of the state-of-the-art model checking, as indicated by the very good performance of the model checking reported in Table 10). In the future, we will investigate the enhancement of the
BCU-based approach to model and verify other reliability techniques, such as application-transparent checkpointing [15] and middleware-level error detectors and mitigation techniques (e.g., [35]).
CHAPTER 5

CHECKPOINTING VMS AGAINST TRANSIENT ERRORS

5.1. Introduction

Virtual machines (VMs), also called guest systems, are frequently deployed to host a variety of IT services, such as web services, virtual desktops, and databases. To ensure continuous service availability, these systems must be capable of tolerating runtime errors. Checkpoint and rollback techniques can be applied to enhance VM availability.

Virtual machine monitors (VMMs), such as VMware and Xen, provide mechanisms (a) to save a VM state (by stopping the VM and dumping the execution state into persistent storage) and (b) to migrate the VM to a remote node (e.g., [3]). Most existing VM checkpoint techniques [36], [37], [38] exploit these two mechanisms. For example, CEVM [36] and VNsnap [37] first use live migration to create a replica of the protected VM in memory and then dump the replica to disk offline.

This chapter presents the description, implementation, and experimental assessment of VM-\(\mu\)Checkpoint, a VM checkpointing framework to protect both the guest OS and applications against runtime errors. Advantages of using VM-\(\mu\)Checkpoint include.

(i) \textit{Small overhead} compared with the VM replica-based failover approach. This is achieved by using in-memory checkpointing and in-place recovery of VMs, i.e., recovery of a failed VM in its current context. No such checkpoint work has been done in the context of virtual environments.
(ii) *Alleviation of checkpoint corruption due to error-detection latency* by taking advantage of knowledge of error detection latency. Using knowledge of fault/error latency for explicitly handling checkpoint corruption is new, and with it we can finally address this important problem;

(iii) *High checkpointing frequency*—tens of checkpoints per second—which reduces the size of each increment when taking a checkpoint.

(iv) *Rapid recovery*—within one second—compared to the stop-and-dump approach provided by VMMs.

As a result, checkpointing during the normal system operation and recovery in response to a guest VM or application failure are completely transparent to the client (i.e., the client does not see a service interrupt).

Traditional checkpointing techniques save checkpoints on disk to tolerate permanent failures. Several VM checkpointing techniques, including Remus [38], save checkpoints in the memory of another node. Here, we propose saving checkpoints in the memory of the same node.

VM-μCheckpoint is designed as a complementary approach to disk-based VM checkpointing rather than its replacement. By providing a rapid recovery, VM-μCheckpoint significantly reduces the failure rate of VMs due to transient errors.18 We created analytical and probabilistic models (presented in Chapter 7) to assess the availability improvement when using VM-μCheckpoint.

The major contributions of this work are:

---

18 At the same time, the checkpoint kept by VM-μCheckpoint can be dumped to disk at a sufficiently infrequent rate to minimize overhead. This means that in the event of a node fails, the VM and the jobs in the node can be restarted from the last valid disk checkpoint.
• Design, implementation, and integration of VM-\(\mu\)Checkpoint in Xen VMM. The VM-\(\mu\)Checkpoint implementation does not introduce any changes to the guest VMs or applications. Copy-on-Write (CoW), dirty-page prediction, and pre-saving algorithms are designed and implemented to achieve high performance. The key innovations in the proposed algorithms are (i) the use of dirty-page prediction and pre-saving, which are not supported by the default Xen’s CoW mechanism, and (ii) a mechanism to overcome the inefficiency of Xen’s CoW in supporting high-frequency periodic checkpointing.

• Use of knowledge of the error detection latency to derive checkpoint intervals that minimize the possibility of checkpoint corruption. Our model-based analysis (in Chapter 7) shows that the availability of guest VMs and applications is improved from 99% to 99.98%, assuming a highly reliable hypervisor (MTTF of 625 days in our study).

• An experimental assessment of VM-\(\mu\)Checkpoint using:
  (i) \textit{SPEC benchmark programs}. The evaluation shows that VM-\(\mu\)Checkpoint incurs an average of 6.3% overhead for SPEC benchmark programs with 50 ms checkpoint intervals.\(^{19}\) This choice represents a design tradeoff between keeping checkpoint size small and minimizing chances of checkpoint corruption due to latent errors.
  (ii) \textit{Apache server, an example network application}. The results show 17.5% throughput reduction when taking a checkpoint every 50 ms. This overhead is significantly lower than the existing VM checkpointing techniques, e.g., Remus [38].

\(^{19}\) The recent work of fault-injection into the Linux kernel [39] shows that about 95% of crashes occur within 100 million CPU cycles (or within 50 ms on a 2 GHz processor) after an error occurrence. We select a checkpoint interval of 50 ms in experiments to cover the latency for 95% of errors.
5.2. VM-µCheckpoint Design

Figure 17 illustrates how VM-µCheckpoint is deployed to protect virtual machines (also called guest systems) running on top of a hypervisor. The protected VM in the figure is the guest system to be checkpointed. Another guest system on the same physical machine is selected as the checkpointing VM where VM-µCheckpoint is installed. (The checkpointing VM can be a guest system dedicated to the checkpointing service, not necessarily a privileged guest system, e.g., Dom0 in Xen.) The hypervisor and the kernel of the checkpointing VM are instrumented to support checkpointing and recovery.

![Deployment of VM-µCheckpoint](image)

**Figure 17: Deployment of VM-µCheckpoint**

The starting point of the proposed approach is the observation that short-latency errors are dominant. This is demonstrated by several previous fault injection experiments, including recent work on error injection into the Linux kernel [39] that shows about 95% of crashes occur within 100 million CPU cycles (or within 50 ms on a 2 GHz processor) after an error occurrence. Fault injections into processor micro-architecture [40] also show small error latencies. State-of-the-art error detection techniques (e.g., [41], [42], [43]) also help to limit the error latency to low values.
At the same time, it has been shown in many studies that a vast majority of failures are transient (up to 95%). Our experiments on IBM Power series systems also demonstrate that, in VM environments, errors impacting the hypervisor rarely affect more than a single guest VM.\(^{20}\)

**Latency-driven selection of the checkpoint interval.** We define a parameter $T_B$ to be a user-specified bound on error latency. By setting the checkpoint interval greater than an acceptable latency bound (e.g., 95\(^{th}\) percentile), we effectively bound the probability of a latent/undetected error affecting the checkpoint to be small (in the best case, <5\%). In addition, by always holding two checkpoints in sequence and, on detecting an error, reverting to the earlier checkpoint (in time), we further reduce the probability of checkpoint corruption (shown is Section 7.1 to be less <3\%). This is primarily due to two factors: (i) the earlier checkpoint is taken at a time at least $T_B$ in the past, and (ii) since by choice (i.e., per the latency distribution) the chance of an error occurring in an interval $T_B$ is greater than 95%, we have a high confidence that the checkpoint taken is error free. Hence, VM-$\mu$Checkpoint assumes the earlier checkpoint is correct, or committed. When an error is detected or causes a failure, the system is rolled back to a committed checkpoint.

A user-level process in the checkpointing VM, referred to as the checkpoint agent in Figure 17, takes a checkpoint of the protected VM periodically, at intervals of $T_{ck}$ (where $T_{ck} > T_B$), and stores the checkpoint in the checkpointing VM. Since at each checkpoint the Copy-on-Write (CoW) mechanism is invoked to identify and store the needed state information, the checkpoint agent stores only a small fraction of the protected VM state rather than the entire system image. This approach allows the checkpoint agent to store checkpoints of multiple guest systems on the

\(^{20}\) My colleague, Weining Gu, conducted this error injection campaign. The result of the campaign is not published yet.
same physical machine and in a small amount of memory.

5.2.1. Checkpointing Algorithms

At the beginning of a checkpoint interval, all memory pages in the protected VM are set as read-only. From that point on, any write to a read-only page triggers a page fault, the original data of the page is copied into the checkpoint kept in the checkpoint agent memory, and the stored memory page is set as writable. The checkpoint therefore consists of original data of only those pages updated within the interval. As mentioned above, the two most recent checkpoints are kept all the time. When an error in the protected VM is detected or causes a failure, (i) the last checkpoint is copied back into the current state of the protected VM, and then (ii) the earlier of the two kept checkpoints is copied into the system. This method restores the system to the state in which the earlier checkpoint (a committed checkpoint, which is unlikely to have been corrupted) was taken. A detailed explanation of this method follows.

Figure 18 illustrates the timelines of this checkpointing/recovery scheme. Two complete checkpoint intervals, \([t_0, t_1]\) and \([t_1, t_2]\), are shown in Figure 18. The horizontal axis at the top of the figure represents error-free execution of the protected VM, while the horizontal axis at the bottom represents execution when an error occurs at \(t_f.s\). The error is detected (or the application/system fails) at \(t_f.d\) (the error latency \(T_f = t_f.d - t_f.s \leq T_h\)). At \(t_f.d\) the two most recent checkpoints are those taken at \(t_0\) and \(t_1\). We first restore the data preserved during the time interval \([t_1, t_f.d]\) into the protected VM, then we restore the data preserved during \([t_0, t_1]\), to roll back the system to the state at time \(t_0\).
In the algorithm described above, called Copy-on-Write Basic (CoW-B), setting all memory pages as read-only at the beginning of a checkpoint interval potentially results in a large number of page faults and a significant performance overhead. An optimized version of the basic algorithm called Copy-on-Write Pre-saving (CoW-P) is designed to reduce the resulted page faults; checkpoint-caused page faults are reduced by 75% when the checkpoint interval is 50 ms in our experiments, see Section 5.4.3).

**The CoW-B algorithm.** This algorithm is depicted as the timeline (a) in Figure 18. Here are the notations used in our discussion:

- $t_i$: Beginning time of the $i^{th}$ checkpoint interval
- $S_i$: State of the protected VM at time $t_i$
- $DP_i$: Data of the memory pages preserved by VM-µCheckpoint’s mechanism during $[t_i, t_{i+1}]$
- $S_t$: State of the protected VM at any time $t$ ($t \in [t_i, t_{i+1}]$)
- $DP_i(t)$: Data of the memory pages preserved by VM-µCheckpoint’s mechanism during $[t_i, t]$ for any time $t$ ($t \in [t_i, t_{i+1}]$)

The following operation reflects inherent relationship between $S_i$, $S_t$, and $DP_i(t)$:

$$S_t = restore(S_i, DP_i(t)), \quad (1)$$
where \( restore(S_i, DP_i(t)) \) denotes an operation of copying the data preserved in \( DP_i(t) \) into their corresponding memory pages in \( S_i \) to restore the system to state \( S_i \).

In the example error scenario shown in Figure 18, \( t_1 \leq t_{f,.d} \leq t_2 \), \( T_f \leq T_\beta \leq T_\delta \), and \( S_0 \) is the last committed checkpoint. Applying the operation (1) twice, we can derive the expression that depicts restoration of \( S_0 \):

\[
S_0 = restore(S_1, DP_0(t_1)) \\
= restore(restore(S_f, DP_1(t_{f,.d})), DP_0),
\]

where \( S_i = S_1 \), \( DP_0(t_1) = DP_0 \), and \( S_f \) denotes the system state at \( t_{f,.d} \). At the restoration time \( t_{f,.d} \), \( S_f \), \( DP_1(t_{f,.d}) \), and \( DP_0 \) are all available, and we can restore the memory state of the protected VM into \( S_0 \). After restoration, neither \( DP_1(t_{f,.d}) \) nor \( DP_0 \) is valid any more, as the system is now in state \( S_0 \), so they are discarded after the restoration.

**The CoW-P algorithm.** This algorithm reduces the number of page faults by predicting the pages to be updated in the upcoming checkpoint interval and pre-saving the predicted pages in the checkpoint when this interval begins. These pre-saved pages are marked as writable and do not raise page faults.

The typical checkpoint intervals selected in our approach range from tens of milliseconds to several seconds. Due to the space and time locality of memory accesses, pages that were updated recently tend to be updated again in the near future. Therefore, the pages dirtied in the previous checkpoint interval are used to predict the pages to be updated in the upcoming interval.

Specifically, a page table supported by current-generation processors maintains an entry for each
memory page. The page entry has two control bits—the write permission bit and the dirty bit—which are leveraged for our prediction. (We manipulate the shadow copy of this page table maintained by the VMM, rather than the page table in the guest operating system. In this way, the guest system’s use of its page table is not interfered with.) The write permission bit controls whether the page is writable, and the dirty bit shows whether the page has been updated since the dirty bit was last cleared. At the beginning of a checkpoint interval, both of the bits for non-dirty pages are cleared (i.e., set as read-only and not dirty). While the pages dirtied in the previous checkpoint interval are saved in checkpoint, their write permission bits are set to allow writes to them, and their dirty bits are cleared to enable tracking of whether they will be updated during the upcoming interval. If a page dirtied in the previous interval is not updated during the upcoming interval, then next time (i.e., after this upcoming interval) this page is not pre-saved and is set as read-only. Figure 18 (b) shows the timeline of the CoW-P.

More formally, let \( H_i \) denote data of the memory pages updated in the checkpoint interval \([t_{i-1}, t_i)\) (\( H_0 \) is obtained by profiling system execution before \( t_0 \)), \( DP_i' \) denote data of the pages preserved by CoW-P during \([t_i, t_{i+1})\), and \( DP_i'(t) \) be data of the pages preserved by CoW-P during \([t, t)\) for any \( t \in [t_i, t_{i+1}] \). Then we use \( H_i \) as prediction of \( DP_i \). Using the \( restore() \) operation defined in (1), we have:

\[
S_i = restore(S_i, DP_i'(t) \cup H_i).
\]  

Due to the inaccuracy of dirty page prediction, \( H_i \) includes data of pages that are not updated in \([t_i, t]\). Similar to the discussion on CoW-B, the expression that represents restoration of \( S_0 \) is derived as:
\[
S_0 = \text{restore}(S_t, DP_0'(t_1) \cup H_0)
= \text{restore}(\text{restore}(S_f, DP_1'(t_{f-d}) \cup H_1), DP_0' \cup H_0),
\]

where \( S_t = S_1 \), \( DP_0'(t_1) = DP_0' \).

5.2.2. Disk-Based Checkpointing

In order to recover a VM against a permanent failure or a hypervisor failure, VM checkpoints can be saved in disks. Figure 19 illustrates the extension of VM-\(\mu\)Checkpoint to support disk-based VM checkpoint.

\[
\text{collect}(\text{collect}(DP_0, DP_1), \ldots, DP_k)
\]

Specifically, in addition to the in-memory checkpointing described in Section 5.2.1, the checkpoint agent scans the protected VM and saves every memory page (i.e. the SCANDATA in Figure 19). Suppose the scan starts in \([t_0, t_1]\) and finishes in \([t_k, t_{k+1}]\). We define an operation \(\text{collect}(DP_0, DP_1)\) merges the data in \(DP_0\) and \(DP_1\): if a memory page is saved in both \(DP_0\) and \(DP_1\), only the data of the page in \(DP_0\) is preserved after the merge. Then,

\[
DPS = \text{collect}(\ldots\text{collect}(\text{collect}(DP_0, DP_1), DP_2), \ldots, DP_k)
\]

\[
S_0 = \text{collect}(DPS, \text{SCANDATA}),
\]

where \(DPS\) is the \(t_0\)-state of the memory pages which are modified (updated/created/deleted)
during \([t_0, t_{k+1}]\). So the checkpoint agent keeps \(DP_0\), \(DP_1\), \(\ldots\), \(DP_k\) as well as \(SCANDATA\) in order to support disk-based checkpoint. After \(t_{k+1}\) the checkpoint agent writes \(collect(DPS, SCANDATA)\) to disk, the VM checkpoint at \(t_0\).

5.2.3. Discussion

Checkpoint data. The state of a VM includes the virtual processor state, the entire kernel address space, and the address spaces of all the processes in the VM, including all memory pages belonging to these address spaces. From the hypervisor point of view, all these memory pages are managed as pseudo-physical pages. The hypervisor does not need to differentiate whether a page belongs to a user process or to the kernel.

The checkpoint data in VM-\(\mu\)Checkpoint consists of the original state of virtual CPUs and the original state of dirty pages (pseudo-physical pages). The hypervisor provides functionality to capture the state of virtual CPUs. For example, in the Xen VMM, when the hypervisor suspends a VM, the state of the virtual CPUs is frozen and can be saved in a hypervisor data structure called \(vcpu\_guest\_context\). This data structure is listed below (for Xen 3.3.1 on top of 32bit Intel x86 processors). From the data structure, we can see that all the user registers, control registers, debug registers, FPU registers, etc., are saved in the checkpoint.

```c
struct vcpu_guest_context {
    /* FPU registers come first so they can be aligned for FXSAVE/FXRSTOR. */
    struct { char x[512]; } fpu_ctxt;      /* User-level FPU registers */
    unsigned long flags;                   /* VGCF flags */
    struct cpu_user_regs user_regs;        /* User-level CPU registers */
    struct trap_info trap_ctxt[256];       /* Virtual IDT */
    unsigned long ldt_base, ldt_ents;      /* LDT (linear address, # ents) */
    unsigned long gdt_frames[16], gdt_ents; /* GDT (machine frames, # ents) */
    unsigned long kernel_ss, kernel_sp;    /* Virtual TSS (only SS1/SP1) */
    /* NB. User pagetable on x86/64 is placed in ctrlreg[1]. */
    unsigned long ctrlreg[8];              /* CR0-CR7 (control registers) */
    unsigned long debugreg[8];             /* DB0-DB7 (debug registers) */
}
```
Because we leverage the shadow-paging feature of the Xen (all of the guest system’s pseudo-physical pages are managed by the shadow page table; more details are given in Section 5.3.1), we are able to control whether a page is read-only and to trace whether a page is dirty. When there is a write to a read-only page, a page fault is triggered and reported to the hypervisor, and we can save the pre-write state of the page in the checkpoint.

There are exceptional cases when a memory page may be updated with the shadow paging bypassed. For example, if a privileged instruction in the guest system tries to write to a page, the instruction is trapped into the hypervisor due to the privilege violation. Then the hypervisor emulates the instruction by mapping and writing to the target page. To deal with this issue, we may modify the `emulate_privileged_op()` function in Xen such that the pre-update state of the modified pages is correctly checkpointed.

Another example is the memory shared between different VMs on top of the same physical machine. For example, Xen provides the grant table mechanism, which allows memory pages of a VM to be shared with other VMs. This can be used for transferring I/O data between domain 0 and another VM. In this case, if the memory page is provided by domain 0, then the page is not managed by the shadow page table in the protected VM.

We found such a scenario. The network driver of a VM consists of two parts, the front part in the VM and the back part in the domain 0. The two parts share a data buffer. When traffic data is...
received from outside, the back end fills the buffer with data (and marks the corresponding buffer slot as occupied) while the front end reads data from it (and marks the buffer slot as empty). Let us say the front end is to read slot 8 of the buffer when a failure occurs. The VM recovers to its checkpoint when the front end is to read a previous buffer slot, say slot 3. As the buffer is owned by domain 0, the buffer is not checkpointed or rolled back upon the failure. So the front end, after recovery, finds the slot is empty and reports a kernel exception. To handle this problem, we should save the shared pages and the relevant data structure state in the checkpoint. When the VM gets recovered upon a failure, the checkpointed shared pages are restored in the domain 0. The back end of the network driver in domain 0 should be modified appropriately to know about this recovery and be able to work correctly with the recovered state.

Selecting $T_B$ and $T_{ck}$. It should be clear from the above discussion that both error latency and checkpoint overhead are considered when selecting a checkpoint interval $T_{ck}$. In our scheme, checkpointing with larger interval incurs smaller overhead while causing a longer output delay and a larger checkpoint size (when output commit is employed). Hence, there is a trade-off in $T_{ck}$ selection. For example, if a small output delay is desired, a small $T_{ck}$ is preferred as long as $T_{ck}$ is no larger than the selected $T_B$ and checkpoint overhead is acceptable.

Error detection latency depends on error detection techniques (e.g., [41], [42], [43]). As error detection is not in the scope of this work, we consider system failure as a kind of error detection. To obtain the distribution of error detection latency, we inject errors into a target system and measure the latency from error activation to failure occurrence. We conducted an analytical model to study the impacts of $T_{ck}$ on checkpoint corruption and system availability in Chapter 7. Based on the analytical model and the obtained error latency distribution, we can select the proper $T_{ck}$. 
**Error model.** VM-µCheckpoint recovers a guest system and applications in the system from any transient hardware error or transient software error (including both application and system errors). Transient hardware errors include those occurring in the processor (functional units, registers, caches, buses, and control logics) and memory due to events such as radiation or current disturbance. Transient software errors, or Heisenbugs [44], include exceptional conditions (e.g., counter overflow and interrupt arrival with bad timing), occasional device driver faults, race conditions, and corrupted parameters or data due to bad transmission. Note that transient failures of the checkpointing VM are handled by an immediate restart of the failed checkpointing VM.

VM-µCheckpoint cannot guarantee recovery if either of the following holds: (i) *Checkpoint corruption.* There is a small but finite probability of checkpoint corruption. In this case, VM-µCheckpoint aborts recovery and restarts the VM and the interrupted jobs. (ii) *Failure of the hypervisor due to a transient fault.* In this case, we first restart the hypervisor and restart all jobs executing prior to the failure. If this is unsuccessful, the system rolls over to an adjacent physical node and restarts.

**I/O handling.** This work focuses on the analysis, design, and implementation of memory-state checkpointing in VM-µCheckpoint. For I/O handling, we adapted the output-commit mechanism applied in [38], [45] to fit into VM-µCheckpoint. In this mechanism, output of a system is held (i.e., not delivered to hardware devices) until a checkpoint is taken. This mechanism masks recovered errors of the system, i.e., these errors are not viewed by other components (disks, network cards, nodes, etc.).

Basically, the checkpoint agent in VM-µCheckpoint is designed to hold and release output of the protected guest system; if preferred, a copy of input to the protected VM is saved in the
checkpoint agent for replay. The hypervisor and the checkpointing VMs are instrumented to provide support to the checkpoint agent for this purpose. Note that the checkpoint agent maintains two pools of held outputs and saved inputs corresponding to the two checkpoints.

5.3. Implementation

Our fully working prototype is implemented in Xen. The source codes of the Xen hypervisor and the checkpointing VM are instrumented, while there is no change to the protected VM.\textsuperscript{21}

Xen supports multiple types of virtualized guest systems, including ParaVirtualized systems (PV) and Hardware-aided Fully-Virtualized systems (HVM). We select a Xen PV guest system as the protected system in our implementation because we have more experience with the Xen PV system. The proposed VM-\textmu-Checkpoint mechanism can also be implemented to support Xen HVM systems.

5.3.1. Overview of Shadow Paging

The VM-\textmu-Checkpoint implementation leverages a feature of virtualization technology, shadow paging, for identifying all memory pages belonging to a specific virtual machine, setting page access privileges, and intercepting page faults for preserving memory pages.

Two types of addresses are recognized in an operating system: \textit{linear address} and \textit{physical address}. A linear address is the reference of a memory object in a process address space; a physical address provides the information of where the memory object is really located in the physical memory. Note that a processor is always within a process context, no matter when the

\textsuperscript{21} The I/O recovery mechanism is not implemented yet in the current prototype.
processor is executing a user-level instruction, servicing an interrupt/system call, or doing any other task. As a result, a linear address is always used to access a memory object.

In a system without virtualization, the processor translates a linear address into the corresponding physical address and accesses the correct object in physical memory via a memory management unit (MMU) in the processor. The MMU consults a page table, maintained by the operating system, for address translation.

The shadow-paging feature of the Xen allows a separate page table (shadow page table, or SPT) to be created for a virtual machine, which also maintains its own page table (guest page table, or GPT). The SPT is different from the GPT in that the SPT maps a linear address to the real memory location in the physical machine (called the *machine address*), while the GPT maps the linear address to the virtual physical address in the guest operating system’s view (called the *guest physical address*).

Two points should be noted here: (i) Xen is able to access any memory location with a given machine address by setting up a mapping from a linear address within the Xen hypervisor address range (this address range is shared among all guest system address spaces) to the given machine address, and (ii) the MMU hardware looks up in the SPT instead of the GPT while translating linear addresses.

The Xen hypervisor synchronizes the SPT with the GPT so that the two page tables are consistent at all times, i.e., any linear address of a process in the guest system has the correct virtual physical address in the GPT and the correct real machine address in the SPT with the same bookkeeping information in the two tables.
The guest system is unaware of the SPT’s existence. Xen implements the SPT in a transparent fashion for systems on processors lacking native support to shadow paging. Moreover, latest-generation processors have built-in design (e.g., Intel Extended Page Table and AMD Nested Page Table) to support shadow paging, which masks the SPT to guest operating systems.

5.3.2. Data Channel

The checkpoint agent allocates a number of memory pages for storing two checkpoints. Each of these memory pages has the following information kept in a page record: (i) a pointer to the memory page in the checkpoint agent address space, (ii) the machine address of this memory page, and (iii) the guest physical address of the saved page in the protected system (i.e., the content of this page is stored in this memory page).

The page records of these memory pages for storing checkpoint are organized in a circularly linked list (shown in Figure 20). The list is linked with both pointers in the checkpoint agent address space and machine address information. Thus, both the checkpoint agent and the Xen hypervisor are able to traverse the page list and access these memory pages.

![Figure 20: Memory Pages for Checkpoint Storage](image)

This list of memory pages for checkpoint storage is the data channel (shown in Figure 17) set up between the protected guest system and the checkpoint agent. At any time two checkpoints are
maintained by the VM-\mu{Checkpoint}, they coexist in this list of pages (for example, checkpoint 0 and 1 are in the list of pages in Figure 20). Each checkpoint occupies a range of these pages. When the latest checkpoint (checkpoint 1 in the figure) is committed, the older checkpoint (checkpoint 0) is discarded by invalidating its range of pages, and the new checkpoint (checkpoint 2) is stored following the last occupied page.

Besides hosting the memory pages for checkpointing memory data in the protected system, the checkpoint agent provides space for storing the states of all virtual CPUs in the protected system.

5.4. Experimental and Measurement Results

The testbed consists of a physical machine with an AMD Athlon 2800 (1.8 GHz) processor and 1.5 GB memory. There are only two guest systems (Linux 2.6.18) running on top of Xen 3.3.1 in the testbed. The Dom0 is selected as the checkpointing VM, and the other guest system (a DomU) is the protected VM. 512 MB and 1 GB memory are assigned to the Dom0 and the DomU, respectively. We use only two VMs in experiments in order to accurately measure performance overhead in a relatively simple deployment.

To summarize major findings in our experiments:

a) VM-\mu{Checkpoint} achieves much better performance than existing migration-based VM checkpoint. For workload of SPEC CINT 2006 benchmark and checkpoint frequency of 20 times per second ($T_{ck}=50ms$), an average of 6.3\% overhead is incurred when CoW-P is deployed. With the same checkpoint algorithm and checkpoint frequency, Apache server throughput is reduced by 17.5\%. In contrast, Remus [38], a migration-based VM replication/checkpoint technique, reports approximately 50\% overhead in their experiments for the same checkpoint frequency. If we
reduce the frequency to 5 times per second the average overhead is only 3.8% for the SPEC CINT 2006 workload.

b) VM-µCheckpoint achieves much better performance than existing migration-based VM checkpoint. For workload of SPEC CINT 2006 benchmark and checkpoint frequency of 20 times per second (T_{ck}=50 ms), an average of 6.3% overhead is incurred when CoW-P is deployed. With the same checkpoint algorithm and checkpoint frequency, Apache server throughput is reduced by 17.5%. In contrast, Remus [38], a migration-based VM replication/checkpoint technique, reports approximately 50% overhead in their experiments for the same checkpoint frequency. If we reduce the frequency to 5 times per second, the average overhead is only 3.8% for the SPEC CINT 2006 workload and around 9% for the Apache web server (using CoW-P).

c) The speedup the CoW-P algorithm gains over CoW-B is significant when checkpoint frequency is high. With CoW-P deployed with 50 ms checkpoint intervals, Apache throughput is 82.5% of the baseline performance, which is larger than 74.3% when CoW-B is deployed.

d) Checkpoint sizes are relatively small with short checkpoint intervals selected in our experiments. The results show that, with 50 ms checkpoint intervals (using CoW-P), all checkpoint sizes are less than 2000 memory pages (8 MB) with an average of 655 pages (2.6 MB) for the SPEC CINT 2006 workload (the size of the entire system state is up to 51461 memory pages, or 206 MB).

5.4.1. Program Execution Time

A set of SPEC CINT 2006 benchmark programs are executed in the protected guest system with VM-µCheckpoint deployed. A suite of experiments are conducted involving each of these
benchmark programs: (i) a baseline case (no checkpoint), (ii) CoW-B algorithm deployed with 4 different checkpoint intervals (1000 ms, 600 ms, 200 ms, and 50 ms), and (iii) CoW-P algorithm deployed with the same 4 intervals. A given program executes with the same input across all experiments.

Program execution times are measured, and normalized execution times are illustrated in Figure 21 (95% confidence intervals of execution times are computed but are not presented in this figure). Normalized execution time is computed by dividing program execution time by the execution time in the corresponding baseline case. We observe from Figure 21 that:

![Figure 21: Experiment Results in Terms of Execution Time of SPEC CINT 2006](image)

i) For all programs the impact of the checkpoint on the program execution time is no more than 11% (the normalized execution times are no more than 1.11) and the average overhead is 6.3% (the average of the normalized execution times is 1.063) when the CoW-P algorithm is deployed with 50 ms checkpoint intervals. Compared with around 50% overhead in Remus, this is great improvement. If we increase the checkpoint interval to 200 ms, the average overhead is now 3.8% (using CoW-P).
ii) The performance overhead increases as checkpoint frequency grows.

iii) Use of CoW-P gains larger speedup over CoW-B for high checkpoint frequency. This is because the pre-saving in CoW-P reduces the number of page faults when the checkpoint interval is small. For low-frequency checkpoint, the pre-saving does not provide much improvement; in certain cases it even degrades checkpoint performance. Such performance degradation can be observed in experiments for perlbench, omnetpp in Figure 21. This result is due to the fact that memory access locality plays a significant role when checkpoint intervals are short. With a checkpoint interval as large as 1 s, there are (in general) a lot of mispredictions, and the pre-saving does a lot of wasteful work preserving pages not to be updated.

5.4.2. Web Server Throughput

We conducted experiments to study how VM-μCheckpoint affects Apache web server throughput when the web server runs on the protected guest system. Web clients reside on three physical machines with each machine hosting 50 clients. These clients request the same load of web pages, one request immediately after another, from the server simultaneously via a 100 Mbps LAN. The output-commit mechanism is disabled in these experiments (as the I/O handling is not the focus of this work), and consequently, we compare our performance with Remus results when the output commit is also disabled.
Figure 22: Impacts of VM-µCheckpoint on Apache Web Server Throughput (percentage represents the ratio between the corresponding throughput and the baseline throughput, e.g., 82.5% = 229.8/278.7)

We measured throughputs of the web server in multiple experiments with VM-µCheckpoint deployed at different checkpoint intervals. As the same load of web requests are processed in these experiments, the measured throughputs can be compared for evaluating the impact of our checkpoint on throughputs.

Figure 22 shows the measured server throughput as a function of checkpoint intervals. The percentages indicated along the data points on the graph represent the ratio between the throughput measured with the checkpoint deployed and the throughput in the baseline case (when checkpoint is not deployed). The three observations we made from measurements of program execution times in Section 5.4.1 are confirmed by the throughput results.

i) The throughput is reduced by 17.5% when a checkpoint is taken 20 times per second. Remus reports approximately 50% overhead for SPECweb benchmark with the same checkpoint frequency (when output commit is enabled their overhead is around 72%). If the checkpoint interval is increased to 200 ms, the throughput is reduced by only 9%. Our overhead results are
conservative because we run a stressful load of web requests in experiments, and the typical server workload is not as intensive.

  ii) Checkpoint overhead increases with higher checkpoint frequency.

  iii) The CoW-P algorithm has performance improvements over CoW-B, especially in cases with small checkpoint intervals (the gaps between the two curves keep increasing and are much larger for intervals of 200 ms and 50 ms in Figure 22).

5.4.3. **Overhead Measurement**

The number of checkpoint-caused page faults is a direct measurement of the time overhead of our checkpointing (other page faults are not counted as checkpoint overhead). Checkpoint size, i.e., the number of memory pages kept in a checkpoint, is the space overhead measurement.

A number of experiments are conducted to study VM-µCheckpoint overhead with different checkpoint algorithms (i.e., CoW-B or CoW-P) at different checkpoint intervals. In each of these experiments, 12 programs of SPEC CINT 2006 benchmark are executed in a sequential way, and the total duration is about half an hour. We measure the numbers of checkpoint-caused page faults and the checkpoint sizes (in terms of numbers of memory pages) in every checkpoint interval (e.g., 50 ms) of this experiment duration. For example, Figure 23 shows the numbers of checkpoint-caused page faults during an experiment with CoW-B deployed at a checkpoint interval of 1000 ms (the x-axis represents the execution time of the experiment). The labels 1 to 12 indicate the periods in correspondence to the executions of each of the 12 programs, respectively.
Figure 23: Checkpoint-Caused Page Faults When CoW-B Is Deployed at the Checkpoint Interval of 1000 ms

Table 11: Average Checkpoint-Caused Page Faults in Experiments

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>T_{ck}(ms)</th>
<th>Overall</th>
<th>perl-bench</th>
<th>gcc</th>
<th>milc</th>
<th>dealII</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoW-B</td>
<td>50</td>
<td>491.1</td>
<td>342.4</td>
<td>396.0</td>
<td>1112.7</td>
<td>235.8</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1057.2</td>
<td>1076.6</td>
<td>1398.1</td>
<td>1298.0</td>
<td>996.0</td>
</tr>
<tr>
<td>CoW-P</td>
<td>50</td>
<td>124.7</td>
<td>139.7</td>
<td>172.2</td>
<td>35.7</td>
<td>163.3</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>521.5</td>
<td>182.0</td>
<td>527.9</td>
<td>319.7</td>
<td>842.8</td>
</tr>
</tbody>
</table>

Table 12: Average Checkpoint Sizes (in Number of Memory Pages) in Experiments

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>T_{ck}(ms)</th>
<th>Overall</th>
<th>perl-bench</th>
<th>gcc</th>
<th>milc</th>
<th>dealII</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoW-B</td>
<td>50</td>
<td>491.1</td>
<td>342.4</td>
<td>396.0</td>
<td>1112.7</td>
<td>235.8</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1057.2</td>
<td>1076.6</td>
<td>1398.1</td>
<td>1298.0</td>
<td>996.0</td>
</tr>
<tr>
<td>CoW-P</td>
<td>50</td>
<td>654.5</td>
<td>565.7</td>
<td>626.3</td>
<td>1154.6</td>
<td>524.2</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>2162.4</td>
<td>1260.0</td>
<td>2144.6</td>
<td>1873.9</td>
<td>3151.6</td>
</tr>
</tbody>
</table>

Results. The overall checkpointing overheads measured throughout the experiments, as well as the overheads for several individual programs, are presented in Table 11 (time overhead) and
Table 12 (space overhead). The major findings from the experimental results are summarized below.

- **Average checkpoint sizes are very small, less than 2% of the size of the entire system state when checkpoint interval is 50 ms.** Table 12 shows that with CoW-P deployed at a checkpoint interval of 50 ms, the average checkpoint size is 654.5 memory pages or 2.6 MB, while the size of the entire system state during the experiment is up to 51,461 memory pages (206 MB). The maximum checkpoint size observed is less than 8 MB (2000 pages; due to space constraints the figure showing this data is not given here), less than 4% of the entire system state size. When the checkpoint interval is increased to 1000 ms, most checkpoints are less than 10,000 pages, and the average size is 2162.4 pages (8.6 MB, or 4.2% of the entire state).

- **Dirty page prediction and pre-saving effectively reduce page faults by 75% when the checkpoint interval is 50 ms** (124.7 page faults in CoW-P, while 491.1 in CoW-B, as shown in Table 11). When the checkpoint interval is 1000 ms, CoW-P still achieves 51% reduction of page faults (1057.2 reduced to 521.5). The reduction is less for larger checkpoint intervals, as there is more memory access locality within shorter intervals.

An interesting observation is that there are a small number of single peak values in Figure 23 (up to 21,363; they are cropped to make the figure easier to read). These peak values are caused by a program’s loading of a large amount of data (mostly for read).

5.4.4. **Experiments on Virtual Machine Recovery**

Experiments were conducted to test the correctness of the proposed technique in recovering a virtual machine and to measure the recovery time for evaluation. We regard application failure as
error detection in these experiments rather than installing an error detector to do this job. For this purpose, a small program is developed that causes a segmentation failure after executing for a while. The instrumented hypervisor-level exception handler then issues an “error detected” request via a divide-by-zero exception.

The SPEC CINT 2006 benchmark programs run as the workload on the protected virtual machine in these experiments. The small program is launched to generate a failure while the workload is running. The protected virtual machine is then rolled back to the last committed checkpoint. The measured recovery time depends on the number of memory pages restored during recovery. As most of checkpoint sizes range from several hundred to several thousand memory pages (shown in Table 12), the measured recovery time ranges from 144 ms to 1017 ms with the average of 639.4 ms (the 95% confidence interval is 639.4 ms ± 193.1 ms).
In Chapters 2 and 3, we discussed the RMK architecture and a number of error detection techniques (e.g., system hang detection) on standalone systems. In Chapter 4, we discussed the technique of checkpointing VMs. In this chapter, we see how the RMK architecture is enhanced for a virtualized environment and how the enhanced RMK allows error detection and checkpoint/recovery to be integrated in virtualized environments.

6.1. RMK Deployment in Virtualized Environment

Figure 24 depicts the RMK deployment in the protected VM and the hypervisor. Similar to the RMK in a standalone system, the RMK is installed as a device driver in the VM. The kernel source code is not required, as the kernel of the VM does not need to be recompiled. Figure 24 shows the SHD module is installed in this RMK together with two RMK pins: P_PMC and P_SCHL.

The Xen hypervisor does not provide a standard mechanism to allow for dynamically loading code into the hypervisor space as the Linux kernel module does. So we instrumented the Xen hypervisor and encapsulated the hypervisor-level RMK (including the RMK core and RMK modules/pins) into a Xen hypercall. A hypercall is like a system call for a hypervisor. The instrumented Xen is recompiled to get the RMK installed in the hypervisor.
The RMK modules \textit{COWB}, \textit{COWP}, and \textit{recovery} implement the two checkpoint algorithms and the recovery algorithm in VM-\( \mu \text{Checkpoint} \). The RMK pins \textit{P\_VCPU} and \textit{P\_PTABLE} wrap the hypervisor functionalities of manipulating virtual CPUs and the shadow page table, respectively. \textit{P\_VMSIGNAL} intercepts received signals, and \textit{P\_VMSCHL} intercepts scheduling of VMs by the hypervisor.

6.2. Detecting and Recovering from VM/Application Crashes

When a guest system or an application in this guest system crashes, e.g., due to a NULL pointer or segmentation fault, an exception is raised by the hardware and reported to the hypervisor. The \textit{P\_VMSIGNAL} pin intercepts this exception, determines whether the exception causes a crash (e.g., segmentation fault) or not (e.g., page fault), and produces an EVT\_ERRORDETECTED event if the exception causes a crash. Note that here we capture crashes of both the protected VM

\textbf{Figure 24: Integrated Error Detection and Checkpoint/Recovery under RMK in a Virtualized Environment}

- Hypervisor
- P\_PMC
- P\_PMC\_HELPER
- P\_VMSCHL
- P\_VMSIGNAL
- P\_VMSIGN
- P\_VMSIGN
- P\_PTABLE
- P\_VCP
- COWP
- COWB
- recovery
- RMK core
- SystemHangDetection
- guest OS
- checkpointing VM
- protected VM
- app
- Checkpoint agent
- guest OS
- app
and applications in the VM.

The *recovery* module subscribes the event. Upon receiving the event, it suspends the failed guest system (or the guest system in which the failed application is located) and requests the checkpoint agent for recovery. The checkpoint agent then recovers the guest system state with its checkpoint via the *recovery* module (details of the checkpointing and recovery procedures are given in Chapter 5).

### 6.3. Detecting and Recovering from VM Hangs

Recall that we count executed instructions to detect system hangs in the SHD module (Chapter 3). In a system executing equal-priority tasks, the number of instructions executed between two consecutive context switches in the protected VM is bounded. The VM system in a hang state does not relinquish the virtual CPU and does not schedule any thread. When the instruction number between consecutive context switches in the VM exceeds a preset value, a VM hang is flagged.

Specifically, the P_SCHL pin intercepts the guest system scheduler by means of binary rewriting (so there is no need to recompile the guest system kernel). When a context switch occurs in the VM, a hardware counter is reset to zero by the P_PMC pin. Because the hypervisor does not allow the protected VM to modify hardware counters (privileged instructions are involved), the P_PMC_HELPER pin in the hypervisor modifies the hypervisor source code to grant such accesses to the VM.

The P_VMSCHL pin instruments the hypervisor scheduler to check whether the hardware counter value exceeds a preset threshold value. If the value exceeds the threshold, P_VMSCHL generates
an EVT_ERRORDETECTED event, which will be processed by the recovery module and the checkpoint agent to recover the VM from its checkpoint. Compared to the system hang detection on the standalone system (Chapter 3), we do not need to take advantage of NMI interrupt to flag a system hang. This is because the hypervisor allows for checking the hardware counter value directly, even when the VM has failed.

Moreover, P_VMSCHL ensures that the hardware counter counts only instructions executed by the protected VM. When the VM is switched off the processor by the hypervisor, the hardware counter is suspended; when the VM is switched onto the processor again, the hardware counter is resumed. Here we assume only one VM is protected. When multiple VMs are to be protected, we can keep in the hypervisor multiple variables that record the counts for these VMs, and the number of executed instructions counted by the hardware counter is added to one of these variables properly for the corresponding VM.

6.4. Error Injection Experiments

We conducted error-injection experiments to show the validity of the automatic approach that RMK provides for error detection and VM checkpoint/recovery. Our experiments showed that, if the error injector is a script or program within the protected VM, the VM is recovered from the checkpoint successfully by RMK, but the same error is injected again immediately. This is because our error injection script/program is deterministic even when pseudo-random numbers are used for the error injection.

Therefore, to avoid this repetition of error injection in an automatic experiment campaign, information from outside the protected VM is required to control the error injection behavior. For example, a flag variable in the protected VM can record whether an error is to be injected, and this
flag variable is set by the hypervisor outside the protected VM. Because we just want to show the validity of our approach rather than do a systematic error injection campaign, we did not implement such an automated experiment, but rather conducted these experiments manually. As a result, only tens of experiments were run. To make our life easier in these manual experiments, a simple program is applied as the workload.

Table 13 lists the results of our manual experiments. Three kinds of errors were injected, as described in the following sections.

Table 13: Results of Error Injection Experiments

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Fault/Error</th>
<th>Injected Faults/Errors</th>
<th>Activated and Detected</th>
<th>Recovered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal-triggered crashes</td>
<td>Sends SIGTERM to the application in the protected VM (fail-stop)</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>Bit flips into kernel registers</td>
<td>Suspends the protected VM, flips a bit in a register value in the VM, and resumes the VM</td>
<td>85</td>
<td>31 activated and detected</td>
<td>31</td>
</tr>
<tr>
<td>System hangs (threshold of 200 ms)</td>
<td>Loads a device driver which runs an infinite loop in the protected VM</td>
<td>30</td>
<td>30</td>
<td>24*</td>
</tr>
</tbody>
</table>

*Due to inconsistent shared state between the protected VM and Dom0 for I/O operations

6.4.1. Signal-Triggered Crash

We kill the workload application by sending a SIGTERM signal to the application process. On receiving this signal, the workload program executes a divide-by-zero instruction, which traps the processor into the hypervisor. The P_VMSIGNAL pin then generates the EVT_ERRORDETECTED event, and the recovery module restores the checkpoint to the protected VM. We conducted 35 experiments, and in all of them the protected VM is successfully recovered.
6.4.2. Bit Flips in VM Kernel Registers

We also injected bit-flip faults into kernel registers of the protected VM. The error injector is placed outside the protected VM, i.e., in the dom0, to avoid repeating the same error injection after recovery from the checkpoint.

The error injector in the dom0 first suspends the protected VM via the hypervisor. As a result, the state of the virtual CPUs in the VM is saved in a hypervisor data structure called `vcpu_guest_context`. Then we randomly select a bit in the generic register file (i.e. EAX, EBX, ESI, EDI, ESP, EIP, CS, SS, ES, etc.) in the `vcpu_guest_context` and flip it. Then we resume the protected VM, and the flipped value is written back to the corresponding register in the virtual CPUs.

We did tens of experiments and found that the activation rate is fairly small (i.e., only 31 out of 85 injected faults get activated). For the activated and manifested errors, all of them are detected by the P_VMSIGNAL pin and are successfully recovered. To better evaluate the coverage of the checkpoint/recovery, we then injected into only the EIP register. Now all of these faults are detected and recovered.

6.4.3. System Hangs

We also injected system hangs in the protected VM. Specifically, we load in the VM a device driver that runs an infinite loop. In the 30 experiments conducted, there are 6 cases in which the recovery from the checkpoint fails.

We looked into the details of the cases when recovery fails and discovered that the shared state between the protected VM and the dom0 for handling I/O operations is inconsistent with the state
of the protected VM after recovery from the checkpoint. Figure 25 illustrates the details of the inter-domain shared memory for I/O operations in Xen.

![Diagram of inter-domain shared memory for I/O operations in Xen]

**Figure 25: Inter-Domain Shared Memory for I/O Operations in Xen**

The Xen hypervisor uses a split driver model for handling I/O operations (network, disk, etc.). The blkfront is the front end of the driver in the protected VM, and the blkback is the back end in Dom0 (as shown in Figure 25). Shared memory is used to facilitate I/O data transfer. These shared states include request ring buffer, producer/consumer pointers (blkfront and blkback follow a producer-consumer model), buffers, protocol status for the split driver, event channel state, etc.

When a request arrives at either the blkfront or the blkback, a shared buffer is created in the protected VM or the dom0 to host I/O data, and this buffer is registered through the grant table mechanism in the hypervisor. After processing the request, the buffer is released through the grant table.

If the protected VM is recovered from checkpoint, the blkfront may expect a shared buffer to be present and registered in the hypervisor’s grant table. But this may not be true. The recovery then fails because a non-existent buffer is accessed. Our experiments show that this scenario happens when the error detection latency is large. That is why we only observed failure of recovery in cases when system hangs are injected (200 ms is used as the threshold value for detecting VM
To handle this problem, we should instrument the hypervisor and the blkback driver in the dom0 to handle I/O correctly. Besides holding output data until checkpoint is committed (similar to the “output-commit” mechanism used in Remus and Revive I/O), we should also save the shared memory and the grant table in the checkpoint.

This work is not done yet in the current prototype implementation because I/O handling is not the research focus of this thesis work, and significant engineering efforts are required to analyze the source code because the split-driver model is not well documented.
CHAPTER 7

MODEL-BASED ANALYSIS OF VM CHECKPOINTING

7.1. Checkpoint Corruption Model

Two factors are important in determining checkpoint corruption: error occurrence instant and error detection latency. Both are addressed in our latency-driven checkpointing provided by VM-μCheckpoint. In this section, we construct a model of checkpoint corruption scenarios to study how the latency-driven checkpointing alleviates checkpoint corruption. The following three assumptions are made in our model to simplify the analysis while still providing valuable insight into checkpoint corruption behavior:

(i) Unmasked errors\(^{22}\) are eventually detected by either application-level (e.g., embedded assertions) or system-level (e.g., application failure, exception handling, kernel panic) detection mechanisms, and only detected errors can trigger checkpoint-based recovery;

(ii) Error occurrence probability is uniformly distributed during any given period; and

(iii) Error latency is exponentially distributed.

The checkpoint corruption model is constructed for a given unmasked error occurrence. We first identify the checkpoint interval in which the error occurrence falls. As an example, Figure 26 (a) shows that an error occurs between \(chkpt0\) and \(chkpt1\). The time offset of the error occurrence

\(^{22}\) An unmasked error is a transient error that remains alive throughout the program life and is not overwritten by the program.
relative to the time of $chkpt0$ is denoted as $a \ (0 \leq a < T_{ck})$. The system continues execution after the error occurrence, and the error is detected after a latency of $l$ (also shown in Figure 26 (a)). $a$ and $l$ are two random variables. $a$ is uniformly distributed within $[0, T_{ck})$, and $l$ is exponentially distributed at a rate $\lambda$. Then, the pdf (probability distribution function) for $a$ is given by:

$$pdf\ (a) : f(x) = \frac{1}{T_{ck}}, x \in [0, T_{ck}) ,$$

and the pdf for $l$ is given by:

$$pdf\ (l) : g(y) = \lambda e^{-\lambda y}, y \in [0, +\infty) .$$

Because the error latency is independent of the error occurrence, $a$ and $l$ are independent random variables. Then, we can derive the pdf for $a+l$ as follows:

$$pdf\ (a+l) : h(x, y) = f(x) g(y), x \in [0, T_{ck}), y \in [0, +\infty) .$$

![Figure 26: Timeline of Checkpoint Corruption Scenarios](image)

Note that any checkpoint taken after the occurrence of an unmasked error and before the detection of the error (e.g., $chkpt1$ in Figure 26(a)) must be corrupted. Otherwise, if $chkpt1$ was in a correct state, then the error detection/failure in Figure 26(a) could not have happened; there are no other
errors. In our model, this condition of checkpoint corruption is represented as $a < T_{ck} \& a + l > T_{ck}$.

The probability of the error corrupting this checkpoint is:

$$P\{a < T_{ck} \& a + l > T_{ck}\} = P\{a + l > T_{ck}\} = 1 - P\{a + l \leq T_{ck}\} \text{ for } a \in [0, T_{ck}) \ .$$

$$P\{a + l \leq T_{ck}\} = \int_{x+y \leq T_{ck}} h(x, y) \, dx \, dy = \int_{x+y \leq T_{ck}} f(x) g(y) \, dy \, dx$$

$$= \int_{0}^{T_{ck}} \int_{0}^{T_{ck} - x} \lambda e^{-\lambda y} \, dy \, dx = \int_{0}^{T_{ck}} \frac{1}{T_{ck}} (1 - e^{-\lambda(T_{ck} - x)}) \, dx = 1 - \frac{1}{\lambda T_{ck}} (1 - e^{-\lambda T_{ck}})$$

Consequently,

$$P\{a < T_{ck} \& a + l > T_{ck}\} = \frac{1}{\lambda T_{ck}} (1 - e^{-\lambda T_{ck}}) \tag{5}$$

**Selecting $T_{ck}$ to cover short-lived errors.** To mitigate checkpoint corruption, we want to select the checkpoint interval to be larger than the latency of most errors. To do that, it is crucial to get realistic estimates of error latency. How does one obtain such estimates in practice? Two methods come to mind: (i) analyze detection characteristics of detectors deployed in the system/application and (ii) inject faults into the target application/system to measure error latency. For example, according to Gu et al. [39], 95% of Linux kernel crashes have error latency of less than 100M CPU cycles (or 50 ms on 2G Hz processors). If we use this data in our model, then

$$P\{l \leq T_{B}\} = p = 0.95 \text{ for } T_{B}=50 \text{ ms. \ As } P\{l \leq T_{B}\} = 1 - e^{-\lambda T_{B}} \text{ we get } \lambda = 0.0599 \text{ (1/ms) for the test system in [39].}$$

If we select 50 ms as the checkpoint interval $T_{ck}$, i.e., a value covering 95% of error latency, the probability of checkpoint corruption is 31.7% (computed using formula (5)). For $P\{l \leq T_{B}\} = 1 - e^{-\lambda T_{B}} = 0.99$ and $\lambda = 0.0599$, $T_{B}=77$ ms. So when we increase the checkpoint interval to 77ms to cover 99% of error latency, the probability of checkpoint corruption is reduced
to 21.5%.

**Dual checkpoint.** Selecting the checkpoint interval to be larger than the latency of most errors reduces checkpoint corruption probability. However, it does not ensure that any error occurrence with latency less than the checkpoint interval does not corrupt a checkpoint. A dual-checkpoint scheme (as shown in Figure 26 (b)) is necessary to provide this assurance. In this scheme, two checkpoints are kept at any time, and the older of the two (*chkpt1* in Figure 26 (b)) is rolled back to during recovery. In this scenario, *chkpt1* is corrupted only when  \( a < T_{ck} \) \& \( a + l > 2T_{ck} \). Then the probability of checkpoint corruption is:

\[
P(a < T_{ck} \& a + l > 2T_{ck}) = P(a + l > 2T_{ck}) = 1 - P(a + l \leq 2T_{ck}) \text{ for } a \in [0, T_{ck}).
\]

\[
P(a + l \leq 2T_{ck}) = \int_{x+y \leq 2T_{ck}} h(x, y)dxdy = \int_{x+y \leq 2T_{ck}} f(x)g(y)dxdy
\]

\[
= \int_{0}^{T_{ck}} \int_{0}^{2T_{ck}-x} \lambda e^{-\lambda y}dydx = \int_{0}^{T_{ck}} \frac{1}{T_{ck}} (1-e^{-\lambda (2T_{ck}-x)})dx = 1 - \frac{e^{-\lambda T_{ck}}}{\lambda T_{ck}} (1-e^{-2\lambda T_{ck}})
\]

Consequently,

\[
P(a < T_{ck} \& a + l > 2T_{ck}) = \frac{e^{-\lambda T_{ck}}}{\lambda T_{ck}} (1-e^{-2\lambda T_{ck}})
\]

\[
(6)
\]

Table 14 lists the checkpoint corruption probabilities for different error latency percentiles in single-checkpoint and dual-checkpoint scenarios. When dual-checkpoint is used for \( T_{ck} \) of 50 ms (covering latency of 95% of errors), the checkpoint corruption probability is largely reduced to 1.59% (using formula (6)).

---

23 Note that successfully taking the latter checkpoint implies that the system has successfully executed a period no less than \( T_{B} \) since the former checkpoint, and the former checkpoint must not have been corrupted by an error with error latency less than \( T_{B} \).
Without knowledge of error latency, imprecise selection of $T_{ck}$ may result in a large checkpoint corruption probability even when multi-checkpoint schemes are deployed. For example, if two checkpoints are kept but $T_{ck}$ is selected as 20 ms (70% of error latency in our example data), the probability of checkpoint corruption is 17.6%, much larger than the 1.59% when 95% of the error latency distribution is covered.

Table 14: Checkpoint Corruption Probabilities in Different Scenarios

<table>
<thead>
<tr>
<th>Error latency percentile ($p$)</th>
<th>$p$-percentile point of error latency at $T_{ck}$ (ms)</th>
<th>Prob. of checkpoint corruption in single-checkpoint at $T_{ck}$</th>
<th>Prob. of checkpoint corruption in dual-checkpoint at $T_{ck}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>70%</td>
<td>20</td>
<td>58.3%</td>
<td>17.6%</td>
</tr>
<tr>
<td>90%</td>
<td>38</td>
<td>39.1%</td>
<td>4.05%</td>
</tr>
<tr>
<td>95%</td>
<td>50</td>
<td>31.7%</td>
<td>1.59%</td>
</tr>
<tr>
<td>99%</td>
<td>77</td>
<td>21.5%</td>
<td>0.21%</td>
</tr>
<tr>
<td>99.9%</td>
<td>115</td>
<td>14.5%</td>
<td>0.015%</td>
</tr>
</tbody>
</table>

7.2. Availability Model

In this section we construct a Markov model to study the availability improvement provided by VM-$\mu$Checkpoint to protected VMs. The model captures failure and recovery behavior of all the involved components. Specifically, the following failures are modeled:

a) Transient failure of a protected VM (or an application in the VM): The VM is successfully recovered by the checkpointing VM if there is no checkpoint corruption. When there is checkpoint corruption, the VM cannot be recovered by VM-$\mu$Checkpoint, and a new VM is started on the same physical host. The jobs being executed at the time of the failure are resubmitted from the beginning. The checkpoint corruption probability derived from the checkpoint corruption model above is multiplied by the failure rate of a protected VM to obtain the rate of the failures with checkpoint corruption. The difference between the failure rate of the
protected VM and the failure rate with checkpoint corruption is the rate of failures successfully recovered by VM-μCheckpoint.

b) Transient failure of the checkpointing VM: The checkpointing VM is restarted from failure on the same physical host and begins to receive checkpoints from protected VMs. During this procedure, the protected VM is still available for job execution. When a protected VM fails during the failure/restart of the checkpointing VM, our recovery protocol first restarts the checkpointing VM and then restarts the protected VM (and as before, interrupted jobs are restarted from beginning).

c) Failure of the hypervisor and permanent failure of the protected VM or checkpointing VM: A hypervisor is started either on the same physical node or on another node, the checkpointing VM is started on the hypervisor, and then the protected VM is started (we assume that the disk images of VMs can be loaded from any physical host, which is true in most current virtualized environments).

Exponential distribution is assumed for the time to failure and the recovery time for all the components. For ease of explanation and brevity, here we present the availability model for only one protected VM on top of the hypervisor. A generalized model for $n$ protected VMs on top of the hypervisor is described in Section 7.2.3. The following notations are used in the model:

$\lambda_v$  Rate of the hypervisor software failure and all permanent failures.

$\lambda_s$  Failure rate of the checkpointing VM alone.

$\lambda_p$  Failure rate of the protected VM alone.
$r_v$ Restart rate of the hypervisor software.

$r_s$ Rate of restarting the checkpointing VM, including saving the first checkpoint.

$r'_p$ Rate of successfully recovering a protected VM by VM-$\mu$CheckPoint.

$r_p$ Rate of recovering a protected VM when VM-$\mu$CheckPoint fails to do that (due to checkpoint corruption). Job recomputation is considered as recovery overhead.

$p_c$ Probability of checkpoint corruption given an error.

The Markov model in Figure 27 depicts the failure/recovery behavior of the system with VM-$\mu$CheckPoint deployed. The state of the system is denoted as a vector $(k, j)$, where $k$ represents the state of the protected VM and $j$ indicates the state of the checkpointing VM (see Figure 27 for more information on the state representation). The model consists of two submodels: (a) the submodel for transient failures of the protected VM and the checkpointing VM and (b) the submodel for permanent failures and hypervisor failures.

Figure 27 captures all the failure/recovery behavior described above. For example, when a
permanent failure occurs during execution of the protected VM, the entire physical node fails and a hypervisor is restarted on a physical node (shown as a sequence of transitions, $(1,1)\rightarrow F_S \rightarrow (0,F)$, in submodel (b)). Then the checkpointing VM and the protected VM (including interrupted jobs) are recovered in turn (shown as $(0,F)\rightarrow (0,0)\rightarrow (1,1)$ in submodel (a)). The failure/recovery path in the model is highlighted in Figure 27.

The Markov model is solved by computing the equilibrium condition, i.e., the “input flow” into each state equal to the “output flow” out of the state [46]. We use the mathematics tool package CLAPACK [47] to solve these equations and obtain the probability of the system’s staying in each state.

### 7.2.1. Model of Disk-Based Checkpointing

We extended VM-μCheckpoint to support disk-based checkpointing in Section 5.2.2. Then we also extended the availability model (Figure 27) to capture the behavior of recovering VMs from checkpoints in disk. The following notations are used in the model:

- $\lambda_v$ Rate of the hypervisor software failure and all permanent failures.
- $\lambda_s$ Failure rate of the checkpointing VM alone.
- $\lambda_p$ Failure rate of the protected VM alone.
- $r_v$ Restart rate of the hypervisor software.
- $r_s$ Rate of restarting the checkpointing VM, including saving the first checkpoint.
- $r_{pm}$ Rate of successfully recovering a protected VM from in-memory checkpoint.
\( r_{pd} \) Rate of recovering a protected VM from in-disk checkpoint when in-memory checkpoint is not available.

\( r_{pr} \) Rate of restarting a protected VM when in-disk checkpoint is not available. Job recomputation from the beginning is considered as recovery overhead.

\( p_{cm} \) Probability of corrupting an in-memory checkpoint by an error.

\( p_{cd} \) Probability of corrupting an in-disk checkpoint by an error.

The extended model is given in Figure 28. Figure 28(a) illustrates the failure and recovery of VMs, while Figure 28(b) illustrates the failure and recovery of the hypervisor or the node.
7.2.2. Availability Study

After the model is solved, we obtain the availability of the protected VM by adding up the probabilities of the system staying in the states (1,1) and (1,F). To demonstrate the performance of our technique in availability enhancement, we also construct the Markov models for both the baseline case and an existing technique of VM checkpointing based on live migration (Remus [38]) for comparison.
In the baseline case, a VM is on top of the hypervisor and there is no checkpoint of the VM. When a failure occurs to the VM, the VM is restarted with all interrupted jobs started from the beginning. Remus maintains a backup of a VM on a remote host by migrating the state from the primary to the backup periodically (e.g., every 50 ms). When the VM fails, the backup VM begins to execute from the last checkpoint. The detailed behaviors of the baseline and Remus as well as their models are presented in Sections 7.2.4 and 7.2.5.

**Model parameters.** The parameters selected in our model are based on previous empirical study of off-the-shelf servers. The availability study for Windows servers [48] reports that, though the average availability of the servers is around 99.9% (without considering job recomputation), there are also servers with availability around 99% or less. The authors also report the MTTR (mean time to recovery) for all the failures as 0.25 hour (or 15 minutes). This MTTR includes the response time of an administrator who discovers the failure and restarts the failed machine with appropriate recovery.

So in the availability model for VM-μCheckpoint we set the following parameters:

\[ r_v = r_s = 1/15 \text{ min.} \]

\[ r_p = 1/(0.5*\text{average job duration}), \text{ because the VM is restarted immediately and the mean job recomputation during recovery is half of the average job duration.} \]

\[ r'_p = 1/600 \text{ ms} = 100, \text{ as overhead around 600 ms is measured in our experiments.} \]

\[ \lambda_v = 1/15000 \text{ hours} + 1/3 \text{ years} = 0.000001492/\text{min}. 1/15000 \text{ hours} = 1/1.712 \text{ years is the hypervisor failure rate; 1/3 years is the permanent failure rate (according to presentations made by several Intel engineers in DARPA and other forums).} \]

\[ \lambda_s = 1/15000 \text{ min, for the checkpointing VM with 99.9% availability} (r_s = 1/15 \text{ min}). \]

\[ \lambda_p = 1/1500 \text{ min (for a protected VM with 99% availability without considering job recomputation) or 1/15000 \text{ min (for a protected VM with 99.9% availability).}} \]
$p_c = 1.59\%$; the value is derived in the checkpoint corruption model for $T_{ck} = 50$ ms, which covers 95\% of error latency.

In this parameter selection, $\lambda_v$ is much smaller than $\lambda_s$ or $\lambda_p$ because hardware and hypervisor are usually assumed to be much more reliable than the server software and the operating system. This is a realistic assumption (also assumed in [49]) because the hypervisor kernel is small (e.g., 434KB for Xen-3.3.1 vs. 1.5MB for Linux 2.6.18), and hence, verification and test of the hypervisor code is relatively easy.

The parameter values above are also used in the availability models for the baseline and Remus (so the recovery rate of Remus is also 1/600 ms), except that a different $p_c$ value is selected in the model for Remus. For the example data used in Chapter 3, the checkpoint corruption probability is 31.7\% in a single-checkpoint scheme at the checkpoint interval of 50 ms, if the exponential distribution of the error latency is assumed. According to an experimental study in [50], the probabilities of checkpoint corruption range from 27\% to 41\% for different application workloads. We select $p_c = 15\%$ in the Remus model for fair comparison (note that $p_c$ is probability of checkpoint corruption given an error).

**Results.** The availability values computed from these models are compared in Table 15. Our results are better than Remus’s for all the experiment cases. For example, for average job duration of 8 hours (i.e., $1/r_p = 240$ min) on a 99\%-available server ($\lambda_p = 1/1500$ min), we achieve an availability of 99.7\%, while Remus achieves 97.7\%. There are two reasons for our better results:

i) **Transient failures are much more frequent than permanent failures.** So the Remus’s capability of tolerating permanent failures via a backup copy on a remote host demonstrates only slightly more availability than our technique, especially for jobs lasting a couple of hours or less. Table 16 shows the availability results if there is no checkpoint corruption in either model. For an
average job duration of 8 hours on a 99%-available server, the availability of the protected VM is increased from 99.94% to 99.996% by introducing the remote host for tolerating permanent failures.

### Table 15: Availability Comparison with Checkpoint Corruption (note that $I/r_p = 0.5*\text{average job duration}$)

<table>
<thead>
<tr>
<th>$\lambda_p/\text{min}$</th>
<th>$I/r_p$ (min)</th>
<th>15</th>
<th>60</th>
<th>240</th>
<th>1440</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_p=1/1500$min</td>
<td>VM-uchkpt</td>
<td>99.98%</td>
<td>99.92%</td>
<td>99.7%</td>
<td>98.2%</td>
</tr>
<tr>
<td></td>
<td>Remus</td>
<td>99.8%</td>
<td>99.4%</td>
<td>97.7%</td>
<td>87.4%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>99.0%</td>
<td>96.1%</td>
<td>86.2%</td>
<td>51.0%</td>
</tr>
<tr>
<td>$\lambda_p=1/15000$min</td>
<td>VM-uchkpt</td>
<td>99.99%</td>
<td>99.98%</td>
<td>99.93%</td>
<td>99.6%</td>
</tr>
<tr>
<td></td>
<td>Remus</td>
<td>99.98%</td>
<td>99.94%</td>
<td>99.76%</td>
<td>98.6%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>99.90%</td>
<td>99.6%</td>
<td>98.4%</td>
<td>91.1%</td>
</tr>
</tbody>
</table>

ii) The impact of checkpoint corruption on availability is much larger than that of permanent failures in high-frequency checkpointing. VM-µCheckpoint loses little in handling permanent failures, but it gains much more in reducing checkpoint corruption by (a) selecting the proper checkpoint interval to cover latency of 95% of errors and (b) applying a dual-checkpoint scheme to provide an assurance of 100% coverage for these 95% of errors (another 5%-1.59%=3.41% of rest of the errors are also covered, according to our model).

### Table 16: Availability Comparison without Checkpoint Corruption

<table>
<thead>
<tr>
<th>$\lambda_p/\text{min}$</th>
<th>$I/r_p$ (min)</th>
<th>15</th>
<th>60</th>
<th>240</th>
<th>1440</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_p=1/1500$min</td>
<td>VM-uchkpt</td>
<td>99.99%</td>
<td>99.98%</td>
<td>99.94%</td>
<td>99.7%</td>
</tr>
<tr>
<td></td>
<td>Remus</td>
<td>99.97%</td>
<td>99.97%</td>
<td>99.96%</td>
<td>99.92%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>99.0%</td>
<td>96.1%</td>
<td>86.2%</td>
<td>51.0%</td>
</tr>
<tr>
<td>$\lambda_p=1/15000$min</td>
<td>VM-uchkpt</td>
<td>99.99%</td>
<td>99.99%</td>
<td>99.96%</td>
<td>99.8%</td>
</tr>
<tr>
<td></td>
<td>Remus</td>
<td>99.99%</td>
<td>99.99%</td>
<td>99.98%</td>
<td>99.97%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>99.90%</td>
<td>99.6%</td>
<td>98.4%</td>
<td>91.1%</td>
</tr>
</tbody>
</table>

### 7.2.3. Generalized Availability Model for VM-µCheckpoint

We generalize the availability model of VM-µCheckpoint, as described in Figure 27, to cases when there are $n$ VMs on top of a hypervisor. The state of the system at any time is denoted as a
vector \((k, j)\) in the generalized model, where \(k\) is the number of the \(n\) protected VMs that are available and \(j\) is the number of the \(n\) protected VMs whose checkpoints are kept in the checkpointing VM. The system stays in the state \((n, n)\) in the failure-free situation.

The generalized model consists of a number of submodels, illustrated in Figure 29.

The submodels are as follows:

a) Those that capture the transient failure and recovery of protected VMs when the checkpointing VM is available, as well as the failure of the checkpointing VM. Figure 29 (a) illustrates such a submodel for those states where the checkpointing VM keeps checkpoints for \(m\)
protected VMs. In this figure, we can see that the state \((m, m)\) transitions to the state \((m-1, m-1)\) when there is checkpoint corruption associated with the failure, while it transitions to the state \((m-1, m)\) when there is no checkpoint corruption. There are \(n+1\) such submodels for capturing the states with the \(m\) value ranging from 0 to \(n\).

b) The submodel that captures the transient failure and recovery of protected VMs when the checkpointing VM is unavailable, as well as the recovery of the checkpointing VM (Figure 29 (b)).

c) The submodel that captures permanent failures and hypervisor failures, as well as the corresponding recovery (Figure 29 (c)).

So there are \(n+3\) submodels in the generalized availability model in Figure 29. We employ the CLAPACK tool package to compute the equilibrium condition for all the model states and solve the model.

7.2.4. Availability Model for Baseline

To evaluate the availability improvements VM-\(\mu\)Checkpoint provides, we also model the baseline case for comparison purposes (shown in Figure 30). The model captures scenarios with \(n\) virtual machines on top of a hypervisor. There is no checkpointing of VMs. When a failure occurs to a VM, the VM is restarted, with all interrupted jobs started from the beginning.
7.2.5. Availability Model for Remus

Remus maintains a backup of a VM on a remote host by migrating the state from the primary to the backup periodically. When the VM fails, the backup VM begins to execute from the last checkpoint. We design a Markov model for Remus to compare the availability enhancements provided by VM-μCheckpoint and Remus.

Due to the complexity of the Markov model for Remus, here we only describe the model for the cases in which there is only one VM on top of a hypervisor. Figure 31 presents this model and explains how the system state is represented in the model. The $p'_{c}$ in the figure denotes the probability of checkpoint corruption in Remus. The following paragraphs briefly describe part of the model (the highlighted states and transitions in the figure) to illustrate how the system behavior is modeled.
Figure 31: The Availability Model for Remus (1 Protected VM)

The system stays in the state \((1, 1)\) during normal behavior. When a failure occurs to the VM on the primary host and that failure does not cause checkpoint corruption, the backup host resumes the execution of the VM from the last checkpoint immediately. As a result, the system transitions to the state \((0, 1)\). We assume that the previous primary host automatically sets up a backup VM, i.e., begins to collect a checkpoint of the current executing system after the failover, at the rate \(r'_p\).

The previous primary host is the backup host now, and the state \((0, 1)\) transitions to \((1, 1)\) at the rate \(r'_p\).

If a VM failure causes checkpoint corruption, the backup host cannot resume the execution of the VM successfully. So the state \((1, 1)\) transitions to the state \((0, 0)\). At this time, the VM is restarted on the primary host and all the jobs in the VM are restarted from the beginning at the rate \(r_p\). The system state becomes \((1, 0)\). The backup host begins to collect a checkpoint of the VM from the primary host. So the state \((1, 0)\) transitions to the state \((1, 1)\).

When a permanent failure occurs to the primary host, the system transitions from \((1, 1)\) to \((F, 1)\), in which state the VM is still available in the backup host. Note: an independent failure may occur to the backup host at the same time. So the state \((1, 1)\) independently transitions to the state \((1, F)\).
at the rate \( \lambda_V \). Remus tolerates a single permanent failure, as the VM is available in \((F, I)\) and \((I, F)\).

The probability of the system staying in each state of the model is obtained by solving the availability model. The system availability is computed as the sum of the probabilities of the system staying in all states \((k, j)\) where either \(k\) or \(j\) is 1.
CHAPTER 8

RELATED WORK

This dissertation consists of a number of topics including the Reliability MicroKernel framework, system hang detection, application/VM crash detection, and VM checkpoint, as well as experimental evaluation and formal model based analysis of these designs and implementations. Here we give the related work on the RMK, system hang detection, and VM checkpoint, which are the main contributions of this dissertation.

8.1. Related Work on RMK and System Hang Detection

Table 17 summarizes representative examples of studies (in academia and industry), and systems (commercial, and research prototypes) that address issues of providing reliability services to applications using hardware, and system support. Microkernel systems such as Mach [51] and Chorus [52], [53] provide basic resource management and communications rather than explicitly focus on reliability. Reliability architectures in IBM AIX [1], and High-Availability Linux [2] are designed to be closely coupled with the systems. They mostly concern system reliability rather than exploiting application characteristics to improve application reliability. Hardware reliability frameworks, such as the Reliability and Security Engine (RSE) [54], provide application-aware mechanisms using programmable hardware modules to support the detection/recovery of runtime errors.
## Table 17: List of Work Related to RMK

<table>
<thead>
<tr>
<th>Category</th>
<th>Study/System</th>
<th>Reliability Features</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microkernel Architecture</strong></td>
<td>Mach [51]</td>
<td>Reliability is not the primary focus.</td>
<td>Architectural basis to build other OSs.</td>
</tr>
<tr>
<td></td>
<td>Chorus [52]</td>
<td>Reconfigurable microkernel; applies event/exception handling for reliability; predicates are defined in wrappers of functional services to guard against incorrect state and error propagation.</td>
<td>Architectural basis to build other OSs. Wrappers of microkernel services used to check system health rather than support application-aware techniques.</td>
</tr>
<tr>
<td></td>
<td>SGI IRIX</td>
<td>Process checkpointing dumps process image to disk.</td>
<td>Needs OS-level support for preserving the entire process image.</td>
</tr>
<tr>
<td></td>
<td>IBM AIX [1]</td>
<td>Virtual-server-based system protection; application hang detection; a daemon polls the OS kernel to check if processes are being starved.</td>
<td>OS-level support focused on system reliability.</td>
</tr>
<tr>
<td><strong>Reliability Architecture</strong></td>
<td>High-Availa-</td>
<td>Heartbeat used for detection of node failures; membership protocol used for group communication.</td>
<td>Needs support for system reliability, especially cluster health.</td>
</tr>
<tr>
<td></td>
<td>bility Linux [2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sentry [56]</td>
<td>Additional layer placed on top of OS provides error masking, and system service guard.</td>
<td>Supports system reliability rather than application.</td>
</tr>
<tr>
<td></td>
<td>ARMOR [35]</td>
<td>Self-checking middleware provides fault tolerance to applications.</td>
<td>Application instrumentation is required for implementing application-specific reliability techniques.</td>
</tr>
<tr>
<td><strong>Hang Detection of OS or Applications</strong></td>
<td>RSE [57][54]</td>
<td>Processor-level framework provides application-aware error detection, e.g., detection of OS, and application hangs using hardware modules.</td>
<td>Reliability mechanisms are bound to hardware modules; needs OS support for application hang detection.</td>
</tr>
<tr>
<td></td>
<td>Watchdog Device [17]</td>
<td>System hang detection: an external PCI card is used as watchdog.</td>
<td>Extra hardware is required; long latency (timeout sometimes in minutes).</td>
</tr>
<tr>
<td></td>
<td>KHM [16]</td>
<td>System hang detection: a process periodically sets a mark, and the timer interrupt handler checks the mark to detect system hangs.</td>
<td>Fails when interrupt is disabled; long latency when system is heavy loaded; large overhead.</td>
</tr>
</tbody>
</table>
### Table 17: Continued

<table>
<thead>
<tr>
<th>Application Checkpointing</th>
<th>System hang detection: if there is no timer interrupt within a few seconds, the system hang is detected.</th>
<th>Fails when the system hangs with timer interrupts arriving and handled.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libckpt [9], LibFT [14]</td>
<td>Libraries invoked by applications dump application states and/or critical data.</td>
<td>Not application-transparent; user-level knowledge is required for high performance.</td>
</tr>
<tr>
<td>Zap [58]</td>
<td>Virtual machine solution transparently checkpoints applications.</td>
<td>Checkpoints the entire process image.</td>
</tr>
<tr>
<td>Epckpt [59]</td>
<td>New system calls added to the kernel provide transparent checkpoint.</td>
<td>Checkpoints the entire process image.</td>
</tr>
<tr>
<td>Incremental Checkpointing [60]</td>
<td>Application-transparent incremental checkpoints are provided for main memory database.</td>
<td>Application-used library is instrumented to support incremental checkpoints; custom solution for MMDB applications.</td>
</tr>
<tr>
<td>Cache-based Checkpointing [61]</td>
<td>Checkpoint is triggered on cache misses; the checkpoint state is stored in memory and includes processor registers and cache.</td>
<td>Checkpoint interval is very short; error may propagate across checkpointing; large overhead; additional hardware.</td>
</tr>
<tr>
<td>Coordinated Checkpointing [62] [63]</td>
<td>Checkpoint protocols are provided for distributed applications.</td>
<td>Checkpoints the entire process image; focuses on checkpoint protocol.</td>
</tr>
<tr>
<td>Shadow Process Checkpointing [10]</td>
<td>Shadow process is forked to store the process image upon a checkpoint; copy-on-write is applied for dirty page copying.</td>
<td>Important process states like pid not preserved; inefficient copying of pages with only one/two writes; unnecessary allocation of process resources.</td>
</tr>
<tr>
<td>Compiler-Assisted Checkpointing [64]</td>
<td>Compiler inserts checkpoint functions in programs according to user-provided directives for checkpointing critical data.</td>
<td>Not application-transparent; user-level directives are required.</td>
</tr>
</tbody>
</table>

Most existing techniques for detecting hangs of the OS, and/or applications are timer-based; and the timeouts are fixed values that are either preset, or derived from profiling. As a result, the timeout values are often very conservative, and cause large detection latency. Some existing
detection mechanisms cannot operate when the interrupts are disabled due to system hangs, e.g., KHM [16]. Others, like the Linux NMI watchdog timer [18], do not detect hangs if the timer interrupt continues to function despite the system hang.

As there are many checkpoint techniques, we do not list all of them in the table. Some checkpoint mechanisms preserve the entire process image for failure recovery, or process migration [59], [58], while others provide incremental checkpointing [9], [14], [61], [10]. Libckpt [9] checkpoints dirty pages of a process. Mechanisms in [9] and [14] allow users to specify critical data for checkpointing; these require application instrumentation. While an enhanced compiler can be used to automate the application instrumentation, user-provided directives are still needed to identify the location (in the application code) where the instrumentation should be added [64]. Cache-based checkpointing [61] needs additional hardware to enable storing of application-relevant states. More importantly, none of these checkpointing schemes uses OS-level knowledge to avoid inconsistency between the application process image, and the corresponding system state. For example, a checkpoint taken when the target application has pending I/O operations may be inconsistent with the I/O status at the time of recovery upon failures. Our checkpointing scheme solves this inconsistency.

8.2. Related Work on VM Checkpoint

Checkpoint and rollback techniques have been extensively studied in the literature. Checkpoints can be taken in different levels (application, runtime library, compiler, operating system, virtual machine, or hardware). Here we focus on checkpoint techniques in the virtual machine level, as they are more relevant to our objective.
Table 18: The Three Categories of Existing Mechanisms for VM Checkpoint

<table>
<thead>
<tr>
<th>Mechanisms</th>
<th>Brief Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop-and-save</td>
<td>Stops a VM completely, and saves its state to persistent storage</td>
<td>i) large system downtime; ii) provided by all major VMM systems</td>
</tr>
<tr>
<td>Low-freq (e.g. interval of hours or longer) based on live migration, e.g. CEVM, VNsnap, VM snapshots</td>
<td>Creates a VM replica on a remote node via live migration, then the remote node writes the replica to disk</td>
<td>i) Significant recomputation during recovery as checkpoint frequency is low; ii) Large overhead of maintaining a full replica for a protected VM</td>
</tr>
<tr>
<td>High-freq (interval of 10~1000 ms) based on live migration, e.g. Remus</td>
<td>Maintains a VM replica on a separate physical node via live migration, and fail-overs upon a failure</td>
<td>i) Large overhead while migrating latest updates to the remote node continuously (around 50% overhead for 50ms checkpoint interval); ii) maintaining a full replica for a protected VM; iii) fail-stop assumption</td>
</tr>
</tbody>
</table>

**VM checkpointing.** Table 18 lists the existing mechanisms of VM checkpoint. Basically there are three categories of VM checkpoint: (i) the traditional stop-and-save way of VM checkpoint, (ii) low-frequency VM checkpoint based on live migration, and (iii) high-frequency VM checkpoint based on live migration.

In the first category of VM checkpoint, we stop a VM completely and save its state in persistent storage, and then resume the VM. This way incurs a large system downtime during the checkpoint. Then live migration is leveraged to avoid this large downtime.

Most existing VM checkpoint/replication techniques are based on live migration of VMs (e.g., VMWare VMotion [65] and Xen Live Migration [3]), which continually transmit dirty pages of a VM from a source node to a destination node. These techniques exploit the live migration mechanism for the purposes of VM checkpointing, VM rejuvenation, load-balancing, and fast VM forking.

CEVM [36], VNsnap [37], and VM Snapshots [66] are techniques of the second category of VM
checkpoint in Table 18. These techniques employ VM live migration or copy-on-write to create a replica image of a VM with low downtime incurred; then the image is written to disk in the background or by the separate physical node. An ongoing project on VM checkpoint [66] tries to provide a generic API in Xen product for saving a VM snapshot to disk on demand. Basically, the VM memory is scanned and saved to files while the VM runs simultaneously. Copy-on-write is exploited to save the original data of modified VM state during checkpointing.

VM-µCheckpoint is different from these disk-based VM checkpointing schemes in that we aim at (i) providing high-frequency checkpointing and rapid recovery of VMs with low overhead, which allows VM failures to be masked to clients, and (ii) proposing a mechanism to alleviate checkpoint corruption in high-frequency checkpointing (checkpoint corruption has more impacts on service availability than permanent failure, as shown in our model). Disk-based VM checkpointing is too costly and is unable to keep up with the high frequency of rapid checkpointing (tens per second).

The third category of VM checkpoint is the high-frequency VM checkpoint based on live migration. The existing approach in this category is Remus [38], which maintains a backup VM on a separate physical node by periodically transmitting the VM’s dirty pages to the backup. Similar to VM-µCheckpoint, Remus is a mechanism of high-frequency VM checkpointing and failover. But VM-µCheckpoint focuses on error behavior and reliability/availability improvement, while Remus focuses on migration overhead. No study of error behavior or reliability/availability is reported in [38]. As checkpoint corruption is not handled in Remus (fail-stop errors are assumed), our technique is better in improving service availability. Moreover, there is large overhead in high-frequency migration-based VM checkpoint because the latest state updates are migrated to the remote node continuously. For example, our checkpoint algorithm incurs around
6.3% overhead at the checkpoint intervals of 50 ms, while Remus incurs around 50% overhead for the same checkpoint frequency.

Other techniques that may be relevant to VM checkpointing are briefly described as follows. Bradford et al. [67] focus on migrating persistent state of a guest system across WAN so that the guest system can migrate to a node that does not share storage with the original node. [68] revises Xen live migration to fit in a self-migration scenario. Distler et al. [69] and Nagarajan et al. [70] implement proactive VM rejuvenation based on live migration, and [71] uses live migration for load-balancing. Potemkin [72] employs copy-on-write to share data between VMs for efficiently provisioning VMs. Another technique of fast VM forking is [73]. Though one may use these techniques to checkpoint a VM by periodically forking a shadow VM and tearing down out-dated shadow VMs, spawning a VM and tearing down a VM involve a lot of overhead that is not necessary for checkpointing. Moreover, error analysis and reliability/availability study is an integral part of a checkpointing technique for failure mitigation.

**Multi-checkpoint mechanisms.** As far as we know, none of the existing checkpoint techniques considers handling checkpoint corruption by explicitly including error latency bound as a parameter, though multi-checkpoint mechanisms can be leveraged to deal with checkpoint corruption. IBM System Z [74] allows multiple checkpoints of an application to be recorded in persistent storage on demand. Ping-Pong checkpoint [75] maintains two checkpoints to deal with incomplete checkpoint due to errors during the checkpointing procedure, rather than checkpoint corruption due to latency of error detection.

**In-place restoration.** Hardware-level checkpoint techniques [76], [77] use special hardware to take and store a checkpoint. When an error is detected, the checkpoint saved in the special
hardware is restored into the architecture state of the physical machine, including register file and memory. For example, the first update of a memory word or a register during a checkpoint interval is preserved in special hardware in SafetyNet [77].
CHAPTER 9

CONCLUSIONS

The thesis describes the Reliability MicroKernel (RMK) framework, a loadable kernel module for providing application-aware reliability and configuring reliability mechanisms. The RMK prototype has been implemented in native systems of Linux and Windows on a Pentium 4 processor without recompiling the kernel, and is also extended to the Xen VMM to provide highly available VMs (RMK is encapsulated into a hypercall in Xen, and the Xen hypervisor is recompiled but VMs are not recompiled). By combining the low-latency error detection and transparent checkpoint techniques into the RMK framework, we provide an automatic approach to error detection and recovery for real-world systems.

The RMK on native systems supports the detection of application/system failures, and transparent application checkpointing. The experimental evaluation of the RMK using real-world applications shows that the system hang detection, and application hang detection, which exploit characteristics of application and system behavior, achieve high coverage, and low false-positive rates (1 out of 2000 experiments for application hang detection, and no false positive for system hang detection). Because the OS-level knowledge of applications/system is used, the RMK prototype has a low overhead in providing the transparent application checkpoint (less than 0.1% in the experiments), and application failure detection (0.6% performance overhead).

The hang detection of VMs is adapted from the system hang detection in native systems. Moreover, we proposed VM-µCheckpoint, a lightweight VM checkpointing technique, which
minimizes overhead by placing checkpoints in memory and performing in-place recovery. VM-µCheckpoint also addresses the problem of checkpoints becoming corrupted in high-frequency checkpointing. We showed that it is important to take into account expected durations for errors to manifest themselves in order to determine checkpoint intervals.

A model-based study was conducted to show that VM-µCheckpoint achieves better results than existing migration-based VM checkpointing. For example, for an average job duration of 8 hours in a 99%-available VM (on top of a hypervisor with the MTTF of 1.7 years), we achieve an availability of 99.7%, while the migration-based VM checkpointing achieves 97.7%.

Experimental results show that the proposed technique achieves much better performance than existing techniques based on VM live migration. There is an average of 6.3% overhead in terms of program execution time for the SPEC CINT 2006 benchmark when VM-µCheckpoint is deployed at a checkpoint frequency of 20 times per second. (An approximately 50% overhead is reported in a previous technique [38] at the same checkpoint frequency.) Moreover, the checkpoint size is small in VM-µCheckpoint: an average of 2.6 MB in our experiments when the CoW-P algorithm is applied with 50 ms checkpoint intervals.
APPENDIX A

MODELING COORDINATED CHECKPOINTING FOR
LARGE-SCALE SUPERCOMPUTERS

A.1. Introduction

The computational demands of emerging applications such as protein folding is giving rise to a new generation of supercomputers, consisting of several thousand processors (currently in planning). For example, the newly deployed IBM BlueGene/L [78] is expected to scale to 64K dual-processor nodes. Despite the huge computing power they provide, the large number of nodes makes the system significantly more vulnerable to errors. As a result, the effects of the larger number of failures due to errors can impair the system performance and limit its scalability.

Although a hierarchy of error detection and recovery techniques, like ECC, CRC and message retransmission, can be used to correct some errors/failures, there are transient errors/failures that cannot be covered using these techniques, e.g. the corrupted states due to propagation of undetected errors. For these errors/failures, checkpointing and rollback may be used as the last option to recover the application before rebooting or reconfiguring the system. This work focuses on errors/failures that need checkpointing and rollback to recover.

The most commonly used checkpointing scheme for supercomputing systems is coordinated checkpointing, due to its simplicity of implementation. In this approach, cooperative processors synchronize to ensure a global consistent state before taking a checkpoint [79]. The main
problem with coordinated checkpointing is its lack of scalability as it requires all processors to take a checkpoint simultaneously.

There are two main contributions made by this work. First, it builds a model of a large-scale system that uses coordinated checkpointing for recovery from failures with complex semantics. Second, it studies the scalability and performance of the system for several hundred thousand processors by simulating the model with realistic parameter values.

An important issue considered in our model is the effect of scaling from several thousand processors to several hundred thousand processors, i.e. by two orders of magnitude. Issues such as failures during checkpointing and recovery, correlated failures within the system, and checkpointing overhead due to coordination, are of primary importance for the new generation of supercomputers. This is because their larger number of nodes and higher failure rates invalidate some of the assumptions about system behavior made by existing models [80], [81], [82], [83], [84], [85] and exacerbate some effects considered negligible before. These assumptions are:

• The computation interval and the checkpoint overhead are much smaller compared to the mean time between failures (MTBF). However, large-scale supercomputers experience much smaller MTBFs and much larger checkpoint overheads, and hence failures during checkpointing and recovery can occur and must be taken into account [86].

• Failures are independent of each other. This is not a valid assumption, as Tang and Iyer [87] showed that even a small number of correlated failures increase system unavailability considerably.
• The overhead of inter-processor coordination for checkpointing is negligible. However, as the number of nodes increases the coordination overhead grows and it cannot be ignored.

A measure called useful work similar to accumulated reward [88] is used to evaluate system performance. It is defined as the computation that contributes to the ultimate completion of the job (see definition in Section A.8). If a failure occurs before the computation can be checkpointed, the computation since the last checkpoint needs to be repeated after the recovery, and is not counted as useful work. Accurate modeling of useful work requires knowledge of future behavior of the system and cannot be represented using simple Markov models. Instead, Stochastic Activity Networks (SAN) are used to model the system behavior. The modeling power of SANs allows us to concisely represent complex system phenomena such as checkpoint coordination, failures during checkpointing and recovery, and correlated failures. The SAN model is studied using simulation and the impact of system parameters on system performance and scalability is evaluated.

A.2. Related Work

Checkpointing Models. One of the earliest models for computing the optimal checkpointing interval is by Young [80]. This model assumes that the MTBF of the system is very large compared to the checkpoint and recovery time, and hence does not consider failures during checkpointing and recovery. Daly [81] presents a modification of Young’s model for large-scale systems. This model takes into account failures during checkpointing and recovery, and multiple failures in a single computation interval. However, it does not model the coordination overhead of the checkpointing protocol itself or consider correlated failures.
Kavanagh and Sanders [82] evaluate two time-based coordinated checkpointing protocols based on analytical and simulation models, which take the overhead of coordination into account. However, they do not consider failures during checkpointing and recovery as they assume that the MTBF of the system is much greater than the checkpoint interval.

Plank and Thomason [83] investigate the use of spare nodes to provide redundancy in the system to handle permanent failures. We do not consider permanent failures in our model and assume that all nodes can be recovered by restarting the system from the last-saved checkpoint. Plank and Thomason do not consider the overhead of coordination in the model or the effect of scaling their model to a large number of nodes. A recent paper by Elnozahy et al. [84] extends the work of Plank and Thomason to systems consisting of thousands of nodes. It considers the effects of failures during checkpoint and recovery and multiple failures in a single computation interval. However, it does not consider the effects of coordination among the nodes in the checkpointing protocol, nor does it consider correlated failures.

Vaidya [85] derives an analytical expression for the optimal checkpointing frequency in a uniprocessor system. It distinguishes the checkpoint latency from the overhead of a checkpointing scheme. This model considers failures during checkpointing/recovery but does not take into account the scalability of the checkpointing protocol or the system.

**Large-Scale Systems.** Bronevetsky et al. [89] presents a compiler-based technique for asynchronous, coordinated checkpointing. Agarwal et al. [90] consider an adaptive, incremental checkpointing technique for scientific applications on large-scale systems. Finally, Zhang et al. [91] extensively study failure data analysis in large-scale supercomputing systems and show the
existence of temporal and spatial correlation among failures in large-scale systems. We consider
temporal correlations in our model (correlated failures) but not spatial correlations.

A.3. Target System

This study focuses on a typical abstract structure commonly shared by many supercomputers and
a basic coordinated checkpointing protocol whose variants are applied in the supercomputing
world.

A.3.1. Architecture

Each node of the supercomputing system is a tightly-integrated unit consisting of multiple
processors. For example, Blue-Gene/L has 2 processors per node and ASCI Q has 4 processors
per node. Future systems could have 8, 16 or 32 processors per node.

Usually large-scale supercomputing systems have dedicated nodes for job computation (compute
nodes) and for I/O operations (I/O nodes). A set of compute nodes shares the connections to an
I/O node, and all the I/O nodes are connected to a parallel file system through a separate
connection network. For example, IBM BG/L has 64K compute nodes and 1024 I/O nodes. The
network bandwidth from 64 compute nodes to one I/O node is 350 MB/s, and the bandwidth from
one I/O node to the file system is 1 Gb/s.

Data writes from compute nodes to the file system are performed in two steps: first from compute
nodes to I/O nodes, and then from I/O nodes to file system. The I/O nodes locally buffer the
application data or checkpoint they receive from the compute nodes, and write it to the file system
in the background while the compute nodes continue with the computation. The two steps are
reversed for data reads with the exception that reads cannot be done in the background, as the
application may have to wait for the data to be read before proceeding, depending on the nature of the read.\textsuperscript{24}

A.3.2. Checkpoint Protocol

There are two checkpointing approaches used in supercomputing systems. One is application-based, where a global barrier is explicitly used in the application for saving a global consistent state. This places the burden of checkpointing on the application (e.g. in BlueGene/L \textsuperscript{[78]}). The other approach is system-supported checkpointing (e.g. the algorithm used by Cray in the IRIX OS \textsuperscript{[92]}). Our checkpointing protocol is a system-supported synchronous checkpointing and follows the basic principles of coordinated checkpointing, e.g. Koo and Toueg’s protocol \textsuperscript{[93]}.

In our protocol, a single coordinator node, or master, periodically initiates the checkpointing as follows:

1. The master broadcasts a ‘quiesce’ request to all the compute nodes.
2. On receiving ‘quiesce’ each node quiesces its operations, i.e. stops all its activities at a consistent and interruptible state, and replies ‘ready’ to the master.
3. After receiving ‘ready’ from all the compute nodes, the master broadcasts ‘checkpoint’ to all the compute nodes.
4. On receiving ‘checkpoint’ each compute node dumps its state to an I/O node.
5. When all the compute nodes are done dumping their states, the master broadcasts ‘proceed’ to all the compute nodes, and the I/O nodes begin to write the checkpoint to the file system in the background.
6. On receiving ‘proceed’ each compute node continues its activity from the point where it was quiesced.

Further, a timeout period is specified at the master to avoid waiting indefinitely for the ‘ready’ responses. This indefinite wait can occur as a result of an erroneous or failed node that does not respond to the quiesce request. If all the responses are not received within this time, the master

\textsuperscript{24} While current supercomputing systems may not have this capability, future systems might allow this two-step I/O.
times out and broadcasts a ‘abort’ message to all the compute nodes. Then the compute nodes abandon the checkpointing and proceed with their computation.

A.3.3. Application

The application is a parallel, scientific computing workload composed of multiple computation tasks. Each compute processor runs exactly one task of the parallel application and no other tasks.

Application tasks may be performing computation, communication or I/O at any time. Since most parallel, scientific applications are written using the BSP (Bulk Synchronous Parallel) model [94], the multiple tasks more or less coordinate their actions and behave as one cohesive unit.

The application is instrumented with a number of checkpoint primitives at its safe points (e.g. a global barrier), where it can be safely quiesced, like the end of a loop. For example in IRIX, the programmer inserts checkpoint functions in the source code, and the OS calls these whenever it wants to take a checkpoint.

A task that is doing an I/O write, cannot quiesce until it finishes the I/O operation, as this could leave the I/O in an inconsistent state and possibly corrupt the file system. While there are methods to address this, ensuring global coordination is complicated and the simple approach of non-preemptive I/O is preferred in practice. I/O reads of a task can be stopped for checkpointing at any time, and hence, are not specifically considered in our model.
A.3.4. Failure and Recovery

On the failure of a compute node, the entire application rolls back to the last saved checkpoint and recovers; i.e., we only consider failures that require recovery from a checkpoint. Therefore, permanent/persistent errors are not modeled.

Failures of compute nodes and I/O nodes are always detected without any latency. The mechanism for failure detection is not modeled.

When an I/O node fails, all the I/O nodes need to be restarted. This assumption is reasonable since in the BSP model, the application needs the I/O operations on all the I/O nodes to be completed before continuing the computation.

Failures of the master are not treated differently from a compute node failure, as the master is just another compute node in a real system, and perfect detection is assumed.

As nodes have multiple processors, the node failure rate is the product of the processor failure rate and the number of processors per node. The system parameter MTTF is used to refer to the per-node mean time to failure throughout this appendix unless specified otherwise. Then per-processor MTTF is MTTF*number of processors per node. It is assumed that advanced design and error handling techniques are applied to maintain low node failure rates, e.g. use of multiple cores on a chip.

As there is no consensus on MTTF in the literature, we assume an MTTF value from 1 year to 25 years due to both hardware and software errors, as we note that (i) ASCI-Q has a per-node MTTF of 1 year [84]; (ii) IBM 380 X processor has an MTTF of 8 years [95]; (iii) IBM mainframes have
an MTTF of 25 years; and (iv) IBM G5 processor is advertised with an MTTF of 45 years [96] (hardware failures only).

A.3.5. Correlated Failure

This appendix models two categories of correlated failures: (i) correlated failures due to error propagation only and (ii) generic correlated failures.

For correlated failures due to error propagation, we assume that recovery fully restores the application/system state and propagated errors do not cross recovery boundaries. The error propagation is characterized by a short error burst, which typically impacts the recovery. The duration of the error burst is referred to as the correlated failure window. The system may need to recover several times before a successful recovery [97]. A typical value of the correlated failure rate is 600 times the normal failure rate [87] (see Section A.6).

Correlated failures may also be caused by factors other than error propagation, e.g. common causes such as increases in the node temperature or some environmental phenomena. Usually a hyper-exponential distribution is assumed for modeling generic correlated failures, i.e. the system experiences an independent failure rate and a correlated failure rate alternatively. Unlike correlated failures due to propagation, the semantics of generic correlated failures is not necessarily limited to a short duration, but forms a global view of the system for the entire system life.

A.4. Overall Composition of the Model

The system is decomposed into several sub-systems. Each subsystem is modeled as a separate SAN submodel, and the overall model is obtained by integrating these submodels. All the
compute nodes are modeled as a single unit and all the I/O nodes are modeled as another unit. This allows the model to scale to a large number of nodes without requiring a large simulation time. Table 19 lists the submodels of the entire system and Figure 32 illustrates how these submodels (each oval in Figure 32 represents a SAN submodel) are integrated into an overall model. The arrows in the figure illustrate the logical interactions between the submodels. These interactions are implemented by state sharing between the SAN submodels. The dots in the submodels in Figure 32 indicate the initial position of the tokens in the corresponding SAN. It should be emphasized that Figure 32 is not a state diagram, in that the ovals are not representations of the states of the system at any particular time. The submodels are organized into four modules: computing & checkpointing, failure & recovery, correlated failure, and useful work computation.

**Computing & checkpointing module.** The compute_nodes submodel depicts the computation and checkpointing behavior of the compute nodes in the failure-free mode. While the compute nodes are in execution, the application may be performing either computation or I/O operations and this is represented in the app_workload submodel. The master submodel represents the master node in the coordinated checkpointing protocol. It triggers and coordinates the checkpointing, as modeled in the compute_nodes submodel. The coordination among the compute nodes is modeled in the coordination submodel. The io_nodes submodel captures the I/O operations conducted by I/O nodes. It receives data from the compute_nodes submodel, writes/reads checkpoints to/from the file-system, and writes data on behalf of the application in the app_workload submodel. These five submodels form the computing & checkpointing module of the system model and are further described in Section A.5.
## Table 19: Submodel List

<table>
<thead>
<tr>
<th>Module</th>
<th>Submodel</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing &amp; Checkpointing</td>
<td>App_workload</td>
<td>Application state: performing computation or I/O operations</td>
</tr>
<tr>
<td></td>
<td>Compute_nodes</td>
<td>Compute processor state in the checkpoint cycle: executing (including both application’s computation and I/O operations), quiescing, or checkpoint dumping</td>
</tr>
<tr>
<td></td>
<td>Coordination</td>
<td>Coordination procedure for checkpointing</td>
</tr>
<tr>
<td></td>
<td>io_nodes</td>
<td>I/O processor state: idling (including data transmission between compute nodes), writing application data, writing checkpoint, or reading checkpoint; if checkpoint is locally buffered</td>
</tr>
<tr>
<td></td>
<td>Master</td>
<td>System checkpointing state: if checkpointing is started or not</td>
</tr>
<tr>
<td>Failure &amp; Recovery</td>
<td>comp_node_failure</td>
<td>Failure behavior of compute nodes</td>
</tr>
<tr>
<td></td>
<td>comp_node_recovery</td>
<td>Recovery behavior of compute nodes</td>
</tr>
<tr>
<td></td>
<td>io_node_failure</td>
<td>Failure behavior of I/O nodes</td>
</tr>
<tr>
<td></td>
<td>io_node_recovery</td>
<td>Recovery behavior of I/O nodes</td>
</tr>
<tr>
<td></td>
<td>system_reboot</td>
<td>System reboot operation</td>
</tr>
<tr>
<td>Correlated Failure</td>
<td>correlated_failures</td>
<td>Correlated failure behavior</td>
</tr>
<tr>
<td>Useful Work</td>
<td>useful_work</td>
<td>Useful work computation</td>
</tr>
</tbody>
</table>

![Figure 32: The Overall Composition of the Model](image-url)
**Failure & recovery module.** A compute node or I/O node may fail in any of its states. The occurrence of failures in compute nodes is modeled in the `comp_node_failure` submodel. The recovery is initiated following the detection of the failure, and is modeled in the `comp_node_recovery` submodel. As failures may also occur during recovery, compute nodes may experience multiple failures and subsequent recoveries in the `comp_node_recovery` submodel before the final successful recovery, after which the system resumes the normal execution and checkpointing cycle. Failures of compute nodes do not affect the I/O nodes if error propagation is not considered. The behavior of I/O nodes is similar, except that when an I/O node fails while writing application data to the file system, the application results are lost and the system rolls back to the last checkpoint. This is represented in Figure 32 by an arrow from the `io_node_failure` to the `comp_node_failure` submodels.

The recovery process occurs in two stages. First, the I/O nodes read the checkpoint from the filesystem and buffer it in their local memories. Then the compute nodes read the checkpoint from the I/O nodes and complete the recovery. The compute nodes then go back to the execution state, the master process gets reset and the system exits the correlated failure window if there was one. If the checkpoint is already locally buffered in the I/O nodes when a compute node fails, the first stage is skipped. If an I/O node fails while writing out a checkpoint, the checkpoint is aborted and the I/O nodes get restarted, but the compute nodes are not affected.

If the number of unsuccessful recoveries in the `comp_node_recovery` and/or `io_node_recovery` submodel(s) exceeds a predefined threshold, the whole system, including the compute nodes and I/O nodes, is rebooted in `system_reboot` (“severe failures” transitions from `comp_node_recovery` and `io_node_recovery` to `system_reboot` in Figure 32). When the reboot completes, I/O processors are ready for execution, but compute nodes still need to read the last checkpoint and recover. So
the arrows of “reboot completes” from the system_reboot submodel point to the io_nodes and comp_node_failure submodels, instead of the compute_nodes submodel in Figure 32.

**Correlated failure module.** The correlated_failures submodel models the semantics of correlated failures separately from the compute and I/O nodes’ failure and recovery submodels. It controls the rates of all failures in the system. When a correlated failure occurs, the system enters a correlated failure window, in which it experiences failures with a higher rate than the independent failure rate. Note that independent failures can continue to occur when the system is within a correlated failure window.

**Useful work module.** The useful_work submodel calculates the useful work completed by the system. A positive reward is accumulated when the compute nodes perform job computation or I/O operations, and a negative reward equal to the amount of the lost work is applied when a compute node fails.

**A.5. Modeling Computing and Coordinated Checkpointing**

In this section, we describe the details of modeling the computing and coordinated checkpointing module using SANs. Due to space limitation, detailed SAN models of the other three modules are not described in this appendix. The reader may refer to the report [98] for this.

Figure 33 shows the SAN submodels for the computing and coordinated checkpointing module. States are shared among the submodels with the same names. Selected shared states are numbered in Figure 33 to help identify them.
Figure 33: Submodels for Computing and Checkpointing
When the application is started in the system, the compute nodes start out in the *execution* state and the master is in the *master_sleep* state. We assume the application starts doing computation and the *app_workload* is in the *compute* state. The I/O nodes are in the *ionode_idle* state. Initially, each of these states has a token, indicated by block arrows in Figure 33. In our model the non-random events are modeled as deterministic activities, and exponential distribution is assumed for random events. To simplify the model while still preserving its validity, message transmissions are not explicitly modeled in SAN, but the parameters of the corresponding events are appropriately set to include the message transmission latency. The following steps detail the behavior of the model.

- First, assume that the checkpoint interval expires and the *checkpoint* activity is enabled. The master moves from the *master_sleep* state to the *master_checkpointing* state and starts a timer as shown by the *start_timer* gate (Figure 33d).

- The compute nodes are initially in the state *execution*. When the master moves to *master_checkpointing*, the compute nodes move to the *quiescing* state after a latency of *recv_quiesce_bcast_time* (broadcast overhead) (Figure 33a).

- Henceforth, the behavior depends on whether the application workload is performing computation or I/O. If the *app_workload* is in the *compute* state, the coordination for checkpointing is started, as shown in the *to_coordination* activity. If the *app_workload* is in the *IO* state, the compute nodes wait till the I/O completes before starting the coordination activity (Figure 33c).
• After the coordination activity (coord) completes, a token is placed in the complete_coordination state, enabling the activity coordinate in compute_nodes and the compute nodes move from quiescing to checkpointing (Figure 33e, Figure 33a).

• If the timer expires before the coordination is complete, it places a token in the timedout state. This activates the skip_chkpt2 activity in compute_nodes, causing the compute nodes to abort the checkpointing and move to the back_to_execution state (Figure 33d, Figure 33a and Figure 33e).

• When the compute node is in the state checkpointing and the I/O node is in the state ionode_idle, the dump_chkpt activity is enabled. The checkpoint dump time depends on the checkpoint size and the bandwidth between the compute nodes and the I/O nodes (Figure 33a).

• After storing the checkpoint, the compute nodes go back to the execution state. The completion of this activity also places tokens in the enable_chkpt state (Figure 33a).

• When the I/O node is in ionode_idle, it sees the token in the enable_chkpt state and goes to the writing_chkpt state. This enables the write_chkpt activity, which models the writing of the checkpoint to the file system. The latency of the write depends on the checkpoint size and the bandwidth between the I/O node and the file system (Figure 33b).

• If the I/O node is not in ionode_idle, the compute node has to wait for the I/O node to come to the ionode_idle state before sending the checkpoint to it. This prerequisite is enforced by the ionode_is_idle input gate (Figure 33a).

• When the checkpointing is completed or aborted, tokens are placed in the two states chkpt_completed_or_aborted and to_reset_processor_state. The tokens cause the master to move back to the master_sleep state and the app_workload to reset at the compute state (Figure 33c).
Since the model considers all the compute nodes as a single unit, it does not reflect the discrepancy in the quiesce times among the compute nodes and does not show how the variation in the quiesce time among the nodes can cause the master to timeout. This behavior is modeled separately in the *coordination* submodel (Figure 33e). It is assumed that each node has an identical, exponentially-distributed quiesce time with the mean of MTTQ. We use a random variable $Y$, representing the maximum of all the quiesce times, to model the coordination time as follows.

Let $n$ and $X_i$ denote the number of compute nodes and the $i$th node’s quiesce time, respectively, and $Y = \max\{X_i\} \ (1 \leq i \leq n)$. Then, the CDF of $Y$ is $F_Y(y) = (F_{X_i}(y))^n = (1 - e^{-\lambda y})^n$ where $\lambda$ is the quiesce rate of a single compute node. $Y$ can be generated from a uniform random variable $U$ between 0 and 1 by $Y = \frac{-l}{\lambda} \log(1-U^{1/n})$. The value of $Y$ is used as the latency in the *coord* activity in the *coordination* submodel to represent the coordination process.

### A.6. Modeling Correlated Failures

Two categories of correlated failures are modeled in the appendix: (i) correlated failures due to error propagation, and (ii) generic correlated failures. Both are modeled by appropriately increasing the node/processor failure rates. This section describes how these increased rates are derived.

**Correlated failures due to error propagation.** When an independent failure occurs in the system, with some probability $p$, there is a conditional probability of a second failure due to the first. This results in an increased failure rate. We compute this failure rate increase for all nodes by multiplying the independent failure rate with a constant parameter called $frate\_correlated\_factor$. 
Figure 34: Birth-death Markov process of correlated failures

Figure 34 shows the birth-death Markov process of correlated failures due to error propagation. \( \lambda_i \) and \( \lambda_c \) denote the rates of the system-wide independent failures and successive correlated failures, respectively. \( \lambda \) is the independent failure rate of a single node. \( \mu \) denotes the recovery rate of the system. \( F_i \) is the system state in which \( i \) failures have occurred before a successful recovery. As we assume that any successful recovery wipes off all latent errors, all the \( F_i \) states transit directly to \( F_0 \) with the recovery rate. It is also assumed that the failure rates at all the \( F_i \) states \( (i > 0) \) are the same. So, the conditional probability of another failure occurrence, provided that a failure occurs, is \( p = \frac{\lambda_c}{\lambda_c + \mu} \), and consequently, \( \lambda_c = \frac{p \mu}{(1-p)} \). Let \( n \) denote the number of nodes, and \( r \) denote the multiple \textit{frate}\textunderscore\textit{correlated}\textunderscore\textit{factor}. Then according to the model, \( \lambda_c = \lambda_i + r n \lambda = n \lambda (1+r) \), and hence, \( r = \frac{p \mu}{(1-p)n \lambda} - 1 \). For a given set of \( n, \lambda, \mu, \) and \( r \), i.e. \textit{frate}\textunderscore\textit{correlated}\textunderscore\textit{factor}, actually represents the conditional probability \( p \). As long as \( \lambda_c > \lambda_i \), \( r \) can be chosen independently to study a range of correlated failure effects. For example, when \( n=1024, p=0.3, MTTR=10 \) min, and \( MTTF=25 \) yrs, \( r \) is about 600.

Generic correlated failures. The system may suffer from generic correlated failures at any instant of the system life. A correlated failure coefficient \( \rho \) is assumed to model generic correlated failures, which is the unconditional probability of a correlated failure occurring at any time. Table 20 lists the parameters used for modeling generic correlated failures. Then, the failure rate of generic correlated failures is given by \( \lambda_s = \lambda_{si} + \rho \lambda_{sc} = n \lambda + \rho r n \lambda = n \lambda (1 + \rho r) \). Note that \( \lambda_{si}, \lambda_{sc}, \) and \( \rho \) are not the same as the \( \lambda_i, \lambda_c, \) and \( \rho_c \) in the discussion of correlated failures due to error.
propagation, because they model different probabilities. The symbols $n$, $\lambda$ and $r$ have the same meanings in both models.

**Table 20: Parameters for Modeling Generic Correlated Failures**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_s$</td>
<td>Failure rate of the entire system</td>
</tr>
<tr>
<td>$\lambda_{si}$</td>
<td>rate of independent failures in the system</td>
</tr>
<tr>
<td>$\lambda_{sc}$</td>
<td>rate of correlated failures in the system</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>independent failure per node</td>
</tr>
<tr>
<td>$r$</td>
<td>Increased failure rate due to correlated failures</td>
</tr>
<tr>
<td>$\rho$</td>
<td>correlated failure coefficient</td>
</tr>
<tr>
<td>$n$</td>
<td>number of nodes</td>
</tr>
</tbody>
</table>

**A.7. Experiment Setup**

We use the Mobius modeling environment [99] for creating and simulating the SANs. Steady-state simulation is used with an initial transient period of 1000 hours to allow the system to enter the steady state. The confidence level is 95%. Unless otherwise specified, the parameter values used in the simulation are given in Table 21. The table also explains the rationale for choosing these parameter values based on field data or projections of future systems.
Table 21: Experiment Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value /Range</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint Interval</td>
<td>15 mins to 4 hrs</td>
<td>Derived from other studies [78], and private communication with vendors</td>
</tr>
<tr>
<td>MTTF (Mean Time To Failure per node)</td>
<td>1 – 25 years</td>
<td>Including software and hardware failures recovered from checkpoint. 1 year for ASCI Q and 25 years for IBM mainframes.</td>
</tr>
<tr>
<td>MTTR (system-wide Mean Time To Recovery of compute nodes)</td>
<td>10 minutes</td>
<td>Average time for all compute nodes to read checkpoint and reinitialize themselves.</td>
</tr>
<tr>
<td>MTTR of IO Nodes</td>
<td>1 minute</td>
<td>Time to restart the I/O nodes</td>
</tr>
<tr>
<td>Number of Compute Processors</td>
<td>8K to 256K</td>
<td>Projection of current and future supercomputers</td>
</tr>
<tr>
<td>MTTQ (per-node Mean Time to Quiesce)</td>
<td>0.5-10 s</td>
<td>Time to close I/O and network file handles, clean up states, and perform computation until reaching a safe point</td>
</tr>
<tr>
<td>Broadcast Overhead</td>
<td>1 ms</td>
<td>E.g. data for hardware broadcast trees in Blue-Gene/L [78]</td>
</tr>
<tr>
<td>Software overhead for transmission</td>
<td>1 ms</td>
<td>Measurement of message latency in TCP/IP and UDP</td>
</tr>
<tr>
<td>Period of I/O – compute cycle in application</td>
<td>3 minutes</td>
<td>Experimental data on I/O characteristics of parallel applications [100]</td>
</tr>
<tr>
<td>Fraction of Computation</td>
<td>0.88 – 1.0</td>
<td>Experimental data on I/O characteristics of parallel applications [78]</td>
</tr>
<tr>
<td>Timeout Value</td>
<td>20 secs to 2 min</td>
<td>The period for the master to timeout and cancel the checkpointing.</td>
</tr>
<tr>
<td>Probability of correlated failure</td>
<td>0 to 0.2</td>
<td>Experimental data on correlated failures, e.g. [87]</td>
</tr>
<tr>
<td>Correlated failure Rate</td>
<td>1/MTTF* (100~1600)</td>
<td>Projections on error propagation within a locally-federated cluster of nodes in the supercomputer</td>
</tr>
<tr>
<td>Correlated Failure Window</td>
<td>3 mins</td>
<td>Experimental data for persistence of correlated failures in the system due to error propagation</td>
</tr>
<tr>
<td>System Reboot time</td>
<td>1 hr</td>
<td>Anecdotal evidence for startup time of a large cluster</td>
</tr>
<tr>
<td>Aggregate bandwidth between compute nodes and one I/O node</td>
<td>350 MBps</td>
<td></td>
</tr>
<tr>
<td>Number of compute nodes per I/O node</td>
<td>64</td>
<td>E.g. Blue-Gene/L field data [78]</td>
</tr>
<tr>
<td>file system bandwidth per I/O node</td>
<td>1 Gbps</td>
<td></td>
</tr>
<tr>
<td>Checkpoint-size per node</td>
<td>256 MB</td>
<td></td>
</tr>
<tr>
<td>Average size of I/O data per node</td>
<td>10 MB</td>
<td>Experimental data on typical characteristics of parallel applications [100]</td>
</tr>
</tbody>
</table>
A.8. Experiment Results

Simulations are conducted to study the behavior of the model. As the modeled system is complicated and there are multiple mechanisms/parameters present, we study the system by analyzing the effect of one feature at a time. Hence, the base model without coordination or correlated failures (but with failures during checkpointing and recovery) is first studied to understand the basic system behavior. Then we study the effects of coordination and correlated failures. The following two metrics are used to evaluate system performance.

- **Useful work fraction**: fraction of time the system makes forward progress towards the completion of the job. It does not include work that is repeated due to failures.

- **Total useful work**: the product of useful work fraction and the number of compute processors. It indicates how many processors of the same kind are required to achieve the same performance assuming failure-free computation.

A.8.1. Study of Base Model

For the base model we assume independent failures and consider the coordination time to be a fixed ‘quiesce’ time. The system performance is analyzed for a range of parameters, including the number of processors, checkpoint interval, MTTF per node and MTTR of the system. The values assumed for these parameters are:

- Number of processors per node: 8

- MTTF per node: 1 year

- MTTR of the system: 10 minutes
• Number of processors: 64K

• Checkpoint Interval: varied from 15 minutes to 4 hours

We report results for the number of processors instead of the number of nodes in this study so that they can be easily scaled to different number of processors per node. The major results are as follows:

✧ For a given checkpoint interval (30 mins), MTTR (10 mins) and MTTF (1 yr per node), there is an optimum number of processors (128 K) for which total useful work done by the system is maximized. Adding more processors than this optimum value will hurt the system performance due to failure effects.25 The range of processors considered in this analysis is from 8K to 256K, and the optimum value of the number of processors varies from 128K to 32K as the MTTR varies from 10 minutes to 80 minutes.

✧ For the system to be scalable, checkpoints should be taken on the granularity of minutes (15-30 min), rather than hours as is the current practice. While in theory there is an optimal checkpoint interval, for any practical range there is no ‘optimal’ checkpoint interval for which the useful work is maximized, contrary to what several other studies have shown [80], [81]. This is because the overhead of checkpointing is relatively low in our system as the checkpoint writing is done in the background, and the effect of failures dominates the effect of taking checkpoints frequently.

25 Elnozahy et al. [84] also make a conjecture that increasing the number of nodes beyond a certain extent hurts performance, but do not quantify the extent of the performance loss.
Even when the useful work is maximized, the overall useful work fraction is no more than 50% for an MTTF per node of 1 year. Hence, more than 50% of the system’s resources are spent in checkpointing and recovering from failures.

If the number of processors per node is increased to 32 instead of 8, and the per-node MTTF is maintained the same as 1 year, it is possible to increase the total useful work for the same number of nodes. This is because more compute power is provided per node, while maintaining the same failure rate. The optimum number of processors is in the range of 500K. However, the useful work fraction is unaltered, as the system failure rate, which depends only on the number of nodes and the per-node failure rate, is the same.

**Variation of total useful work with number of processors.** Figure 35a, c, and e show the variation of total useful work with different number of processors. In all three figures, there is an optimum value of the number of processors for which total useful work is maximized. The rationale behind this is as follows: on one hand, more processors provide higher computing power for the job, while on the other hand more processors incur more frequent failures and hence more computation is wasted due to failures. For small numbers of processors the former factor dominates, while for sufficiently large numbers of processors the latter outweighs the former. Consider how the optimum number of processors varies as the MTTF, MTTR and checkpoint interval.

- The optimum value decreases with smaller MTTFs as shown in Figure 35a (from 128K processors for an MTTF of 1 year per node to 64K processors for an MTTF of 0.5 years per node).
Figure 35: Study of Base Model
- The optimum value decreases with larger MTTRs (from 128K processors for an MTTR of 20 minutes to 64K processors for an MTTR of 40 minutes) as shown in Figure 35c.

- The optimum value decreases with larger checkpoint intervals as shown in Figure 35e (from 128K processors for a checkpoint interval of 30 minutes to 64K processors for a checkpoint interval of 60 minutes).

This is because smaller MTTFs increase failure rate, larger MTTRs increase the penalty of a failure and the larger checkpoint intervals cause more work to be lost upon a failure. All the three aggravate the effects of failures, thus lowering the equilibrium point between the computing power and the failure effect.

**Variation of total useful work with checkpoint intervals.** Figure 35b, d and f show the variation of total useful work for different checkpoint intervals. The results indicate that for a large-scale supercomputing system there is no optimum value of the checkpoint interval within the range of values considered (15 minutes to 4 hours). This contradicts previous studies [80], [81] which have shown the existence of an optimum value of the checkpoint interval. This is because the loss of job computation due to failures in large-scale systems outweighs the overhead of frequent checkpointing, as our checkpoint overhead is low. The theoretical optimum value of the checkpointing interval is less than 15 minutes. However, checkpoint intervals less than 15 minutes are not considered because checkpoints as frequent as these may overwhelm the I/O subsystem and the network and hence, are not practical.

Further, the total useful work is approximately constant for checkpoint intervals between 15 to 30 minutes, but decreases sharply as the checkpoint interval is increased beyond 30 mins (for an
MTTF of 8 years, the total useful work only decreases from 43,000 job units to 40,000 job units when the checkpoint interval is increased from 15 minutes to 30 minutes, but drops to 30,000 job units when the checkpoint interval is increased to 60 minutes). This suggests that current checkpoint intervals in the granularity of hours are not appropriate for large-scale systems because of the high system failure rate. The checkpoint intervals should be between 15 and 30 minutes.

**Useful Work Fraction.** The discussion above only uses total useful work as the performance metric. The *useful work fraction* steadily decreases as the number of processors increases. This is because the greater number of processors does not contribute to the useful work fraction, and the failure effect degrades the useful work fraction. So, even when the maximum total useful work is achieved at the optimum number of processors, the useful work fraction is still small. For example, for an MTTF of 1 year per node in Figure 35a, the peak of total useful work is obtained with 128K processors, for which the useful work fraction is only about $56000/131072=42.7\%$, i.e. over 50% of system time is spent in handing failures. Thus, the overall failure rate of the system must substantially decrease for the useful work fraction to improve significantly.

**Effect of Increasing Number of Processors per Node.** So far, we have assumed that each node has 8 processors, and that the MTTF of a node is 1 year. In the future, advances in semiconductor and processor technology may allow 16 or 32 processor cores to be integrated on a single node, while maintaining the same MTTF per node of 1 year. We studied the variation of total useful work with the number of nodes when each node has 32 and 16 processors, respectively, for a per-node MTTF of 1 and 2 years. For fair comparison, the number of processors is fixed at 1000K. The results are shown in Figure 35g and Figure 35h and are summarized as follows:

---

26 One job unit is the amount of work done by a failure-free processor without checkpointing in unit time.
• The optimum number of processors is obtained by multiplying the number of nodes by the number of processors per node. The optimum number of processors is now in the range of 500K to 1000K processors.

• For a given MTTF, the optimum number of nodes increases with the number of processors per node as more compute power is provided at the same failure rate.

• For a given number of processors per node, the optimum number of nodes increases as the MTTF increases because the failure effect is less dominant.

This reinforces the earlier observation that integrating more processors per node and maintaining the same node failure rate increases the total useful work. However, the useful work fraction remains the same (still less than 50%) as it depends only on system failure rate, which in turn depends only on number of nodes and the MTTF per node.

**Effect of failures during checkpointing/recovery.** We also studied the effects of failures during checkpointing/recovery on system performance. We observed that they do not exert as significant an effect as failures during computation on useful work fraction, because the duration of checkpointing/recovery is much smaller than that of computation, and hence, incurs less loss of useful work. Detailed analysis of failures during checkpointing/recovery is not presented.

For the remainder of Section A.8, we assume an increased per-node MTTF of 3 years as otherwise the failure effects dominate the system performance for large numbers of nodes. An MTTF of 3 years corresponds to a per-processor MTTF of 24 years for our system consisting of 8 processors per node, which is close to the 25-year MTTF of IBM mainframes reported in the literature [96].
A.8.2. Effect of Coordination

The coordinated checkpointing protocol requires all the compute processors to arrive at a safe point to take the checkpoint, and a timeout is used to avoid waiting indefinitely. This is not considered in the base model. This section first investigates the pure coordination effect without the timeout mechanism or failures, and then combines them into the study. Three main points are observed from the results:

✧ Coordination does not affect the system performance significantly, as the coordination effect is logarithmic in the number of compute processors (Figure 36), because we assume the processors have identical exponentially-distributed quiesce times. So coordination scales well for practical systems.

✧ Combination of timeout and coordination behaves like a probabilistic checkpoint-abort. Small timeouts (80 s or less in Figure 37) hurt useful work fraction, whereas large timeouts (100s or larger) do not significantly degrade useful work fraction.

✧ As long as the coordination timeout is equal to or larger than a threshold value, the system performance is insensitive to the timeout value. The threshold value is fairly small for practical systems (100 s in our experiment).

*Coordination only.* We assume that all the processors have identical, exponentially-distributed quiesce times with a mean of MTTQ (mean time to quiesce per processor). Figure 36 illustrates the pure coordination effect on useful work fraction for different MTTQs. Failures and timeouts are not considered. According to the figure, the coordination effects are logarithmic in the number of compute processors. This is because an identical exponential distribution is assumed for each
processor. Moreover, the rate of increase of coordination time (or overall quiesce time) is proportional to the MTTQ, and the coordination effect is also proportional to the checkpoint frequency (figures not shown here).

Effects of failures and timeouts. Figure 37 shows the system performance in the presence of failures with an MTTF of 3 years per node, checkpoint interval of 30 minutes, and MTTQ of 10 seconds. We use “no coordination” to indicate the case when no variation in the quiesce times among the compute processors is assumed and the quiesce time of the system as a whole is exponentially distributed with a mean of 10 seconds.

Figure 37 shows that the coordination without a timeout mechanism does not significantly degrade system performance, because the only additional overhead is the small coordination time. If a timeout is applied, the master may time out before the coordination is completed and abort the checkpointing. Then if a failure occurs in the next computation interval, it causes the computation completed in the last interval to be lost. So the combination of coordination and timeout actually
behaves like a probabilistic checkpoint-abort mechanism. The probability depends on the coordination time (MTTQ and number of compute processors) and the timeout. Small timeouts incur large probabilities of checkpoint abortion, and the benefit of limiting the processors’ waiting time is offset by the loss of work due to frequent checkpoint abortions. The drastic curve drops for timeouts of 20-100 seconds in Figure 37 clearly show the performance degradation.

Figure 37 also shows that the system is insensitive to timeouts provided they are large enough because the overall coordination time increases slowly with the number of processors. For example, the 8192-processor system’s performance with a timeout of 100s is only slightly better than a timeout of 120 s and no timeout.

A.8.3. Effect of Correlated Failures

We recall that there are two categories of correlated failures considered in the appendix.

Correlated failures due to error propagation only. Correlated failures due to error propagation are modeled with three parameters: probability of correlated failure \( p_e \), \( \text{frate}_{\text{correlated factor}} \) \( r \) and correlated failure window. As shown in Section A.6, a typical value of \( r \) in real systems is of the order of a few hundred. In our experiments, \( r \) values of 400, 800 and 1600 are used for various \( p_e \) values, with a correlated failure window of 3 minutes.

The results of correlated failures in Figure 38 show that useful work fraction is not susceptible to correlated failures due to error propagation (ranging between 0.51 and 0.56 in the figure), because we assume these failures only occur during recovery, and we observed that failures during recovery do not exert a significant effect on useful work fraction.

Generic correlated failures. Generic correlated failures are modeled with two parameters:
correlated failure factor \( (r) \) and correlated failure coefficient \( (\rho) \). An \( r \) value of 400 and \( \rho \) value of 0.0025 is used in our experiment. So the entire system failure rate gets doubled because of generic correlated failures. The results illustrated in Figure 39 show that, unlike correlated failures due to error propagation, there is a large performance degradation when generic correlated failures are present, and the performance degradation prevents the system from scaling well. For a system comprising 256K processors with a MTTF of 3 years per node, the useful work fraction is reduced by 0.24 (51%).

A.9. Conclusions

This appendix models a large-scale supercomputing system with coordinated checkpointing and rollback recovery. Unlike existing models in the literature, failures during checkpointing/recovery, coordination for checkpointing and correlated failures are included in the model. The impact of these factors on system performance (measured as useful work fraction and total useful work) as well as the scalability of systems with several hundred thousand processors is studied by simulating the model. The major conclusions from this study include:
i. For a given checkpoint interval, MTTR and MTTF, there is an optimum number of processors for which total useful work done by the system is maximized, e.g., for an MTTF per node of 1 year and an MTTR of 10 minutes, it is around 128K.

ii. The overall useful work fraction is relatively low because of the effect of failures in large-scale systems.

iii. Correlated failures must be taken into account as they degrade the performance and limit the system scalability.
APPENDIX B

CHECKPOINTING OF CONTROL STRUCTURES IN MAIN MEMORY DATABASE SYSTEMS

B.1. Introduction

Main memory database (MMDB) systems store data permanently in main memory, and applications can access the data directly [101]. This offers high-speed access to shared data for applications such as real-time billing, high-performance web servers, etc. However, it also makes database systems highly vulnerable to application errors/failures, as the database is directly mapped into the application’s address space.

In addition to user data, the database maintains the control structures (e.g., lock/mutex tables and file tables) necessary for data operation. A database management system (DBMS) maintains data integrity, including recovery in the case of an error/failure of either applications or database services. However, the integrity of control structures is often not well maintained due to the less uniform interfaces to control structures (compared with those to user data). As a result, errors in control structures can become a major cause of system downtime and, hence, an availability bottleneck.

In this appendix, we propose and evaluate an application transparent, low-overhead checkpointing strategy for maintaining the consistency of control structures in a commercial MMDB. The proposed solution is based on the ARMOR architecture and an ARMOR runtime infrastructure.
[102], [103]. It eliminates (or significantly reduces) cases requiring database major recovery, a lengthy process that can take tens of seconds and adversely impact availability. Importantly, the approach can be adapted relatively easily to other applications, and the ARMOR runtime support creates a foundation for providing system-wide error detection and recovery. This work makes the following contributions:

- Introduction of a framework to provide support for checkpointing of MMDB control structures.

- Design and implementation of two checkpointing algorithms. (i) Incremental checkpointing: (a) At runtime, a post-transaction (upon transaction completion) state of the control structure(s) accessed by each write transaction (an update of the control structures) is collected and merged with the current checkpoint. (b) At recovery time, the checkpoint is used directly to restore the correct state. (ii) Delta checkpointing: (a) At runtime, a pretransaction (before any updates occur) state of the control structure(s) accessed by a given transaction (both write or read-only) is preserved as a current checkpoint (delta). (b) At recovery time, the current state of control structures in the shared memory is merged with the delta checkpoint to restore the state.

- Performance evaluation of the proposed checkpointing algorithms. The data show that for a rather harsh workload of 60% write transactions, the performance overhead varies in the range of 1% to 10%, depending on the frequency of transactions.

- Database availability estimation under different frequencies of crashes that require major recovery. The data show that under the error rate of one crash per week, a checkpointing-based solution provides about five nines of availability, one nine more than the baseline system.
B.2. Target System Overview

**Target system.** The target system in this study is a commercial relational MMDB intended to support development of high-performance, fault-resilient applications requiring concurrent access to shared data [104], [105]. The process accessing the shared data can be either a client or a database service. A service is a process that performs functions to assist the proper processing of transactions. For example, the cleanup service detects failures of connected clients/services and performs recovery (including launching major recoveries).

In addition to supporting user data, the database supports control structures (SysDB) necessary for correct operation. Figure 40 depicts the example architecture of SysDB containing three tables: (i) the process table, which maintains process ids and mutex lists for each process as well as information on database mapping into the process address space, (ii) the transaction table, which maintains logs and locks for active transactions, and (iii) the file table, which keeps user database files. Each client/service process maps SysDB into its own address space before accessing the database.

**Reliability problem.** The error model we address in this appendix is the inconsistency of control structures due to the abnormal termination (crash) of one of the clients or services. Upon such a crash, the target system denies services to all other user processes and restarts the entire database system; this is a major recovery. It may take tens of seconds, depending on the size of the data files, and can significantly degrade system availability (not acceptable for services provided to critical applications). A major reason for these problems is the way the database handles access to control structures in SysDB. The system employs multiple mutexes to guarantee the mutual

---

27 In the current implementation, we do not detect silent corruption of data, i.e., incorrect data being written to the database.
exclusion semantic in accessing control structures by user processes. When a client or service crashes while still holding a mutex, the database may remain in an inconsistent state. Since there is no way for the system to identify which updates the crashed client has made, the cleanup process restarts the database to bring the system back into a consistent state. Because major recovery imposes significant system downtime, an approach is needed to eliminate or reduce cases in which it is needed.²⁸

**Figure 40: Example Control Structures (SysDB)**

### B.3. ARMOR High-Availability Infrastructure

The ARMOR infrastructure is designed to manage redundant resources across interconnected nodes, detect errors in both the user applications and the infrastructure components, and recover quickly from failures when they occur. ARMORs (Adaptive Reconfigurable Mobile Objects of Reliability) are multithreaded processes internally structured around objects called *elements* that contain their own private data and provide elementary functions or services. All ARMORs

²⁸ In this discussion, we consider only preserving the consistent state of control structures. Any inconsistency brought to the user data is handled/recovered by the default database services, such as two-phase commit, checkpointing, and logging. As long as *SysDB* consistency is preserved, the system can operate correctly and recover user data.
contain a basic set of elements that provide a core functionality, including the abilities to (i) implement reliable point-to-point message communication between ARMORs, (ii) respond to “Are-you-alive?” messages from the local daemon, and (iii) capture ARMOR state.

ARMORs communicate solely through message-passing. The ARMOR microkernel (present in each ARMOR process) is in charge of distributing messages between elements within the ARMOR and between the ARMORs present in the system. A message consists of sequential operations that trigger element actions. This modular, event-driven architecture permits the ARMOR’s functionality and fault tolerance services to be customized by choosing the particular set of elements that make up the ARMOR. Several ARMOR processes constitute the runtime environment, and each ARMOR plays a specific role in the detection and recovery hierarchy offered to the system and the application. The Fault Tolerance Manager (FTM), Heart Beat ARMOR (HB), and Daemons are fundamental components of an ARMOR-based infrastructure. For more details on ARMOR architecture the reader is referred to [102], [106], [103].

B.4. ARMOR-Based Checkpointing

Embedded ARMORs. In most of cases, an ARMOR launches the application and monitors its behavior. The application is treated as a black box, and only limited services can be provided by the ARMOR infrastructure, e.g., restart of the application process. In the embedded ARMOR solution, an application links the core structure of the ARMOR architecture (the ARMOR microkernel) and uses the ARMOR API to invoke/interface with the underlying element structure of an embedded ARMOR. The embedded ARMOR process appears (i) as a full-fledged ARMOR to other ARMORs and (ii) as a native application process to non-ARMOR processes (e.g., database clients). As a result, the application can take advantage of all services provided by
ARMORs (e.g., adding or removing elements to customize ARMOR functionality) without having to change the original application’s organization. In this way, the application does not need to be rewritten, and only lightweight instrumentation with a few ARMOR APIs is needed to embed the ARMOR-stub into the application.

**ARMOR-based checkpointing.** In order to expose ARMOR services, the database is instrumented in two ways: (i) ARMOR stubs are embedded in the database processes, facilitating communication channel(s) between the database server and the ARMOR infrastructure. (ii) Functionalities are embedded for checkpointing *SysDB* data structures; this modifies selected library functions of the database but preserves function interfaces and, hence, is transparent to clients.

Figure 41 illustrates the basic architecture of the ARMOR infrastructure integrated with the target database. The FTM, FTM daemon, HB, and Daemon constitute the skeleton of the ARMOR infrastructure. The solid lines are the ARMOR communication channels. An ARMOR element called the *image keeper* is embedded into the Daemon ARMOR to maintain the image (checkpoint) of the data structures in *SysDB*. When a client or service process opens the database, the database kernel library creates an *Embedded ARMOR (EA)* stub within the process and establishes the communication channel between the EA and the Daemon. From then on, the checkpoint data can be transmitted directly from the source process (with its EA stub) to the destination ARMOR, which maintains the image in memory. The image is then stored on disk by ARMOR’s checkpoint mechanism.
Arrows in Figure 41 depict the data flow during system operation. Each client or service, when it acquires a mutex (or releases a mutex, depending on the checkpointing strategy applied), sends the related checkpoint through the ARMOR communication channel to the image keeper. The image keeper processes the message according to the checkpointing strategy. If there is no error, the checkpoint reflects the latest consistent state of the SysDB. When a client/service crashes while holding a mutex, the cleanup service requests from the image keeper the saved correct copy of the relevant data structure(s). On the successful restoration of the data, the cleanup service allows the system to continue normal execution without invoking a major recovery.

B.5. Checkpointing Algorithms

This section discusses two algorithms for checkpointing control structures of the target database system: incremental checkpointing and delta checkpointing. We begin with a brief description, summarized in Table 22, of similarities and differences between the two proposed alternatives.
Table 22: Comparison of Incremental and Delta Checkpointing Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Incremental Checkpointing</th>
<th>Delta Checkpointing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Similarities</td>
<td>Close checkpointing architecture, small overhead, no checkpoint taken for read-only access, same way of handling mutex overlap and access to external devices while holding a mutex.</td>
<td></td>
</tr>
<tr>
<td>Differences</td>
<td>1. At startup – an image of all control structures is stored as an initial checkpoint;</td>
<td>1. At startup – an initial checkpoint is an empty data set, i.e., no need to store any control structures;</td>
</tr>
<tr>
<td></td>
<td>2. At runtime – a post-transaction (upon transaction completion) state of the control structure(s) accessed by each write transaction is collected and merged with the current checkpoint;</td>
<td>2. At runtime – a pre-transaction (before data updates occur) state of control structure(s) accessed by a given transaction (write or read-only) is preserved as a current (delta) checkpoint;</td>
</tr>
<tr>
<td></td>
<td>3. At recovery time – the checkpointed image of control structures is directly loaded to the shared memory from the image keeper;</td>
<td>3. At recovery time – the image of control structures from the shared memory is merged with the latest delta stored in the image keeper;</td>
</tr>
<tr>
<td></td>
<td>4. Must checkpoint data updates due to operations within and outside mutex blocks.</td>
<td>4. Must checkpoint only updates due to operations within mutex blocks.</td>
</tr>
</tbody>
</table>

B.5.1. Incremental Checkpointing

In the incremental checkpointing scheme, only updates (incremental changes) to data are sent to the image keeper. The basic algorithm is as follows:

- **After system initialization, the database server sends the image of all the control structures to the image keeper.**

- **In the following processing, a client/service acquires a mutex and then performs operations on the control structures. On each write operation, any changes to the data are stored in the local buffer.**

- **After all updates are successfully finished, the mutex is released, and the client/service delivers the buffered increments to the image keeper for maintaining the up-to-date checkpoint of the control structures.**

- **Upon a crash while the mutex is held, the cleanup service requests from the image keeper the latest checkpoint data and restores the corrupted control structures.**

**Handling mutex overlaps.** In some cases, a single section of control structures is protected by
multiple mutexes. To properly handle this scenario, the image keeper maintains, in addition to the checkpoint, the mapping between a mutex and the data section(s) protected by this mutex. To assist in the mapping, the checkpoint increments sent to the image keeper piggyback the mutex id and the information necessary to identify the correct sections in the control structures.

Figure 42 depicts an example configuration of control structure images kept in the image keeper and illustrates the mapping of mutexes to control structures. With this mapping, overlapped data sections can be protected and their consistency with the corresponding copies in $SysDB$ can be preserved, even when multiple mutexes are acquired at the same time.

**Handling data access without mutex protection.** The proposed algorithm works correctly as long as all updates to control structures are performed within the mutex blocks. There are, however, cases in which control structures are updated directly, without mutex protection, e.g. during database initialization (when it is assumed that no processes try to access the database). While this example is a rather benign case, practice shows that application developers often make somewhat arbitrary decisions and allow the accessing of control structures without mutex protection.\(^{29}\) Handling such scenarios would require (i) locating, in the application code, all the places of potential updates outside mutex blocks and (ii) augmenting the implementation to ensure the checkpoint in the image keeper is up-to-date. This can be difficult given the size and complexity of real-world applications, such as our target database system. Delta checkpointing, discussed next, is an attempt to alleviate this problem.

---

\(^{29}\) Identification of all cases of updates outside mutex blocks would require reviewing/profiling the entire code base. We could not do this due to limited access to the database.
Figure 42: Control Structure Images

B.5.2. Delta Checkpointing

The delta checkpointing algorithm is based on the following assumption: In a correctly implemented system, any access to control structures outside the mutex blocks, after system initialization, does not violate data consistency. Consequently, crashes outside the mutex blocks do not cause data inconsistency, and sections of control structures not updated by any currently executing mutex block are always consistent.\(^\text{30}\) As a result, the image keeper does not need to maintain a copy of all control structures (as in incremental checkpointing, discussed in the previous section). It is sufficient to preserve data sections modified (plus information on the type and parameters of the update operation) while executing a given mutex block. In other words, the algorithm only needs to recognize, collect, and send to the image keeper the modified data section, \emph{delta} (\emph{delta} is a before-image). Upon a failure of a client/service while executing a mutex block, the primary copy of the control structures still exists in the shared memory. The entire image of the related structures (\emph{base}) can then be delivered to the image keeper. Using delta and base, the image keeper computes the original (at the time of entering the mutex block) control structures.

\(^{30}\) Under this assumption, incremental checkpointing (Section 5.1) still needs to determine all locations in the code where \emph{SysDB} is updated.
image \((\text{orig} = \text{base} + \text{delta})\) and sends it back to the cleanup process for recovering the data. The possible updates include data insertion, deletion, and replacement. In summary, the algorithm includes two basic steps:

(1) When a client/service acquires a mutex, it sends the delta to the image keeper. The delta's content depends on the update to be performed in the mutex block.

(2) Upon a crash while the mutex is held, the cleanup service sends the base to the image keeper. The image keeper merges the base with the saved delta and generates the valid image of control structures. The cleanup service requests from the image keeper the regenerated image and restores the corrupted control structures.

Observe that the image keeper merges the base with the latest delta it receives. To avoid using the wrong delta in the case of recovery, it is important to send out a delta at each mutex acquisition, even if the delta is empty, i.e., no changes to the control structures were performed during the current mutex block.

**B.5.3. Image Keeper**

The image keeper is a separate element within the ARMOR process that collects and maintains checkpoint data representing the correct state of control structures. It is a passive component, which means that it is only invoked by the incoming messages (checkpoint updates) and that it performs proper actions according to the received messages. Figure 43 illustrates the basic structure of the image keeper, which consists of (i) a set of memory blocks for preserving images of control data structures (control structure images) and (ii) management support (manager) for updates of the checkpoint and recovery actions in response to client failures. The image keeper
communicates with the database processes by means of the ARMOR communication channel.

The control structure images in Figure 43 represent a memory pool that stores the images of control structures in SysDB. Different mutexes map their corresponding data sections into the copy of control structures in the image keeper (Figure 42).

![Figure 43: Structure of the Image Keeper](image)

**B.6. Performance Evaluation**

This section presents performance measurements of the prototype implementation of the ARMOR-based incremental checkpointing scheme applied to the target database system. The testbed consists of a Sun Blade 100 workstation running the Solaris 8 operating system on top of a 500 MHz UltraSPARC-II CPU with 128 MB of memory. The measurements are conducted in (i) error-free scenarios, in which normal operation of the database under a synthetic workload mimics actual database activity, and (ii) error-recovery scenarios, in which the database recovers from the checkpoint after a failure while executing transactions issued by synthetic clients. While

---

31 Due to the limited time for accessing the target system, we provide measurements only for incremental checkpointing. Because checkpoint data transmission time dominates the performance of both schemes in error-free scenarios and both algorithms transmit a similar amount of data, it is expected that the performance of the two schemes is similar.
checkpointing is applied in the context of the *file table mutex*, the proposed solution applies to other mutexes as well.

**B.6.1. Performance of ARMOR-based Incremental Checkpointing in Error-Free Scenarios**

**Workload.** Each workload invocation involves execution of a sequence of transactions, which arrive with a predefined frequency, and each transaction is represented as a set of operations of variable execution time. Some of the operations need to acquire the file table mutex (*mutex*) to preserve mutual exclusion in accessing shared data by multiple clients.

Operations associated with mutex acquisition can be either data write (*write*) or data read (*read*). The operation pattern within a transaction is a sequence of alternate reads and writes, e.g., *read – write – read – write – …. The sleep function is used to emulate the execution time of operations that (i) do not require mutex acquisition (mutex-free operations), (ii) occur when the mutex is held (mutex operations), or (iii) represent idle time, i.e., the period after completion of a current transaction and before arrival of the next.

**Parameters.** The workload is flexible and can be configured to mimic actual execution scenarios. The tunable workload parameters (and other experiment settings) are as follows:

- *Transaction frequency (freq)* – number of transactions arriving within one second.
- *Number of mutex acquisitions per transaction (num_acq)*.
- *Percentage of read-only operations (read_per)* – fraction of mutex acquisitions for read-only operations.
- *Mutex operation time (mutex_op)* – processing time while holding the mutex, i.e., the interval
between the time the mutex is granted and the time it is released.

- **Mutex-free operation time (other_op)** – processing time for mutex-free operations within the transaction.

- **Delivered data** – the amount of checkpointed data. In the case of the file table mutex, the typical data to be checkpointed is a single file entry in the file table (approx. 3700 bytes). In all measurements, the size of the checkpointed data is assumed to be 4000 bytes.

- **Experiment duration** – time duration of the experiment. The transaction frequency should satisfy the following requirement for experiments to run correctly:

  \[ \frac{1}{freq} \geq \text{mutex}_\text{op}*\text{num}_\text{acq} + \text{other}_\text{op} + \text{time}\{\text{get/release mutexes+checkpointing}\} \]

**Results.** The workload configuration parameters for performance measurements are as follows: \( \text{num}_\text{acq}=5, \text{mutex}_\text{op}=0.002 \text{ s}, \text{other}_\text{op}=0.03 \text{ s}, \text{delivered data}=4000 \text{ bytes}, \text{experiment duration}=20 \text{ s}. \)

Table 23 shows the time per transaction (with 95% confidence intervals) for four transaction frequencies and with read-only percentages (\text{read}_\text{per}) ranging from 0\% to 100\%. The transaction time includes mutex-free operations, mutex operations, mutex acquisition/release, and checkpointing time. Table 24 depicts the performance overhead of checkpointing per transaction. The case of \text{read}_\text{per}=100\% is the one without checkpointing, and hence, it is the baseline against which the overheads in other scenarios are computed and compared.
Table 23: Transaction Time [s]

<table>
<thead>
<tr>
<th>freq (1/s)</th>
<th>read_per (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0.75</td>
<td>0.109 ±0.001</td>
</tr>
<tr>
<td>1.25</td>
<td>0.115 ±0.0095</td>
</tr>
<tr>
<td>1.75</td>
<td>0.124 ±0.0122</td>
</tr>
<tr>
<td>2.25</td>
<td>0.130 ±0.0116</td>
</tr>
</tbody>
</table>

Table 24: Performance Overhead of Checkpointing per Transaction [s]

<table>
<thead>
<tr>
<th>freq (1/s)</th>
<th>read_per (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0.75</td>
<td>0.008 (7.9%)</td>
</tr>
<tr>
<td>1.25</td>
<td>0.013 (12.7%)</td>
</tr>
<tr>
<td>1.75</td>
<td>0.020 (19.2%)</td>
</tr>
<tr>
<td>2.25</td>
<td>0.028 (27.4%)</td>
</tr>
</tbody>
</table>

From the results, one can see that as transaction arrival rate increases, the performance overhead and, hence, the transaction time increases. This is because when there are more requests, ARMORs take more time to process each individual request. However, the frequency increase does not significantly degrade the performance; the overhead ranges from 1 ms to 28 ms for 80% and 0% (all transactions are write) read operations, respectively. If more than half of the mutex acquisitions are read-only, the performance overhead is very small. Note that the variation of the run time without checkpointing (last column in Table 23) can dominate the performance overhead (columns 5 and 6 in Table 23). In real applications, more than 50% of mutex acquisitions are for
read-only operations; the measurement data indicate that under such workloads the overhead due to checkpointing is negligible.

**B.6.2 Performance of ARMOR-Based Incremental Checkpointing in Error-Recovery Scenarios**

The database used in the test consists of five *db files*. Each file contains 100 tables, and each table contains two thousand 200-byte records. So the total size of the user database is 200 MB, a typical size for the database the target system processes in practice. Three clients are used in the error recovery scenario: (i) *testsc*, which updates the table records one after another without acquiring a file table mutex, (ii) *testsc_fimutex*, which repeatedly acquires a file table mutex, updates the table records, and releases the mutex, and (iii) *fimutextest*, which gets a file table mutex and emulates a crash while still holding the mutex. Table 25 presents measurements comparing the performances of both major recovery and ARMOR-based incremental checkpoint recovery. The time listed in Table 25 represents the recovery time, i.e., the time from the crash of the failed client (*fimutextest*) to the first successful acquisition of a file table mutex by the waiting client (*testsc_fimutex*).

**Table 25: Performance of Major Recovery and ARMOR-based Incremental Checkpointing**

<table>
<thead>
<tr>
<th>Trial</th>
<th>Major Recovery [s]</th>
<th>ARMOR-based Checkpointing [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expr 1 (testsc_fimutex + fimutextest)</td>
<td>Expr 2 (testsc_fimutex + fimutextest)</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>29</td>
</tr>
</tbody>
</table>

Four experiments (each is performed for three trials) are conducted with different clients and
recovery policies. The results reported in Table 25 indicate:

- Major recovery can cause significant system downtime (11 to 31 seconds in our experiments). The downtime depends on how much data is loaded into memory when major recovery occurs. (*Testsc_ftmutex* updates a small fraction of table data, so recovery time is small. When *testsc+testsc_ftmutex* is used, since *testsc* updates a whole table, the loaded data is much larger, and recovery time is greater.)

- ARMOR-based incremental checkpointing eliminates or significantly reduces the downtime due to the client crashes: (i) the crash of a client does not impact other processes as long as they do not acquire the same mutex as the terminated client, (ii) an overhead of 2 to 6 seconds (in our measurements) is encountered by any process that attempts to acquire the same mutex as the terminated client, and (iii) recovery using checkpointing does not depend on the amount of loaded data, as there is no need to reload data.

**Availability.** Availability of the database is estimated based on the data on recovery time, assuming different frequencies of crashes that require major recovery. (We consider database availability using Experiment 2 as an example.) The measured average recovery time for major recovery and checkpointing-based recovery are 28.3 s and 4.7 s, respectively. Table 26 shows the system’s availability for various error frequencies. One can see that under an error rate of one per week, checkpointing-based recovery provides about five nines of availability, which is one nine of improvement compared with the major-recovery-based solution.
B.7. Related Work

A number of checkpoint techniques have been proposed to ensure the durability of MMDBs. In Hagmann’s fuzzy checkpointing [107], the checkpoint is taken while the transaction is in progress. An improved variant of fuzzy checkpointing is proposed in [108] and [109]. Non-fuzzy checkpointing algorithms are introduced in [110], [111] and [112].

Levy and Silberschatz [113] design an incremental checkpointing scheme that decouples transaction processing and checkpointing. The propagator component observes the log at all times and propagates the updates of the primary copy in memory to the backup copy on disk. While these traditional techniques rely on control structures to checkpoint user data, we address checkpointing of control structures themselves.

Sullivan and Stonebraker [114] investigate the use of hardware memory protection to prevent erroneous (due to addressing errors) writes to the data structures. In [115], Bohannon et al. achieve such protection by computing a codeword over a region in the data structures. Upon a write, the data region and the associated codeword are updated. A wild write results in an incorrect codeword, which triggers recovery of the corrupted data region. These schemes protect the critical control structures against erroneous writes. Our checkpointing algorithms defend against client crashes and data inconsistency, which is a different failure model. Another technique that addresses this type of failure is process duplication. For example, Tandem’s
process-pair mechanism [116] provides a spare process for the primary one. The primary executes transactions and sends checkpoint messages to the spare. If the primary fails, the spare reconstructs the consistent state from the checkpoint messages. The idea of lightweight, recoverable virtual memory in the context of providing transactional guarantees to applications is explored in [117]. A Rio Vista system for building high-performance recoverable memory for transactional systems is proposed in [118].

B.8. Conclusions

This appendix presents ARMOR-based, transparent, and performance-efficient recovery of control structures in a commercial MMDB. The proposed generic solution allows eliminating or significantly reducing cases requiring major recovery and, hence, significantly improves availability. The solution can be easily adapted to provide system-wide detection and recovery. Performance measurements and availability estimates show that the proposed ARMOR-based checkpointing scheme enhances database availability while keeping performance overhead quite small (less than 2% in a typical workload of real applications).
REFERENCES


