Efficient High Frequency Checkpointing for Recovery and Debugging

Ph.D. Thesis

Dirk Vogt

Vrije Universiteit Amsterdam, 2019

This research was funded by the European Research Council under the ERC Advanced Grant 227874.

ISBN 978-94-028-1388-3

Printed by Ipskamp Printing BV

VRIJE UNIVERSITEIT

Efficient High Frequency Checkpointing for Recovery and Debugging

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op vrijdag 15 maart 2019 om 9.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Dirk Vogt

geboren te Leisnig, Duitsland

promotor: prof.dr. A.S. Tanenbaum
copromotor: prof.dr.ir. H.J. Bos

"Hofstadter's Law: It always takes longer than you expect,
...even when you take into account Hofstadter's Law."

Douglas Richard Hofstadter

ACKNOWLEDGEMENTS

This thesis marks the end of a remarkable period of my life. A period that, to be honest, lasted longer than I and maybe some other people expected. The fact that my defense is scheduled on the last possible date on which Andy can be my promotor speaks volumes. However, it was a period of learning and growth to which a lot of people contributed in one way or another. I am grateful to every single one of them.

I first would like to express my gratitude towards my advisors Andy Tanenbaum and Herbert Bos. Andy officially retired five years ago. That never diminished the loyalty and support he showed towards his students. One time, after thanking him for his comments on a paper I received from him in the middle of the night, Andy replied with one of his typically brief mails: "My students always come first." I am grateful to be one of them. Herbert's practical advise provided invaluable guidance during my studies. His ability to amplify good ideas, while leaving the bad ones behind created an environment that allowed me to think out loud and helped defining the direction of this thesis. Further, his unbreakable patience and optimism helped me through some darker times of my PhD journey. I cannot express enough how lucky am to have had not only one, but two advisors of such caliber.

Next, I want to thank the members of the reading committee, Alexandru Iossup (VU Amsterdam), Christof Fetzer (TU Dresden), Erik van der Kouwe (Leiden University), Paulo Esteves-Veríssimo (Université du Luxembourg) and Robert Watson (University of Cambridge) for reviewing this thesis and being so accommodating while scheduling the date of the defense.

Further, I would like to express my gratidute towards Cristiano Giuffrida, who has been a mentor, friend and collaborator throughout all my PhD studies. His work laid out the foundation of the work presented in this thesis and his advice and collaboration was invaluable while conducting my research. Thanks also go to Georgios Portokalidis for his collaboration on the speculative checkpointing paper. Thank you for the hospitality during our short

visit to New York and the great input you gave during our weekly status calls. Kaveh Razavi has been a great help when proof reading parts of this thesis. Thank you. I am especially grateful to Ben Gras, not only for being a beacon of wittiness and laughter but also for translating the summary of this thesis into the Dutch language.

I also had the pleasure to advise and co-supervise some brilliant master students while being at the VU Amsterdam: Niek Linnenbank, Sharan Santhanam, Koustubha Bhat and Armando Miraglia. Armando's work is covered in this thesis and I am happy to have him as one of my paranymphs.

And thank you, Caroline Waij, for always being helpful with admistrative tasks.

The VU Amsterdam is fantastic place to meet awesome people. There are of course my P4.69 office mates Cristiano Guiffrida, David van Moolenbroek, Erik van der Kouwe, Raja Appusamy and Thomas Hruby: Thank you for the pleasant work environment and all the interesting discussions. I saw you leaving the nest one after the other and now that it's my turn I am happy to have one of you on my thesis committee. Then there is the rest of the Minix team: Arun Tomas, Ben Gras, Lionel Sambuc, Philip Homburg, Thomas Veerman, Kees Jongeburger and Gianluca Guida. I also want to thank Ana-Maria Oprescu for adopting me when I arrived, Claudio Martella for the fun coffee breaks, and David Moolenbroek and Philip Homburg for their friendship and Dutch lessons. You all made the VU a very special place for me.

While the Django Building not only opened a door to affordable housing (not a given in Amsterdam) it also opened a door to countless evenings with great conversations, food and friendship! Thank you Laura, Cristiano, Raja, Stefano, Kaveh, Andrei, Andre, Jailan and Ismail. And thank you, Christian, for representing Django at my defense as my paranymph.

I wouldn't be complete without the most important people in my life, my family, which—as a great side-effect of me taking such a long time to finish—expanded since I started my PhD. But, let me first thank those that were always there: My parents. Their unconditional love and support, not only during my PhD, gave me confidence, strength and I am grateful for every single day with them.

At last, I want to thank Albana, my love. Thank you for giving me the greatest gift. You supported me more than you can imagine and I don't know how I can make up for all the sacrifices you made during this time.

Dirk Vogt
Amsterdam, The Netherlands, February 2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PUBLICATIONS

This dissertation includes a number of research papers as appeared in the following conference proceedings [1] :

Vogt, D., Giuffrida, C., Bos, H., and Tanenbaum, A. S. Lightweight memory checkpointing. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2015), pp. 474–484 [2].

Vogt, D., Miraglia, A., Portokalidis, G., Bos, H., Tanenbaum, A., and Giuffrida, C. Speculative memory checkpointing. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware '15, ACM, pp. 197–209[3].

Miraglia, A., Vogt, D., Bos, H., Tanenbaum, A., and Giuffrida, C. Peeking into the past: Efficient checkpoint-assisted time-traveling debugging. In *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE)* (Oct 2016), pp. 455–466[4].

The following publications have not been included in the thesis:

Bhat, K., Vogt, D., Kouwe, E., Gras, B., Sambuc, L., Tanenbaum, A., Bos, H., and Giuffrida, C. *OSIRIS: Efficient and consistent recovery of compartmentalized operating systems*. Institute of Electrical and Electronics Engineers, Inc., 9 2016, pp. 25–36.

Vogt, D., Giuffrida, C., Bos, H., and Tanenbaum, A. S. Techniques for

[1] The text differs from the original version in minor editorial changes made to improve readability.

[2] Appears in Chapter 2.

[3] Appears in Chapter 3.

[4] Appears in Chapter 4.

efficient in-memory checkpointing. In *Proceedings of the Ninth Workshop on Hot Topics in System Dependability* (2013).

Hruby, T., Vogt, D., Bos, H., and Tanenbaum, A. S. Keep net working - on a dependable and fast networking stack. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2012), DSN '12, IEEE Computer Society, pp. 1–12.

Vogt, D., Döbel, B., and Lackorzynski, A. Stay strong, stay safe: Enhancing reliability of a secure operating system. In *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS 2010), Paris, France, April 2010* (New York, NY, USA, 2010), ACM.

CHAPTER 1

INTRODUCTION

Over the past decades, we have become more and more dependent on information technology. Computers and their software play an ever increasing role in our daily life—software runs on our computers, phones, TVs and around our wrists in the form of smart watches. But also out of our sight, software has taken over a major role in our way of life as it controls the power plants, flies airplanes, schedules trains and drives our car. Software has become the linchpin of our infrastructure.

While one can argue that software has improved our quality of life in many ways, it also is plagued by a problem, which is as old as software itself: Reliable software is hard to build. "Have you tried turning it off and on again", has become our pop-culture's iconic manifestation of this problem.

Nearly everybody had to face the consequences of a computer crash, and in many cases the cause was failing software. Sometimes this meant some lost work, a dropped phone call or having to go to turn the router "off and on again". However, as software plays a more critical role in our lives the consequences of software failures have become potentially disastrous and include the loss of large sums of money (as nearly all trading happens in software nowadays), structural damages (for instance in case of failure of software controlling power plants) to lost lives (in case of failure of software in critical systems). No wonder a lot of resources are invested in dealing with the

effects of failing software, for example using redundant computer systems, increasing the resources needed achieve a certain goal.

The users of software systems are not the only ones to suffer the effects of unreliable software. They also affect the creators of the software, who are often the source of the problem. A large part of software developers' time accounts to dealing with the consequences of their mistakes. Part of this time can be accounted for 'just' fixing bugs. However, a considerable amount of time is also spent on reliably reproducing a problem to eventually understand and then fix it.

Consequentially, it does not come as a surprise that the research of methodologies and technologies to tackle this problem originated at the same time in which digital computers first hit the industry and universities.

Research on software reliability can be divided in two camps: One camp investigates ways to *create reliable software*, to lower the risk of software defects appearing, while another line of research takes it as a given that software contains bugs and tries to find ways to *deal with the consequences* of these bugs.

Techniques developed by the former camp have helped to increase the quality of software. Improvements in static and dynamic analysis [124][29] and automated testing and test generation[34] can have a big impact on the number of bugs introduced during the creation of the software. Static analysis can detect common errors in computer programs and enforce best practices during the development of software. Automated test generation in turn helps to test portions of the code that may only run rarely or for which writing tests is cumbersome. Further, safe programming languages such as Java make engineering reliable software easier, as they take control of certain complex tasks such as memory management [101] and thread safety [92] away from the software developers. Finally, formal methods can verify the correctness of software [73]. But, formal verification is expensive and hard to apply for existing large code bases.

Despite the efforts to create bug-free software, in practice it seems unavoidable for bugs to end up in the released software: A study [107] from 2002 found the industry standard defect density in software to be around 1 to 25 defects per 1 000 lines of code (kLOC) and observed those bugs to remain undetected for a long time after release. Microsoft in turn experienced a defect density of 10 to 20 defects per kLOC during development and 0.5 defects per kLOC for released code [93]. This translates to around 25 000 software defects in the ca. 50 million of lines of code of the Windows operating system [87]. Coverity's open source scan [131] from 2014 in turn reported an average defect density of 0.61 defects per kLOC amongst 2650

open source projects. For proprietary projects the same report notes 0.76 defects per kLOC. If we extrapolate this number, for example, to the Linux kernel which accounts to ca. 20 million lines of code, this means 12 000 defects. This means that defects in released software are a reality we have to live with, at least for the foreseeable future.

The remaining bugs that end up in released software are targeted by the second body of research that dedicates itself to the development of fault-tolerant software or "damage-control," that is, limiting the consequences of a software defect manifesting itself at runtime.

One crucial step toward building a more reliable software stack is fault isolation to limit the consequences of a crash to the failing component of the system [130]. This way, a crashing component does not take down the whole system and after the crash the component might be restarted. This simple *crash recovery* approach is practical for components with little or no state. One can extend this approach by introducing redundancy and run different implementations of the same software components. This n-version programming approach [36] finds use in highly critical systems such as power plants or flight control. Although the implementations of components differ, as the software is usually written by different teams according to a shared specification, they are expected to produce the same output for the same input. This is done to reduce the likelihood of the redundant components failing for the same input. If the output of one of the redundant component differs—or does not appear at all because the component crashed—the results of the remaining components may be used instead based on a conflict resolution scheme (e.g., majority voting). While the operation continues with only a subset of the redundant components, the crashed component can be restarted.

These two approaches of *crash recovery*, however, have still to deal with consequences of losing a crashed component's state. A common way to recover the state is to keep a copy of it somewhere outside of the reach of the component and request it later during the recovery operation. MINIX 3 for example allows a component to request its state to be stored in a separate "data store" component. Further state changes have to be explicitly communicated to this data store. After a component crashed, MINIX 3's reincarnation server will restart the component, which in turn can request the previously saved state from the data store to reinitialize itself into the state before the crash and resume its operation. However, doing so requires the component to be crafted in a way that takes this state saving and restore operation into account. These operations in turn have to take into account the structure of the data

that comprise the state of a component.

This thesis dedicates itself to an alternative approach for capturing the state of applications. This approach, called checkpointing, eliminates the complexity of these explicit store and restore operations.

1.1 Checkpointing

Checkpointing is the process of storing a snapshot or *checkpoint* of a running system. This system can be one or a group of processes, virtual machines, the whole operating system or even a distributed system. Checkpointing plays a crucial role in increasing fault tolerance by automatic error recovering techniques [152; 135; 53; 83; 54; 121; 117; 125; 51; 68; 115], as it allows one to reset the application state to a checkpoint that was created before an error occurred. However, improving fault tolerance is only one of the many applications of checkpointing. Instead of being restored, a checkpoint can be used to 'peek' into the state prior to the crash, exposing valuable information that may help with debugging the problem [129; 128; 61; 72; 48]. Further, checkpointing also finds applications outside of the reliability/dependability domain such as process migration [106], intermittent computing [145], facilitating automatic software updates [54] and kernel hot patching [70].

Note, that there is more to a process' state then just its memory. In most operating systems, for example, beside the process' memory, also all kernel objects associated with the process such as thread contexts, open file descriptors and sockets have to be considered state of the process which may or may not be subject to checkpointing.

Memory checkpointing is an important cost factor, especially for use cases requiring high checkpointing frequencies, which makes it an interesting and worthwhile target for optimization.

1.1.1 The Costs of Memory Checkpointing

The costs for memory checkpointing can be split in three parts:

Deployment Costs

Ideally, a checkpointing solution can be used on an off-the-shelf operating system without the need to alter the kernel. For performance reasons, however, some solutions opt for an implementation in kernel space, which allows

them to benefit from functionalities and data that are usually not accessible to unprivileged user level processes. This advantage however has to be traded off with the increased cost of deployment: The use of an automatically updated binary kernel package, usually part of a system distribution, is not possible anymore and the users of a kernel-based technique have to compile the kernel themselves. This disadvantage can be partially mitigated by implementing the checkpointing functionality in a loadable kernel module. However, the burden of recompiling this module after installing a kernel upgrade is still on the user. User-level checkpointing solutions instead are easier to deploy, but only have a limited set of primitives at hand to implement the actual checkpointing functionality.

Runtime Performance Costs

The process of taking a snapshot of a process' memory slows down its execution. This cost generally increases with the checkpoint frequency. The runtime overhead consists of direct cost, that is the actual cycles required to copy the application state into a checkpoint, and indirect costs, caused by the pollution of caches and handling of page faults caused by the checkpointing mechanism.

A straightforward way to speed up the process of taking a checkpoint is to limit the amount of data that is part of a checkpoint. An application agnostic way to limit the amount of memory that has to be copied for a checkpoint is to copy only *incrementally* the portion of the state that changed in relation to the previous checkpoint [52; 136; 119; 77; 125; 72; 51; 128; 112; 33; 117]. Using this approach, memory regions only have to be copied if they get modified. To detect which memory regions are modified two approaches can be used: 1) The *copy-on-write*–short CoW—approach detects writes to memory regions—for example, by limiting write access to the region—and copies the *old data* before it gets modified. 2) The *dirty-tracking* approach works by tracking during an checkpointing interval which regions got modified, that is, are dirty, and then copying those regions containing the *new data* when a new cycle gets started. Both approaches can be implemented in userland, but they greatly benefit from using primitives or data that are solely accessible to kernel.

Another way to limit the amount of memory to be checkpointed, is to ignore transient state changes. These are modifications to the application's state that are only valid between two checkpoints. By wisely choosing the points on which checkpoints are taken—either with application knowledge

or the use of profiling—this regions can be tuned to include portions of the call-stack and memory regions that get allocated and deallocated during one checkpointing interval. Further, it is not necessary to store application data that can be easily recovered. Examples include data caches or data that can be re-generated by re-reinitializing the application (e.g., configuration data).

An application developer may choose to mark those 'recoverable' memory regions to exclude them from being subject to checkpointing. However, the need for this application-specific knowledge makes this solution harder to deploy.

Storage Costs

Another factor to consider is the cost of storing the checkpoint. This affects both the runtime overhead of checkpointing and the amount of memory needed for storing the checkpoints. For applications that require a short checkpointing interval storing the checkpoints on persistent storage often incurs prohibitive performance costs, which is why in such cases checkpoints are taken *diskless* and are kept *in-memory* [113]. To store as many checkpoints as possible in memory, the checkpoints have to be stored efficiently. Incremental checkpointing and limiting the amount of application state subject to checkpointing can help to decrease the size of checkpoints. Also other techniques like compression of checkpoints can help to decrease the size of the checkpoints. They, however, slow down the process of storing a checkpoint, creating a trade off between runtime overhead, storage efficiency and recovery time.

1.2 High Frequency Checkpointing

Many use cases require checkpoints to be taken with a high frequency. Examples of such use cases are automatic error recovery techniques that require checkpoints on every client request [53; 83] or on carefully selected rescue points [115; 125]. Further, in debugging scenarios the ability take checkpoints with very high frequency allows the inspection of arbitrary memory states throughout the execution.

Current checkpointing solutions fail to offer the necessary performance needed for this high frequency checkpointing applications, or are hard to deploy.

Further, high checkpointing frequencies lead to a large number of checkpoints, which forces the user of the checkpointing solution to trade off memory space between checkpoint storage and the checkpointed application.

1.3 Improving In-Memory Checkpointing

We believe that the current solutions are not suitable for high frequency checkpointing, as they do not offer sufficiently low performance and storage overhead for high checkpointing frequencies and require awkward trade-offs during deployment.

These drawbacks make it necessary to revisit current memory checkpointing techniques and either augment them with new capabilities for enabling high checkpointing frequencies or design entirely new techniques when necessary.

1.3.1 User-level High Frequency Checkpointing

Some situations make it impossible or undesirable to change the underlying operating system kernel. This limits the effectiveness of user-level checkpointing techniques, as some kernel mechanisms, such as efficient CoW, are not directly exported, but only accessible through interfaces such as the `fork` system call or POSIX signals. Those interfaces are not tuned for high frequency checkpointing. Emerging instrumentation based approaches seem to be better suited for pure user-level checkpointing solutions. In this thesis we want to answer the following questions concerning pure user level checkpointing:

Q1.1 What is the suitability of current user-level memory checkpointing techniques?

Q1.2 How do emerging instrumentation based sub-page granular checkpointing techniques compare to traditional page granular checkpointing techniques?

Q1.3 Further, how can we solve the problem of memory unboundedness of these instrumentation based approaches?

1.3.2 Kernel-level High Frequency Checkpointing

If modifications to the kernel are acceptable, memory checkpointing techniques can benefit from the accessibility of data and mechanisms that are not accessible to pure user-level solutions. For instance, modifications to the kernel may allow us to export an efficient CoW mechanism as a first class primitive to userland.

In this thesis, we want to answer the following questions:

Q2.1 What are the key cost factors for page granular memory checkpointing?

Q2.2 How to expose the kernel's efficient CoW implementation to a high frequency checkpointing solution?

Q2.3 Can we speed up high frequency checkpointing speculatively copying the writable working set into the checkpoint thus reducing the overhead of CoW and what are viable algorithms to establish the writable working set in a checkpointing scenario in the first place?

1.3.3 Dealing with Millions of Checkpoints

A high checkpoint frequency leads to a lot of checkpoints, possibly more than tens of thousands checkpoints per second. In this thesis we explore how to deal with such large numbers of checkpoints.

Q3.2 Can high frequency checkpointing and storing millions of checkpoints ever be practical?

Q3.1 If the answer to the previous question is yes, what is the best way to store a large number of checkpoints in memory taking into account the space and performance trade-offs involved?

1.4 Structure of this Thesis

The three main chapters of this thesis consist of published peer reviewed publications. The first two papers focus on the performance and deploya-

bility aspects of high frequency memory checkpointing. The third paper in turn describes a debugging use case of high frequency checkpointing and concentrates on efficient checkpoint storage and exploration.

Chapter 2 This chapter investigates the performance of current checkpointing techniques, focusing on solutions that do not require changes to the operating system kernel. We propose "Lightweight Memory Checkpointing" a byte-granular checkpointing solution that utilizes a compiler-assisted shadow state organization to store the checkpointed state. We compare LMC against three state-of-the-art page granular userland memory checkpointing solutions and show that, while implemented completely in userland and thus offering good deployability. LMC also offers better performance and an upper bound on memory usage unlike most some existing approaches.

Chapter 2 was presented at the *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)* [140]. Cristiano Giuffrida contributed to the related work section of this paper.

Chapter 3 Chapter 3 in turn re-examines the costs of traditional page granular checkpointing and proposes a new set of abstractions for page granular in-memory checkpointing. We implement those first class checkpointing primitives as an easy to deploy kernel module. Motivated by the results of the examination of checkpointing costs, we propose and evaluate a speculation mechanism to further improve the checkpointing overhead by speculatively pre-copying memory pages that are likely to be accessed during the next checkpointing interval.

Chapter 3 was presented at the *16th Annual Middleware Conference (Middleware '15)* [141]. Cristiano Giuffrida contributed to the related work section of this paper, Georgios Portokalidis contributed to the background section and Armando Miraglia helped to prepare the paper for submission.

Chapter 4 Finally, Chapter 4 proposes checkpointing as an alternative to record and replay debugging solutions. It introduces "DeLorean," a debugging system based on gdb, which allows developers to investigate previous states of execution. To do so, DeLorian makes use of the newly implemented checkpointing primitives introduced in the previous chapter. Further, it puts its emphasis on the investigation of ways to efficiently store a large number—up to millions—of checkpoints in memory.

Chapter 4 was presented at the *27th International Symposium on Software Reliability Engineering (ISSRE 2016)* [95]. The idea and design of DeLorean

is mine. Armando Miraglia implemented Delorean as an extension of SMC's kernel extension and ran experiments as part of his master thesis under my daily supervision. He further contributed to the architecture overview and evaluation section of the paper. Cristiano Giuffrida contributed to the related work section of this paper.

Chapter 5 In Chapter 5 we conclude this thesis by summarizing its content and proposing directions for further research.

CHAPTER 2

LIGHTWEIGHT MEMORY CHECKPOINTING

2.1 Introduction

Memory checkpointing has great potential to improve the reliability of today's software stack as it plays a vital role in many important application domains. Unfortunately, prior solutions generally impose awkward tradeoffs of deployability, performance, and memory usage—in the common scenario of high-frequency memory checkpointing. This paper presents a novel memory checkpointing strategy that combines deployability, performance, and memory usage guarantees for such scenarios, facilitating practical deployment of memory checkpointing in real-world systems.

2.1.1 Memory Checkpointing

The last decade has witnessed a growing interest in memory checkpointing, an important technique that allows users to snapshot the memory image of a running program in main memory (as opposed to the disk, which is much less

efficient [113]), and restore (or simply inspect) the checkpointed image later on. This is, for example, a fundamental building block in automatic error recovery techniques, which need to periodically revert the active memory image to a safe and stable state [82; 152; 53; 83; 117; 125; 51; 68; 115]. Memory checkpointing has also been applied to several other application scenarios, including debugging [129; 128; 61; 72], software transactional memory [83], program backtracking [159; 33], fast initialization [144], and memory rejuvenation [144].

To be of practical use, most application scenarios require high *checkpointing frequencies*. For example, automatic error recovery techniques typically checkpoint the active memory image at every client request [53; 83] or at carefully selected rescue points [115; 125], commonly resulting in thousands of checkpoints per second. In debugging applications, frequent checkpoints allow users to efficiently inspect arbitrary memory states throughout the observed execution [72].

Memory checkpointing can be implemented at the user or at the kernel level. Kernel-level solutions are generally more efficient, but also increase the *reliable* computing base [43] of the entire system. For this reason, prior kernel-level solutions [2; 106; 77; 76] have failed to reach adoption in commodity kernels—even when explicitly seeking mainline inclusion [76]. The lack of mainline support forces users to manually patch the kernel, which ultimately results in *poor deployability* guarantees of the solution in practice. Further, kernel-based implementations, in general, are very specific to the targeted kernel and as a consequence offer *bad portability* to other operating systems.

2.1.2 User-level Memory Checkpointing

Most existing solutions rely on page-granular copy-on-write (COW) or dirty page tracking mechanisms. Both can be implemented either by means of kernel mechanisms (e.g., `fork`-based COW [117] and *soft dirty bit*-based dirty page tracking [13]), or in userland using application-level page fault handling [119; 112]. All these techniques, however, ultimately rely on hardware-supported page protection mechanisms to trigger a minor page fault every time the application tries to first modify a particular memory page. At high checkpointing frequencies, the cost for implementing checkpointing operations and handling those minor page faults inevitably translates to *poor performance*, confirming that traditional user-level interfaces are ill-suited for efficient low-level memory management tasks such as memory checkpoint-

ing [28]. To address these limitations, several recent memory checkpointing solutions resort to more fine-grained instrumentation-based strategies, which—as demonstrated by preliminary results in our prior work [139]—are promising to improve memory checkpointing performance especially in high-frequency checkpointing scenarios. Some solutions rely on static analysis [53; 68] to identify all the memory objects to checkpoint, but are forced to make strong assumptions on the system model to avoid unnecessary copying (and thus *poor performance*) induced by the conservativeness of the analysis.

Those solutions record all the memory writes in an undo log generated by static [83; 159] or dynamic [115] instrumentation. Simply recording all writes in a log provides an efficient checkpointing strategy, but also yields very *poor memory usage* guarantees—as the log may grow uncontrollably when programs repeatedly write data into the same memory location. Current remedies to this problem are largely unsatisfactory. These include swapping the log to disk [152]—translating to *poor performance*—hashing to identify duplicate log entries [159]—translating to *poor performance*—or relying on specialized hardware support—translating to *poor deployability*.

In this paper, we present *Lightweight Memory Checkpointing* (*LMC*), a new memory checkpointing technique, which combines the performance guarantees of undolog-based checkpointing with the memory guarantees of traditional page-granular checkpointing. LMC relies on a compiler-assisted *shadow state* organization—similar, in spirit, to shadow memory approaches used in state-of-the-art memory tracing techniques [124; 160; 162; 102; 111; 147; 161; 103]—to incrementally checkpoint the active memory image at the byte granularity. Our prototype implementation of LMC is targeted to Linux, but as it is a pure userland solution it is easily portable to other operating systems.

The contribution of this paper is threefold: (i) we present the design of lightweight memory checkpointing; (ii) we present a prototype implementation of LMC; (iii) we thoroughly evaluate and compare our prototype implementation to existing user-level memory checkpointing solutions and show that a carefully optimized instrumentation design can provide strong memory usage guarantees without sacrificing performance and, in some cases, even significantly improve the performance of prior undolog-based strategies [159].

2.2 Overview

Figure 2.1 presents a high-level overview of LMC. To deploy LMC, users need to link their program against the LMC checkpointing library (that is `liblmc.a`) and instrument it using the LMC transformation/optimization passes (which are `lmc.so` and `lmc-opt.so`) implemented using LLVM [79]. Both are accomplished by instructing the linker (`ld`) to use our instrumentation strategy via build flags. LMC is currently tailored to Linux programs, but our prototype is portable to other Unix systems—Linux-specific extensions will be explicitly mentioned, hereafter.

The LMC checkpointing library exports a simple API to create and restore memory checkpoints from user programs. Internally, it also maintains LMC's shadow state organization and implements all the necessary hooks used by our instrumentation. The LMC transformation pass relies on such hooks to instrument all the memory writes in the program and incrementally maintain memory checkpoints using byte-granular copy-on-write semantics, i.e., copying every byte of memory to the shadow state at the first modification in the current *checkpoint interval*—interval between consecutive memory checkpoint/restore operations. Finally, the LMC optimization pass carefully reoptimizes the transformed code before generating the final binary.

When a user program issues a memory *checkpoint* request to our checkpointing library, LMC prepares a new shadow state for the current checkpoint and instructs our instrumentation to track all the changes to the current program state into it. By the end of the checkpoint interval, the shadow state contains a copy of all the data in the original program state that has been modified by the program since the last checkpoint operation, as well as all the necessary tracking information to locate such modifications. When a user program issues a memory *restore* request to our checkpointing library, this information allows LMC to automatically revert the current memory image to a given memory checkpoint.

This strategy requires maintaining one shadow state for each checkpoint stored in memory. In the following, we assume a single checkpoint maintained in memory at any given time for simplicity—a common assumption in traditional memory checkpointing applications [144; 152; 53; 83; 117; 125; 51; 68; 115]—but LMC can, in principle, retain an arbitrary number of checkpoints during the execution of the program. The latter is only constrained by the amount of the virtual memory address space available—limited on 32-bit programs, but a plentiful resource on modern 64-bit architectures.

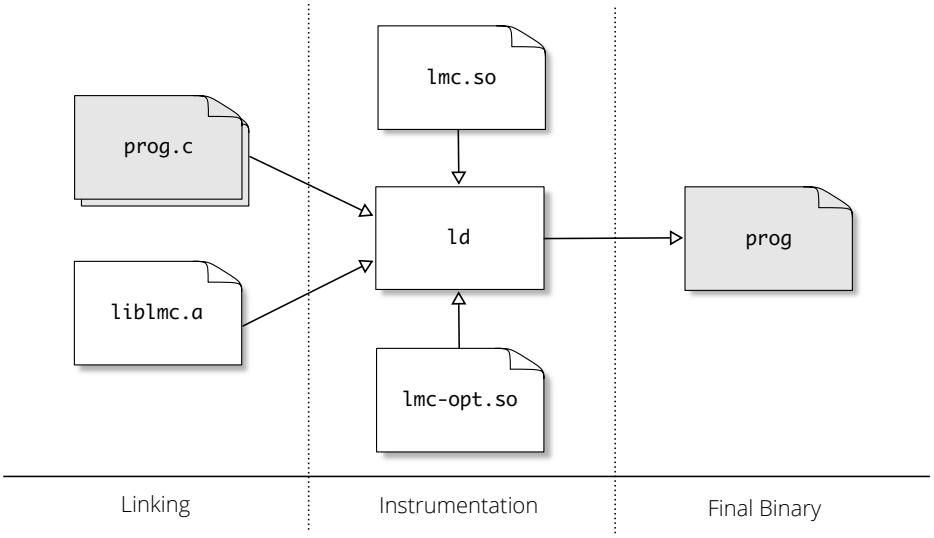


Figure 2.1: High-level overview of LMC.

2.2.1 Memory Write Instrumentation

Our memory write instrumentation tracks all the possible memory altering instructions in the original program, i.e., write instructions and calls to standard memory intrinsics—`memcpy` and `memset`—both referred to as store instructions from now on. Our transformation pass replaces each store instruction with a call to a `store_hook` function provided by `liblmc`, which (i) checks if the soon-to-be-altered memory location has not already been saved in the current checkpoint interval, (ii) copies the original data to the shadow state if necessary, and finally (iii) performs the store instruction as originally intended. Our instrumentation operates entirely at the LLVM IR level, allowing LMC to employ effective optimization strategies on the transformed code. In particular, for optimization purposes, LMC relies both on standard compiler optimizations—e.g., *inlining*, to reduce the costs associated to frequent calls to the `store_hook` function in our library—and on checkpointing-specific optimizations implemented in our optimization pass (Section 2.5).

2.2.2 Shadow State

Our memory checkpointing strategy splits the original program state into a *primary state*—the portion of the memory address space in use by the running program—and a *shadow state*—the portion of the memory address space in use by our instrumentation to incrementally store the data associated to the current checkpoint. The tracking information for the checkpointed data, in turn, is maintained in a separate per-shadow state *tagmap*. Each tag in the tagmap refers to a predetermined memory region, providing information on whether the corresponding data in the primary state has already been saved in the shadow state. The tagmap is entirely maintained in software and fundamental to the internal operations of our `store_hook`.

2.3 Shadow State Organization

Our shadow state organization fulfills three key design goals: (i) *deployability*, that is no changes to commodity operating systems, user programs, or widely deployed security mechanisms such as address space layout randomization; (ii) *portability*, that is support for multiple architectures; (iii) *efficiency*, that is fast shadow state and tagmap management with minimal run-time performance overhead.

2.3.1 Memory Address Space Layout

LMC's shadow state strategy dictates splitting the virtual memory address space of a program into three independent memory areas accommodating the primary state, the shadow state, and the tagmap. To support efficient memory lookups across the different areas, LMC maintains predetermined linear mappings for any given memory address from one area to another. This approach ensures that, given an address in the primary state, LMC can locate the corresponding address in the shadow state and the corresponding tag in the tagmap in constant time—using preassigned offsets. In the simplest shadow state organization possible, this strategy can be enforced by splitting the memory address space into three equally-sized areas, with 1-byte tags in the tagmap each referring to 1 byte in the primary (and shadow) state.

To increase the size of the primary state available to the program and improve the memory locality of the checkpointing activity, however, LMC relies on a more general tagmap implementation, with each tag referring to

a generic primary/shadow memory region of ρ bytes—with ρ selected by empirical measurements by default (see Section 2.4.1). This design choice results in a more general shadow state organization, with the final sizing and positioning of the different memory areas subject to the particular architecture adopted. Figure 2.2 shows the final memory layout adopted by LMC for both 32-bit and 64-bit Linux programs.

32-bit Address Space Layout

On 32-bit architectures, LMC resorts to a compact memory layout—using the `ADDR_COMPAT_LAYOUT` personality on Linux—to locate the entire primary state in the lower half of the memory address space. In particular, this strategy locates the text, data, and heap segments at the very bottom of the memory address space and memory mapped segments—i.e., anonymous mappings, file-backed mappings, and shared library mappings—above, in the first 1 GB of the address space. On Linux—and on typical Unix systems in general—this strategy leaves the stack as the only memory area resident in the upper half of the program's usable address space. To implement its shadow state organization, LMC relocates the stack (Section 2.3.2) during early program initialization to the lower half of the address space, leaving the upper half—starting from 1.5 GB—entirely allocated to the shadow state. The tagmap, finally, is placed at the top of the lower half of the memory address space, with a total size of 12 MB in the default $\rho = 128$ configuration (Section 2.4.1). This strategy yields a program-usable primary state size of less than 1.5 GB, a necessary compromise for any shadow memory organization on the limited 32-bit architecture, which, however, did not prevent LMC from successfully running all the test programs considered in our experimental evaluation.

64-bit Address Space Layout

On 64-bit architectures, LMC's instrumentation generates position-independent executable (PIE) binaries to freely relocate the primary state. Note that, although position-independent code is known to introduce nontrivial performance overhead on particular architectures [110], 64-bit architectures have been designed to efficiently support PIE binaries. As a matter of fact, increasingly many operating system distributions—e.g., Ubuntu—have started to ship 64-bit software packages in PIE format, which also improves the coverage of address space layout randomization and thus software secu-

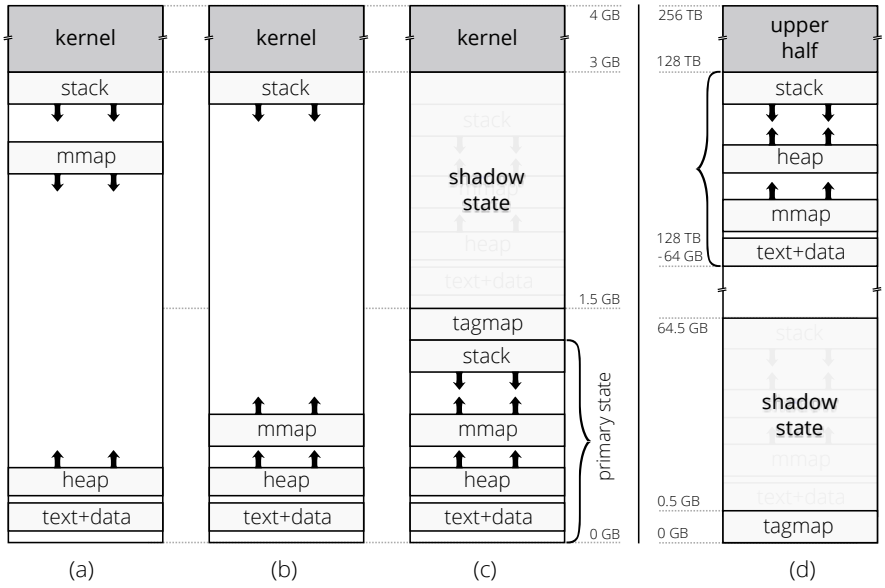


Figure 2.2: The address space layout used by LMC on Linux: (a) Unmodified 32-bit layout. (b) Compact 32-bit layout. (c) Compact 32-bit layout with shadow state organization. (d) 64-bit layout with shadow state organization.

rity. LMC follows the same strategy, which automatically ensures relocation of the entire primary state in the upper 64 GB of the memory address space. The tagmap and the shadow state, in turn, are consecutively allocated at the bottom of the memory address space. Under the default $\rho = 128$ configuration (Section 2.4.1), this strategy yields a 512 MB tagmap. The upper half—starting from 128 TB—finally, is reserved at initialization time and made inaccessible to the program, for simplicity. This strategy still leaves nearly 128 TB of memory address space available to the running program.

2.3.2 Stack Relocation

To relocate the stack on 32-bit architectures, the LMC checkpointing library instruments the program to intercept the application entry point and transparently perform the relocation. On Linux, this is equivalent to overriding `glibc`'s `_libc_start_main` with a library function that allocates a new stack, copies the arguments and environment variables to the new stack loca-

tion, and finally returns control to libc. This strategy, however, is alone insufficient to relocate the stack, given that the operating system is not aware of the change and, for example, can no longer export the program command-line arguments through the *proc* filesystem—i.e., `/proc/pid/cmdline`. To address this problem, LMC relies on kernel support introduced by recent user-level checkpoint-restart solutions [13], which allows a user program owning the necessary capability (i.e., `CAP_SYS_RESOURCE`) to inform the kernel of a new relocated stack via a dedicated interface (i.e., `prctl`).

2.3.3 Tagmap

LMC maintains a tagmap using 1 byte tags and addressing memory regions of a configurable size S_t. To reconfigure the tagmap layout, users can specify a custom S_t (power of two) and recompile the checkpointing library. When called by the instrumented code, our `store_hook` locates the tag associated to the given store instruction by calculating $(a >> log_2(S_t)) + t_{off}$, where t_{off} is the offset between the base address of the primary state and the base address of the tagmap and a is the destination address for the store instruction. If the tag is not set, our `store_hook` first copies the memory region from the primary state into the shadow state and sets the tag in the tagmap before issuing the given store instruction into the primary state. The tagmap is reset to a pristine state at the next checkpoint/restore operation associated to a given shadow state. Store instructions based on memory intrinsics are segmented into multiple memory writes according to the region size selected.

2.4 Tagmap Management

This section details the key issues LMC addresses to manage the tagmap as part of its checkpointing strategy.

2.4.1 Memory Region Size

The memory region size determines the size of the tagmap and also the granularity of the checkpointing strategy, i.e., a smaller region size results in a larger tagmap and a finer level of granularity. Selecting a large region size has two advantages. First, larger regions—and thus smaller tagmaps—yield more

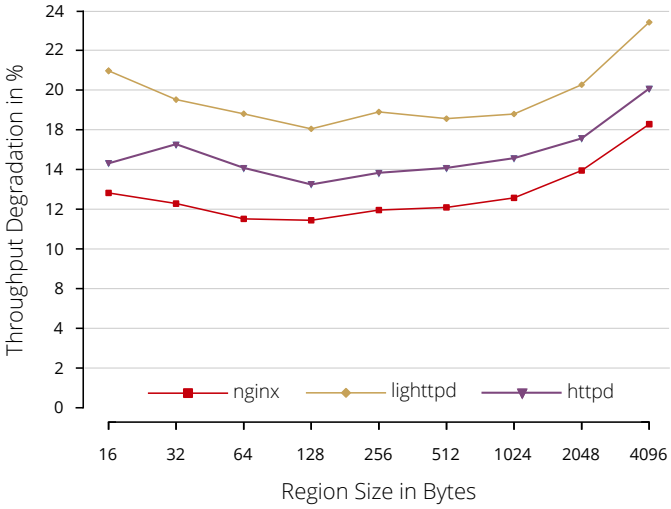


Figure 2.3: Throughput degradation on lighttpd, nginx and Apache httpd for different memory region sizes.

program-usable address space. Second, larger regions increase the probability of two given store instructions pointing into the same region. This leads to fewer—albeit larger—copy operations and better program-perceived locality. Large regions, however, can also lead to unnecessary copying to the shadow state, since the entire region is always copied to the shadow state even when a single byte within the region is modified in the entire checkpoint interval. These observations highlight the different tradeoffs involved in selecting the optimal memory region size.

To investigate the tradeoffs, we measured the LMC-induced throughput degradation for the three most popular web servers according to the experimental setup adopted in our evaluation (Section 2.7). We repeated the experiment across several different memory region sizes and reported the results in Figure 2.3. The figure shows that, in all cases, the throughput increases with the region size for $\rho \leq 128\ bytes$ and decreases for $\rho > 128\ bytes$, due to the excessive amount of unnecessary copying outweighing the benefits derived from increased locality and degrading the overall run-time performance. Based on this experiments, LMC currently assumes $\rho = 128\ bytes$ in its default configuration.

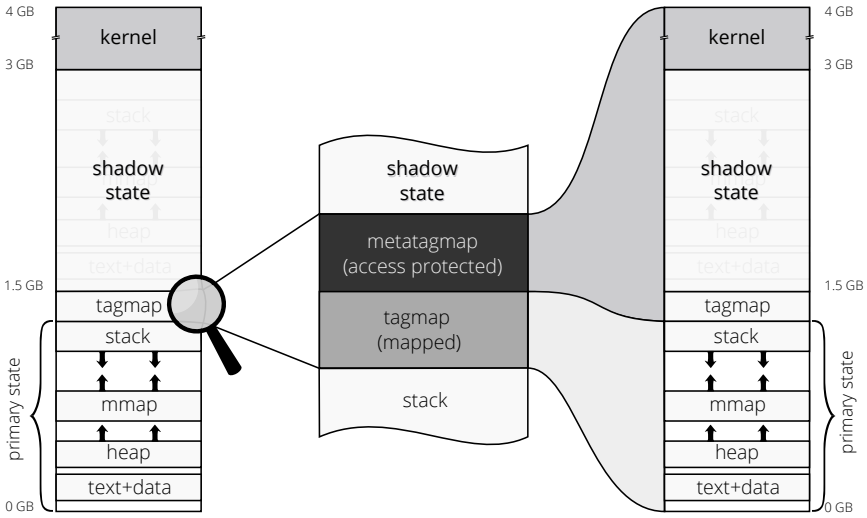


Figure 2.4: Tagmap and metatagmap mappings to memory regions.

2.4.2 Metatagmap

Memory checkpointing is often used in error recovery scenarios, where one cannot safely assume that program-issued store instructions are free from errors that could corrupt arbitrary memory address space areas, including the shadow state and the tagmap itself. To protect the program against accidental metadata corruption, LMC can be configured to use a separate *metatagmap*—allocated within the tagmap and with a similar rationale—to map all the store instructions that erroneously specify a destination address inside the shadow state, tagmap, or metatagmap itself. Inhibiting access to the metatagmap using page protection mechanisms—a strategy inspired by prior work on taint tracking techniques [147]—is sufficient to prevent arbitrary metadata corruption on invalid store instructions and induce fail-stop behavior—i.e., segmentation fault—that can be effectively handled by error recovery techniques instead. LMC's tagmap and metatagmap mapping strategy is depicted in Figure 2.4.

2.4.3 Resetting the Tagmap

The most straightforward tagmap implementation is a bitmap with boolean tag values. This strategy, however, requires zeroing out the entire bitmap every time LMC needs to reset the tagmap at the end of a given checkpoint interval. For this purpose, an option is to simply `bzero` the entire tagmap. Another option is to use the `mmap` system call to request the kernel to overmap the tagmap with zero pages. The latter strategy is generally more efficient—only the pages of the tagmap that are actually mapped in are zeroed—but still introduces nontrivial performance costs associated to increased zeroing, page faulting—demand paging dictates zero pages to be only mapped in at first access—and setting up a new memory area in `mmap`. To eliminate the latter cost, our current LMC implementation relies on the `madvise` system call and the `MADV_DONTNEED` flag to efficiently repopulate a given memory area with zero pages without recreating the area, a strategy commonly employed in modern memory allocators [21]. The other costs, however, are much harder to eliminate without dedicated kernel support.

To avoid incurring such costs at every checkpoint interval, LMC opts for a more sophisticated tagmap implementation based on *epoch* numbers. For this purpose, the LMC checkpointing library keeps track of a 1-byte global epoch number incremented at every checkpoint. Every tag in the tagmap is set with the current epoch number when the corresponding memory region is first modified in a given checkpoint interval. To check whether a particular region has already been saved, LMC simply compares the corresponding tag in the tagmap with the current epoch number. This scheme eliminates the need to zero out the tagmap at every checkpoint interval, only forcing LMC to repopulate the entire tagmap with zero pages when epoch numbers run over, i.e., every 255 intervals assuming 8-bit epoch numbers.

2.4.4 Thread Safety

LMC can natively support thread safe behavior, but our current implementation disables thread safety by default. This is to eliminate extra tagmap management complexity, which is often unnecessary in practice given that memory checkpointing applications typically enforce thread safety on their own to guarantee a sound checkpointing model in a multithreaded context. For example, error recovery techniques rely on a well-defined thread model to implement their recovery activities, assuming nonthreaded execution by construction [53; 159], allowing only one thread to enter a new checkpoint

interval and explicitly blocking all the other threads [115; 152; 68], or only allow checkpoint intervals that have been proven thread safe by static program analysis [157].

Enforcing thread safety at the memory checkpointing level requires LMC to synchronize accesses to LMC-maintained metadata, so that no race conditions can result from multiple threads writing into the same memory region at the same time. To address this problem, the obvious solution is to serialize accesses to the tagmap, epoch numbers, and shadow state using dedicated locks—e.g., mutexes. This strategy, however, may introduce nontrivial complexity and lock contention overhead at runtime. For this reason, LMC opts for a simpler solution which piggybacks on the synchronization mechanisms already present in the original program. To this end, LMC lowers the memory region size to 1 byte—similar to prior memory shadowing techniques for multithreaded programs [111]—a strategy which naturally yields thread safety by construction as long as the original program did not contain race conditions with two threads attempting to modify the same memory byte at the same time. This assumption may be overly conservative in error recovery applications, but such applications generally deal with thread safety explicitly, as mentioned earlier. Further, LMC needs to translate all the atomic instructions—e.g., atomic increment—into fully synchronized store operations.

2.5 Optimizations

This section details the optimizations adopted in our prototype to minimize the LMC-induced performance overhead.

2.5.1 Reducing Instrumentation Costs

Instrumented store instructions introduce nontrivial performance costs even if the target memory region has already been checkpointed. Such costs have two main sources: (i) new call instructions to the `store_hook` function and (ii) new load and branch instructions to check the tagmap as part of the checkpointing activity. As a result, the LMC optimization pass can minimize the performance overhead by preventing instrumentation of store instructions that static analysis can prove (i) *redundant*—i.e., operating on already checkpointed memory regions—or (ii) *transient*—i.e., operating on short-lived memory regions whose effects are never exposed outside the associated check-

point interval and are thus not relevant for the checkpoint. Further, the LMC optimization pass *aggregates* store instructions for which the pass can statically assess good spatial locality.

Redundant Stores

To avoid instrumenting redundant stores, the LMC optimization pass examines each instrumented store instruction I and its pointer operand p, and creates a set S including all the other store instructions that store into the memory location pointed to by p. The analysis establish this fact by checking for equivalence of the pointer operands, i.e., stays conservative in the case of pointer aliasing. In a second step, LMC tests for each instruction $I_{canditate}$ included in S, if $I_{canditate}$ is dominated by I—i.e., I is proven to be always executed before $I_{canditate}$—and if so, un-instruments $I_{canditate}$.

Transient Stores

To avoid instrumenting transient stores, the LMC optimization pass seeks to identify both *heap transient stores*—i.e., store instructions referring to heap objects allocated and freed within a single checkpoint interval—and *stack transient stores*—i.e., store instructions referring to stack objects in short-lived functions, whose lifetime never spans across multiple checkpoint intervals by construction.

Heap transient stores are identified using *checkpoint escape analysis*, which follows the same static analysis strategy used in standard thread escape analysis techniques [120]. To assess whether a memory object escapes a function in the checkpoint interval, LMC relies on data structure analysis [81], an efficient context-sensitive and field-sensitive points-to analysis implemented in LLVM [79]. In particular, LMC follows the approach adopted by poolalloc [80] to identify function-local memory pools. This strategy allows LMC to identify store instructions that never *escape* a given checkpoint interval and can thus be safely left uninstrumented in the final binary.

To prevent instrumenting stack transient stores, LMC eagerly checkpoints the active call stack at checkpointing time and relies on the points-to information provided by data structure analysis [81] to avoid instrumenting *all* the store instructions that are statically proven to always refer to stack objects within the checkpoint interval. This strategy reflects the intuition that checkpoint requests are usually issued when the amount of state active on the call stack—and thus the amount of data to checkpoint eagerly—is relatively

Server	Long-lived stack in kB
nginx	1224
lighttpd	712
httpd	784
prostgresql	3048
bind	352
proftpd	5784
pureftpd	4608
vsftpd	1088

Table 2.1: Long-lived call stack size for different server applications.

small. This is especially evident in common request-oriented checkpointing models [53; 83] in long-running applications, which typically yield minimal long-lived call stack state at memory checkpointing time.

Table 2.1 confirms our intuition, showing that the size of the long-lived call stack for all the server applications considered in our evaluation is typically smaller than 1 memory page, which introduces minimal copying costs—and thus minimal performance degradation—at checkpointing time.

Aggregating Store Instrumenation

While not instrumenting redundant store instructions is effective for single memory locations, it cannot account for stores to different, but spatially collocated memory locations. Aggregating the instrumentation for these collocated store instructions, however, has the advantage that stores to the same underlying memory region are potentially only instrumented once.

LMC performs this optimization for store instructions into the same underlying object by constructing dominator chains of store instructions for each memory object (e.g., `struct`). The pass identifies the underlying memory object by stripping all constant offsets of LLVM's `getelementptr` instruction, and stays conservative in the case of pointer aliasing. After establishing the modified range of the memory object for each chain, all the store instruction in the chain are left uninstrumented and a call to LMC's `store_hook` covering the entire region is placed before the chain's leading instruction.

2.5.2 Reducing Checkpointing Costs

In its current version, our LMC prototype naturally imposes an always-on checkpointing strategy, given that all the relevant store instructions always run instrumented throughout the execution. In other words, either an implicit or explicit checkpoint interval is always active during the execution. While the cost of copying is gradually amortized throughout the execution when no explicit checkpoint request is issued—i.e., the same memory region is never checkpointed more than once—or can even be eliminated altogether by explicitly setting all the tags in the tagmap, the running program is still exposed to nonmarginal tagmap management costs.

To eliminate such costs, an option is to rely on program instrumentation to implement a simple basic block cloning strategy, a well-known technique incurring a relatively small memory [65] and performance impact [146; 55]. Basic block cloning results in a final binary containing two versions of each basic block and additional code to efficiently switch from one version to another on demand. For our purposes, one version would reflect code from the original uninstrumented program and the other version would reflect the corresponding code with store instructions instrumented to perform memory checkpointing. We are planning to thoroughly investigate the impact of such a basic block cloning strategy in our future work.

2.6 Alternative Techniques

This section provides a general overview of existing memory checkpointing techniques and draws a high-level comparison with LMC. An experimental comparison, in turn, is presented in Section 2.7. We focus here on user-level memory checkpointing, and refer the reader to existing surveys [46; 123] for more intrusive kernel-level [56; 85; 76; 151; 52; 2; 77; 106] and VMM-level [26] checkpointing techniques.

2.6.1 Fork-based Checkpointing

The most common way to implement page-granular checkpointing at the user level is to rely on the copy-on-write semantics supported by the `fork` system call. For our purposes, memory checkpointing can be simply implemented by spawning a new child process at the beginning of a checkpoint

interval and programmatically terminate the process at the end of the interval. For its simplicity and isolation properties, fork-based checkpointing has been widely used in prior solutions [51; 128; 112; 117; 152]. This technique, however, can introduce substantial checkpoint-time overhead—`fork` requires creating a new process context—and is not entirely transparent—a new process instance is made "*visible*" to the running program.

2.6.2 Mprotect-based Checkpointing

Another popular page-granular checkpointing strategy is to use the `mprotect` system call to write-protect all the user memory pages and intercept the resulting write faults from a user-level `SIGSEGV` signal handler, a mechanism frequently used to implement generic user-level page fault handling. This mechanism can be used to implement COW semantics similar to fork-based checkpointing or, as an alternative, to implement dirty page tracking entirely in user space—and incrementally copy dirty pages at the end of each checkpoint interval, starting with an initial full memory checkpoint. After copying (or tracking) the faulting page, the write protection can be removed and reestablished only at the next checkpoint request. When compared to fork-based checkpointing, this technique typically introduces a more modest checkpoint-time overhead, but, at the same time, substantially increases the cost of page fault handling and lowers the isolation guarantees—the checkpointed data resides in the same address space as the running program and extra protection mechanisms are necessary to prevent data corruption.Further, this technique cannot be made application-transparent without additional recovery mechanisms in place, given that kernel execution page faulting on a write-protected page may result in a system call returning an error code to the application—i.e., `EFAULT`, according to POSIX [112]. For all these reasons, mprotect-based checkpointing has found relatively limited use in prior checkpointing solutions [33; 112; 119].

2.6.3 Soft Dirty Bit-based Checkpointing

A popular dirty page tracking strategy suitable for page-granular memory checkpointing is to rely on the dirty bit information maintained by the hardware in individual page tables entries. Traditional Unix systems do not directly expose dirty bit information to user programs and typically require nontrivial kernel extensions to implement reliable dirty page tracking for checkpointing purposes [136], a strategy often explored in prior kernel-level

solutions [136; 85; 52]. On recent Linux releases, however, kernel support adopted for emerging checkpoint-restart frameworks [13] allow user programs to implement dirty page tracking by periodically reading and clearing software-maintained dirty bits—or *soft dirty bits*. This scheme, however, still requires extra protection mechanisms to prevent checkpoint data corruption and does not efficiently scale to large address spaces—reading soft dirty bits requires scanning all the mapped memory regions in the address space. Other user-level solutions have also suggested emulating soft dirty bit tracking using block-level checksumming [50; 100], an approach which still shares the same limitations of soft dirty bit-based tracking.

2.6.4 Undolog-based Checkpointing

Undolog-based checkpointing relies on program instrumentation techniques to log all the store instructions—i.e., their data, size, and target addresses—issued by the program within a checkpoint interval and revert the logged changes at restore time. Both dynamic [115] and static instrumentation [159; 83] techniques can be used to instrument the store instructions—we implemented the latter approach in our prototype implementation (similar to the instrumentation strategy used in LMC) to ensure a fair experimental comparison. Previous work on instrumentation-based undolog approaches [159] apply optimizations similar to LMC's "uninstrumentation" optimizations, but do not thoroughly evaluate their impact on runtime performance. To prevent checkpoint data corruption, the instrumented code needs to perform bounds checking at every store instruction to verify that the target address is not in the range in use by the undolog itself.

2.7 Evaluation

We implemented LMC on Linux for Intel x86 and x64 architectures. We evaluated the resulting solution on a Intel Core2 E6550 clocked at 2.4 GHz and equipped with 4 GB of RAM (32-bit experiments) and a Intel Core i5-3340M clocked at 2.4 GHz with 8 GB of RAM (64-bit experiments).

For our experimental evaluation, we selected the three most popular open-source web servers—nginx (v0.8.54), lighttpd (v1.4.28), and Apache httpd (v2.2.23)—the three most popular open-source ftp servers—proftpd (v1.3.3), pureftpd (v1.0.36), and vsftpd (v1.2.1)—the most popular open-source name

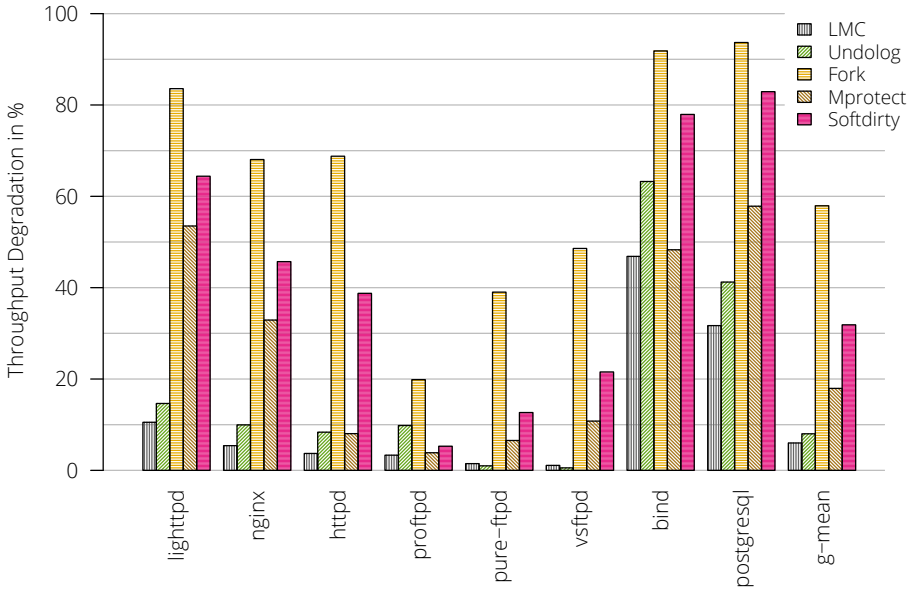


Figure 2.5: Normalized throughput for our server programs across different memory checkpointing techniques (64-bit results).

server—bind (v9.9.3)—and a mainstream open-source database server—postgresql (v9.0.10). We also considered all the C benchmarks in the SPEC CPU2006 benchmark suite. We instrumented all our test programs across different memory checkpointing techniques and enabled all the optimizations described in Section 2.5 for all the compiler-based checkpointing techniques (undolog and LMC itself) unless otherwise stated.

To stress the web servers, we relied on the Apache benchmark (*AB*) [11] part of the Apache httpd suite. To emulate a realistic workload, we configured *AB* to issue a total number of 25,000 requests with 10 concurrent connections and 10 requests per connection through the loopback device. To benchmark the FTP servers, we relied on the pyftpbench benchmark [4], configured to open 100 control connections and request 100 1 KB-sized files per connection. Finally, we relied on the sysbench [5] and queryperf [10] benchmarks to evaluate postgresql and the bind name server, respectively. We ran all our experiments 11 times—while checking that the CPUs were

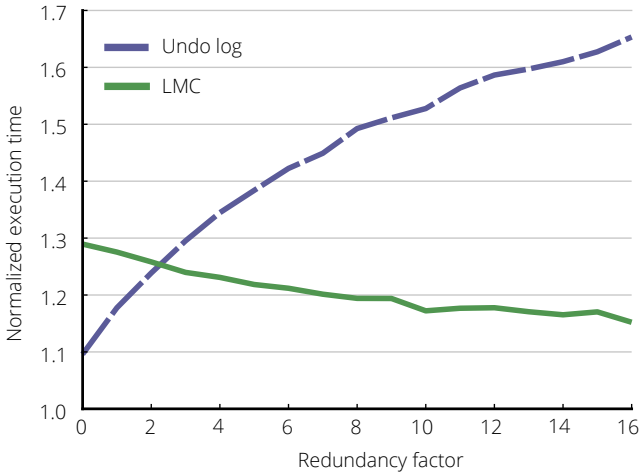


Figure 2.6: Checkpointing-induced microbenchmark execution time for different redundancy factors normalized against the baseline (64-bit results).

fully loaded throughout our tests—and reported the median.

2.7.1 Checkpointing Performance

To evaluate the checkpointing-induced performance overhead, we measured the throughput degradation on our server programs while checkpointing at every client request, following the common request-oriented checkpointing model adopted in prior work [53; 83]. Figure 2.5 presents our results for 64-bit Linux—we omit 32-bit results exhibiting similar behavior.

As we can see in the figure, fork-based checkpointing induces the highest checkpointing performance overhead compared to all the other techniques (88.5% degradation, geometric mean). Further, mprotect-based checkpointing is the top-performing page-granular checkpointing technique in this scenario (55.5% degradation, geometric mean). In particular, mprotect-based checkpointing reported remarkable performance for programs that exhibit good locality, e.g., Apache httpd, which memory profiling revealed modifying the smallest number of memory pages in the selected checkpoint in-

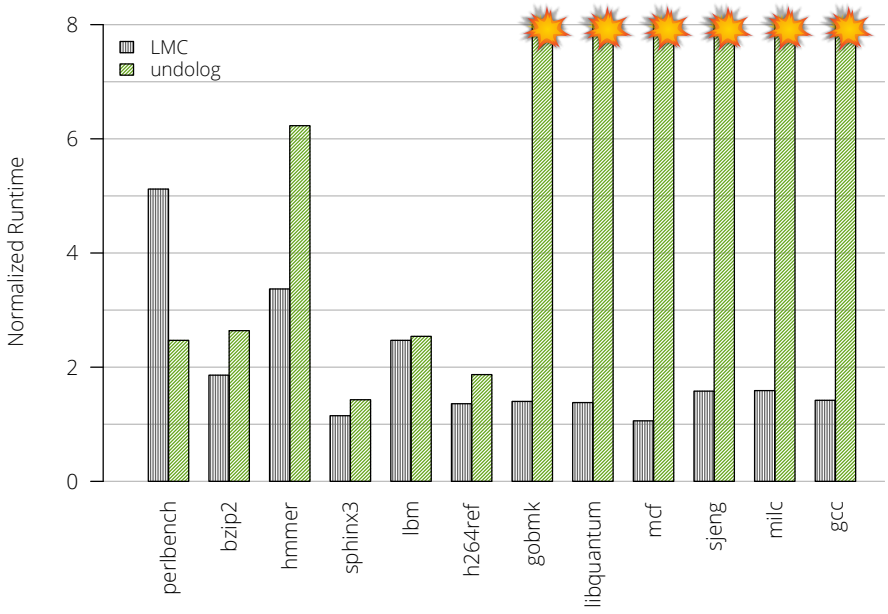


Figure 2.7: Checkpointing performance on SPEC for LMC and undolog (64-bit results).

terval. Finally, instrumentation-based techniques significantly outperform page-granular techniques in most cases, as anticipated. In particular, our results show that LMC performs comparably, and even better on average (15.0% vs 20.0% degradation, geometric mean), than undo log-based checkpointing. In detail, LMC consistently outperforms undolog-based checkpointing for memory-intensive programs, e.g., bind, and programs that exhibit significant write locality, e.g., Apache httpd. In both scenarios, a large number of duplicate writes can cause the undolog to grow quickly, disrupting spatial locality and increasing cache trashing.

2.7.2 Effectiveness of the Optimizations

Figure 2.7 shows the checkpointing performance for the SPEC CPU2006 benchmark suite using our LMC and the undolog checkpointing technique. To simulate an event-based recovery scenario, we identified an inner loop inside the programs. During each iteration of the loop, a checkpoint is taken,

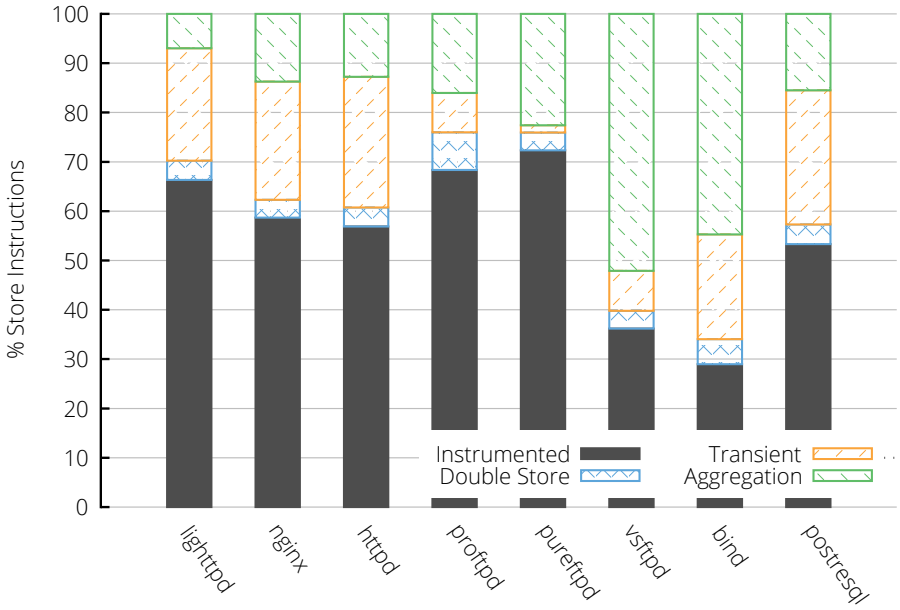


Figure 2.8: Relative number of uninstrumented store instructions for different server programs.

resulting in checkpoint intervals ranging from 13 microseconds (perlbench) to 391 seconds (milc). A special case is gcc, in which case only one checkpoint is taken, since no inner loop could be identified. The results in the figure are ordered by checkpoint frequency.

We limited the undolog to 4 GB leading to programs crashing, due to undolog overflows in cases where the checkpoint interval is fairly large. Programs start to crash when checkpointed with a frequency of 0.39 seconds (gobmk). This underlines the necessity of memory boundedness. Further, the figure shows that in nearly all cases LMC outperforms the undolog-based technique. Only for perlbench, which is checkpointed with an extremely high frequency, the undolog performs better than LMC. In this case, the very low cost incurred by the undolog to start a new checkpoint—i.e., simply resetting the index into the log—is the key to better performance.

To evaluate the effectiveness of the optimizations operated by LMC to reduce the instrumentation overhead, we measured (1) the number of store instructions that are uninstrumented by the different optimization stages and

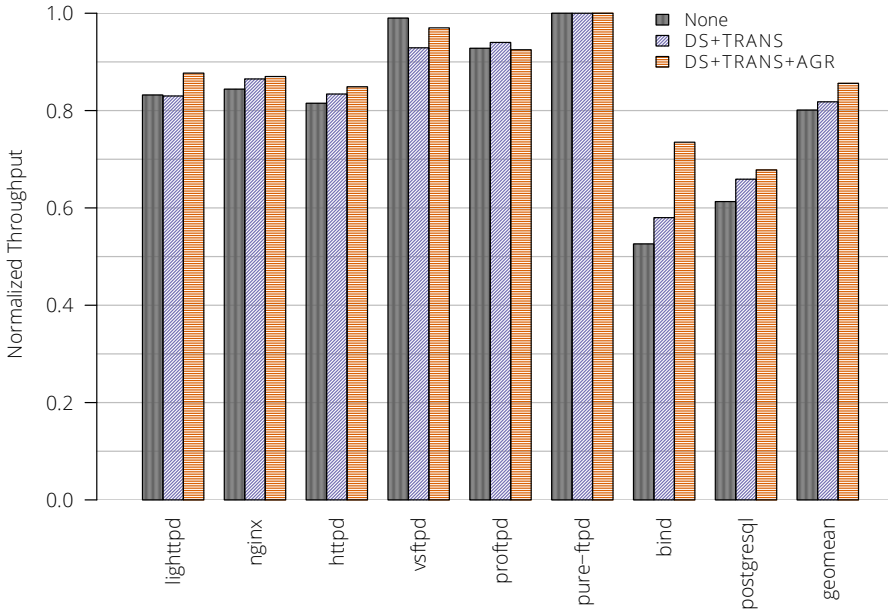


Figure 2.9: Throughput of server programs with LMC using different combinations of optimizations.

	Baseline PSS	LMC			Undolog			mprotect		
Requests:		1	10	100	1	10	100	1	10	100
nginx	1872 KB	20 %	19 %	20 %	9 %	21 %	136 %	12 %	12.6 %	12 %
lighttpd	851 KB	33 %	43 %	33 %	8 %	32%	184 %	16 %	19 %	15 %
httpd	3257 KB	52 %	54 %	52 %	26 %	128 %	1193 %	16 %	16%	15 %
proftpd	71982 KB	94 %	94 %	93 %	420%	900 %	5900 %	8 %	5 %	5 %
pureftpd	268 KB	41 %	39 %	39 %	14 %	33 %	208 %	111 %	121 %	195 %
vsftpd	89 KB	29 %	35 %	35 %	5 %	9 %	11 %	5 %	13%	11 %
bind	8897 KB	27 %	26 %	27 %	79 %	94%	218 %	2 %	3 %	1 %
postgresql	20919 KB	11 %	29 %	13 %	412 %	470 %	1104 %	16 %	5 %	1 %

Table 2.2: Checkpointing-induced PSS increase for our server programs across different memory checkpointing techniques and intervals.

(2) the resulting impact on the run-time performance of the server programs.

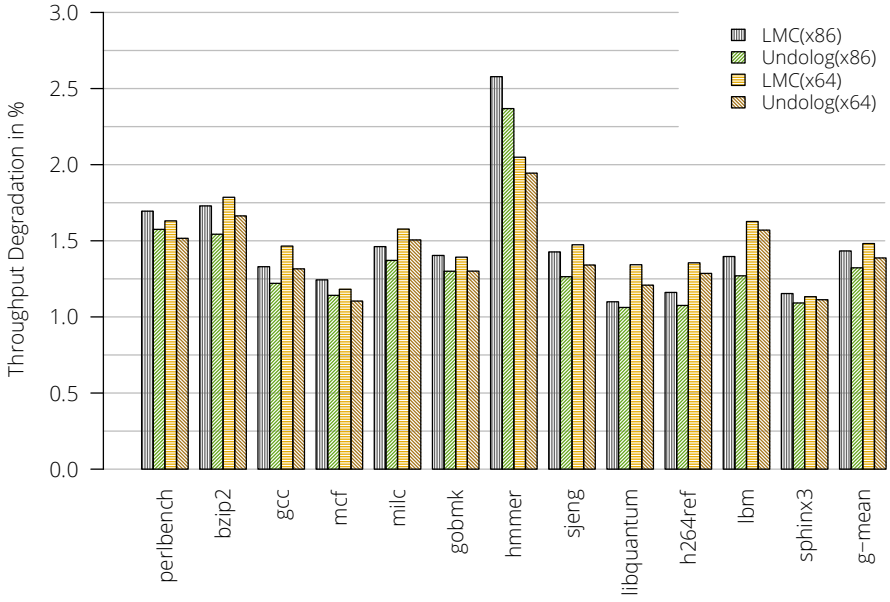


Figure 2.10: Instrumentation-induced execution time for the C programs in the SPEC CPU2006 benchmark suite normalized against the baseline. Actual state saving is disabled in the `store_hook`.

Uninstrumented Store Instructions

Figure 2.8 shows the relative amount of store instructions that are uninstrumented by the different optimization stages. The results show that our optimizations are generally effective, leading to a reduction of instrumented store instructions of between 29 % (bind) and 72 % (pure-ftpd). In addition, the effectiveness of the double store optimization (3.6–7.6 % of all store instructions uninstrumented) is outweighted by the effectiveness of the transient store and aggregation optimizations.

Further, our results show that, in some cases, (proftpd, nginx, httpd, and postgresql) the aggregation optimization is able to uninstrument the largest fraction of store instructions. In the other cases (lighttpd, pureftpd, vsftpd, and bind), the transient store optimization is able to uninstrument the largest fraction of store instructions. We attribute this behavior to the latter programs' heavier use of stack-allocated variables.

Run-time Impact of the Optimizations

Figure 2.9 shows the normalized throughput of our unoptimized (None), doubles store and transient store optimized (DS+Trans) and fully optimized (DS+Trans+AGR) server programs. The general trend shows that uninstrumenting store instructions is—as expected—reflected in better run-time performance, leading to an average performance improvement of 1.8 % for the DS+TRANS case and an additional 4.8 % when also enabling the aggregation optimization.

At the same time, our results also show that an increase in the number of uninstrumented instructions induced by a certain optimization is not necessarily reflected in an equally-sized performance gain, since the uninstrumented store instructions may happen to lie in cold (i.e., nonperformance critical) code paths. This was, for example, the case for postgresql, where the aggregation optimization uninstruments the largest fraction of store instructions but has a smaller impact on the overall performance. Finally, in some cases (lighttpd, vsftpd, and proftpd), our results seem to suggest that optimizations may occasionally have a slightly negative performance impact. In practice, the reported slowdowns are well within the noise caused by optimization-induced memory layout changes [99].

2.7.3 Impact of Duplicate Writes

To compare the previously noted impact of duplicate writes on instrumentation-based memory checkpointing techniques, we relied on a homegrown microbenchmark, which runs through 5000 loop iterations, each of which checkpoints the entire memory image and subsequently writes 1 KB of data into a 128 KB memory range—sampled uniformly at each iteration. To simulate duplicate writes with a redundancy factor of R, we repeated each write operation inside the loop R times. Figure 2.6 shows the time to complete the checkpointing-enabled version of our (64-bit) microbenchmark normalized against the baseline—for growing values of R. As we can see, undolog-based checkpointing yields better performance only for $R = \{0, 1\}$ but it is increasingly outperformed by LMC for greater values of R.

2.7.4 Instrumentation Performance

To evaluate the performance overhead induced by instrumentation-based memory checkpointing techniques with no checkpoint operation issued dur-

ing regular execution, we measured the time to complete an instrumented version of the C programs in the SPEC CPU2006 benchmark suite compared to the baseline. As checkpointing-induced costs are not considered, we disabled logging for undolog-based checkpointing and epoch number management for LMC.

Figure 2.10 shows that the instrumentation-induced performance overhead lies between 17 % and 206 %. Further, the overhead for LMC is slightly higher than that for the undolog. This is especially the case for libquantum and lbm, which introduce significant cache pressure [149]. The latter is further increased by tagmap management operations operated by LMC's `store_hook` function. Finally, Figure 2.10 shows that the overall overhead is slightly higher for 64-bit systems, which we attribute to the position-independent code used by our prototype implementation on such systems. The apparent speedup reported for hmmer, in turn, is likely caused by a higher instruction per cycle ratio on 64-bit architectures [149].

2.7.5 Memory Usage

To evaluate the checkpointing-induced memory usage overhead, we measured the *Proportional Set Size* (*PSS*)—physical memory usage normalized to account for shared memory pages in a multiprocess context—increase on our servers programs while checkpointing at every $R = \{1, 10, 100\}$ requests—highlighting the memory usage growth for the different techniques.

Table 2.2 presents our findings—omitting fork-based checkpointing results, comparable to mprotect-based results but harder to stabilize with non-atomic PSS measurements across multiple processes. As expected, LMC generally consumes more memory than page-granular checkpointing techniques (32% vs 12% increase with $R = 1$, geometric mean), but induces a similarly limited and steady memory usage increase across different checkpoint intervals. Undo log-based checkpointing, in contrast, introduces a substantial memory usage increase, which grows very quickly as we relax the duration of the checkpoint interval. For example, with $R = 10$, undolog-based checkpointing induces a PSS increase of 900 % in the worst case—i.e., proftpd—which quickly grows up to 6000 % with $R = 100$.

2.8 Conclusion

Existing high-frequency memory checkpointing techniques operating at the user level force users to tradeoff performance and memory usage guarantees, a painful compromise when systems reliability is at stake. To address these concerns, this paper presented LMC, a new memory checkpointing technique based on a compiler-assisted shadow state organization which efficiently implements byte-granular copy-on-write semantics. To evaluate the viability of our approach, we implemented LMC for generic 32- and 64-bit Linux programs and evaluated it on eight popular open-source server applications using the common request-oriented checkpointing model. Our experimental results show that LMC matches the performance guarantees of state-of-the-art instrumentation-based strategies—i.e., undolog—while also providing much stronger memory usage guarantees.

CHAPTER 3

SPECULATIVE MEMORY CHECKPOINTING

3.1 Introduction

Memory checkpointing—the ability to snapshot/restore the memory image of a running process or set of processes—has recently gained momentum in several application domains. In automatic error recovery applications, memory checkpointing enables fast and safe recovery to known and stable program states [152; 135; 53; 83; 54; 121; 117; 125; 51; 68; 115]. In debugging applications, it enables users to efficiently navigate through several program states observed during the execution, while empowering advanced debugging techniques such as reverse/replay debugging [129; 128; 61; 72]. Memory checkpointing also serves as a key enabling technology for important first-class programming abstractions like software transactional memory [83], application-level backtracking [159; 33], and periodic memory rejuvenation [144].

Such application domains require very frequent checkpoints in real-world scenarios. For instance, automatic error recovery techniques rely on frequent checkpoints to mask failures to the clients [143]. This is typically ac-

complished by checkpointing the program state at every client request [53; 83]—or at carefully selected rescue points [125; 115; 135; 69]. In advanced debugging techniques, frequent checkpoints allow users to quickly navigate through arbitrary points in the execution history [72; 69]. Finally, first-class programming abstractions implemented on top of memory checkpointing, such as application-level backtracking, typically yield a very high checkpointing frequency by construction [33].

Traditional memory checkpointing techniques rely on commodity hardware—a strategy that provides superior deployability compared to instrumentation-based strategies [115; 152; 32; 84; 38; 159; 83; 139; 140]—to incrementally copy memory pages that were modified by the running program [52; 136; 119; 77; 125; 72; 51; 128; 112; 33; 117]. While *incremental* memory checkpointing is regarded as an efficient alternative to *disk-based* or *full* memory checkpointing [113], it still incurs nontrivial memory tracing costs for every taken checkpoint, resulting in relatively infrequent checkpoints used in practice.

In this paper, we present *Speculative Memory Checkpointing* (*SMC*), a new technique for high-frequency page-granular memory checkpointing. SMC seeks to improve upon current techniques to allow for very *high-frequency* checkpointing at a period that is below the one millisecond boundary, even making it possible to checkpoint *every* request in a highly loaded server. To fulfill this goal, SMC sets out to minimize the memory tracing costs of incremental checkpointing by *eagerly* copying the *hot* (frequently changing) pages, while *lazily* tracing and copying at first modification time only *cold* (infrequently changing) memory pages. Thus, SMC combines the advantages of full memory checkpointing (efficient bulk copies) with that of incremental memory checkpointing (copy only when needed). The key challenge is to find the optimal trade-off between eagerly copying too many memory pages—i.e., unnecessary memory copying costs—and copying an insufficient number of pages which may result in unnecessary memory tracing costs for every checkpoint.

To address this challenge, SMC relies on a general *writable working set* (WWS) model [40] to detect the memory pages that change most often—the ideal candidates for our speculative copying strategy. To obtain fresh and accurate estimates, our implemented SMC framework supports well-established working set estimation (WSE) algorithms. In addition, we complement our framework with *GSpec*, a novel writable WSE algorithm specifically tailored to high-frequency memory checkpointing. GSpec follows a blackbox optimization strategy inspired by genetic computing [96]. The lat-

ter approach provides SMC with a *self-tuning* and *self-adapting* working set estimation strategy by design, which relies on no program-specific parameters and ensures fresh and accurate estimates across several different real-world workloads. This is in stark contrast to traditional WSE algorithms, which, while well established in several application domains such as dynamic memory balancing [164; 66; 90; 142; 89; 37], garbage collection [148; 154; 58], virtual machine restore [156; 155] and live migration [153], are generally ill-suited to high-frequency memory checkpointing. In particular, these algorithms impose a stringent performance-accuracy trade-off that typically results in a nontrivial overestimation of the real writable working set [24]. This is perhaps acceptable in many traditional applications (e.g., dynamic memory balancing with sporadic memory pressure), but leads to substantial overcopying, and thus overhead, for SMC.

Contributions The contributions of this paper are fourfold. First, we present an in-depth analysis of prior page-granular memory checkpointing techniques, evidencing their direct and indirect memory tracing costs. Our investigation uncovers important bottlenecks for prior solutions in high-frequency checkpointing contexts and serves as a basis for our design. Second, we present *Speculative Memory Checkpointing* (*SMC*), a new technique for high-frequency memory checkpointing based on (several possible) writable working set algorithms. Third, we introduce *GSpec*, a novel WSE algorithm which draws inspiration from genetic algorithms to speculatively copy memory pages that are most likely to change in the next checkpointing interval. Finally, we implemented and evaluated a kernel-module-based SMC framework with support for GSpec and other WSE algorithms, demonstrating its performance benefits in high-frequency checkpointing scenarios. Our results demonstrate that our WSE-based strategy is accurate, efficient, robust to workload variations, and effectively reduces the run-time overhead of high-frequency memory checkpointing at the cost of modest memory overhead.

3.2 Background

A straightforward way to implement process checkpointing involves freezing the execution and taking a snapshot of memory by copying it [2; 42; 56; 76; 106; 118]. Even though this approach suffices in certain domains, like process migration, it is wasteful and slow in domains where frequent checkpoints need to be made, as it requires the process to stop for a significant

amount of time and copies potentially large amounts of data indiscriminately.

A more efficient strategy is to rely on *incremental checkpointing*. Incremental checkpointing builds a checkpoint gradually—minimizing the time that a process is suspended, and reducing the amount of data to copy. We can generate incremental checkpoints in two ways. We can make a full snapshot in the beginning, and then track and save all modifications, so we can add them to the snapshot at the next checkpoint [52; 136; 13; 119; 112]. To roll back, all memory is restored using the maintained snapshot. Alternatively, we can do the inverse and copy only the data that are modified after a checkpoint, right before they are overwritten [77; 125; 72; 51; 128; 112; 33; 117]. To roll back we restore only the overwritten data using their copies. We will refer to the former solution as "*copy new data*" (it copies the new data at checkpoint time), and the latter as "*copy old data*" (it copies the old data prior to overwriting them).

Traditional incremental checkpointing mechanisms are usually page-granular, that is, a memory page is the smallest data block copied (although more fine-grained techniques exist [159; 44; 115; 83; 152]). Below we discuss the core mechanisms and techniques employed by these approaches.

Hardware dirty bit Incremental checkpointing techniques rely on *dirty page tracking*. Modern memory management units (MMUs) include a *dirty bit* for each entry in the page tables maintained by the operating system (OS), which is set by the hardware when a page is written. The bit is used by the OS to, for example, determine which pages need to be flushed to disk. Directly using this dirty bit to detect modified pages is potentially fast, but requires extensive changes to the OS kernel [136; 52; 76; 2] which is neither attractive, nor likely to help deployability.

Soft dirty bit Linux also offers a *soft dirty-bit* mechanism, made available to user space through the `proc` file system, which provides the same functionality with HW dirty bits, albeit not as fast (see Section 3.3).

Write Bit The write bit [136], also provided by the MMU, controls whether a virtual memory page can be written. It is often leveraged for checkpointing. For example, when the dirty bit is missing, it is used to emulate the functionality. Briefly, write protecting a page will generate faults on writes. By capturing the faults, we identify the dirty pages and maintain our own soft dirty bit.

Copy-on-write (COW) semantics "Copy old data" approaches that save memory pages on-the-fly are in their majority utilizing the write bit and COW semantics. The most well known use of COW is in the `fork` system call in Linux. `fork` creates a new process, identical to the parent process invoking it, but instead of duplicating all memory pages, the two processes share the same pages which are now marked as read-only and COW. When one of them writes to a page, a fault is generated, causing the kernel to create a copy of the page. User-space checkpointing mechanisms are using `fork` to copy pages on-demand, but COW semantics can be also used directly from within the kernel, by setting the appropriate bits in the page table.

Page Checksums An alternative for determining dirty pages without relying on dirty bits involves periodically calculating the checksum of pages and comparing them over time. The precision of this approach is subject to the accuracy of the algorithm used for computing the checksums [100]. One could also compare the contents of individual memory pages directly [89], but this strategy is generally less space-efficient and more expensive due to poor cache behavior.

3.3 SMC

Checkpointing based on the write bit, which primarily includes approaches using COW, does not require changes within the kernel and can efficiently roll back, but suffers increasing overhead as the number of pages in a checkpoint grows. Besides the unavoidable cost of copying pages, handling page faults also induces overhead. Given a way to establish which pages are going to be modified after a checkpoint, we could avoid the page-faulting overhead and copy only the pages that need to be saved. This is the key idea behind *Speculative Memory Checkpointing* (*SMC*).

Knowing exactly which pages are going to be written after a checkpoint is a difficult problem, which is addressed by SMC through approximation, similar to working set estimation (WSE). Pages that are expected to receive writes are considered to be *hot* and not write-protected but eagerly copied when hitting a checkpoint. In "copy old data" approaches, they are copied and discarded on the next checkpoint, while in "copy new data" approaches, they are copied into the full memory snapshot. The speculative approach followed by SMC can be examined based on *accuracy* and *performance*.

#	Test	CPU cycles
	COW tests	
1	Write to a page after fork.	4016
2	Write to a page, but also fork and terminate the child.	139576
	Copying tests	
3	Copy a page and write into it.	492
4	Same as the above but also checksum page data.	1228
	Soft dirty (SD) bit tests	
5	Write to a page, read SD bits, and copy page.	16136
6	Same as the above, but clear the SD bits first.	33148

Table 3.1: Microbenchmarks that test the various operations performed by incremental checkpointing. The table lists the average number of CPU cycles consumed after running each test 1000 times.

Accuracy A speculative approach is accurate when it can continuously determine the pages that will be written during a checkpoint. Missing hot pages triggers page faults and degrades performance. We refer to such errors as *undercopying*. Respectively, marking rarely written pages as hot leads to more copying than needed, also degrading performance. We refer to these errors as *overcopying*.

Performance Three key factors affect the performance: the overhead of the algorithm that speculates the set of hot pages, the number of undercopying errors, and the number of overcopying errors. Obviously, a very accurate prediction algorithm can reduce the number of errors, but if that comes with an elevated cost, then it overshadows the lack of errors. Similarly, a large number of errors can make SMC more expensive than traditional incremental approaches (e.g., if none of the hot pages are actually written).

Design To guide the design of SMC, we carefully evaluated the impact of common operations performed by traditional incremental checkpointing techniques. Table 3.1 presents our results. An immediately evident result is the substantial overhead introduced by checkpointing strategies using COW pages from user space. This requires forking a new process, managing it, and terminating it when taking a new checkpoint, while the kernel takes care

of copying a page when it is written. The latter is quite fast taking only 4016 CPU cycles, while forking, etc. requires 139,576 cycles (see lines 1 and 2 in Table 3.1). Shedding this overhead is an important factor for high-frequency checkpointing which involves more and potentially shorter (in duration) checkpoints. For this reason, our SMC framework bases its operations in a kernel module that exports checkpointing primitives to user space. A complete user-space solution would have otherwise incurred significantly larger overhead at runtime, mainly due to the cost of managing memory and the MMU bits [28].

To estimate the benefits from using SMC, we compare the time taken to perform a single write when checkpointing with the different incremental checkpointing strategies we described above (see lines 1,3, and 6 in Table 3.1). Under (accurate) SMC, the page would just be copied once correctly placed in the writable pages hot set, and the write would complete normally. When using COW, the kernel would make a copy of the page, before the write completes. Finally, with soft dirty bits, the write completes normally but we then need to read the dirty bits to identify the updated page and save it. The process takes 492, 4016, and 16136 CPU cycles respectively. Note that in practice there are other costs involved with these strategies as well, like calculating the hot pages, marking all pages as COW in the beginning, and clearing the dirty bits (Table 3.1, line 6).

We notice that managing soft dirty bits can be very expensive, and it is preferable to use a page's checksum to identify updated pages, when we are examining a small number of pages. Most importantly, the *direct* cost of saving a page when checkpointing is only a small part of the whole process, which involves many *indirect* costs, like fault handling, managing dirty bits, etc. As a result, a perfectly accurate speculation algorithm incurs eight times less overhead per-page, compared to COW ($\frac{line1}{line3}$ of Table 3.1). We also establish that undercopying and overcopying errors do not cost the same, as the first will result in a COW (approx. 4016 cycles), while the latter leads to a wasted copy (approx. 492 cycles). Thus, on modern architectures, the cost for 1 undercopying error is comparable to 8 overcopying errors.

Finally, a "copy old pages" approach is more favorable because it requires less memory space for each checkpoint (no full snapshot). Other than guiding the design and implementation of SMC, we later use these findings to derive the cost factors for our genetically-inspired *GSpec* WSE strategy.

3.4 Framework Overview

Figure 3.1 depicts the high-level architecture of our SMC framework. To deploy SMC, users install a small kernel module (`ksmc`) and link their programs against a user-level library (`libsmc`). The library offers convenient memory checkpoint/restore primitives to programs and forwards all their invocations to `ksmc` through a fast and dedicated `SMCall` interface that requires no recompilation or restart of the running operating system kernel. Our kernel module can handle requests from a large number of programs in parallel and be safely unloaded when no longer needed, which ensures a fast and safe deployment of SMC. Also note that programs not using speculative checkpointing functionalities are unaffected by the presence of `ksmc`.

When a user program issues a memory *checkpoint* request via `libsmc`, our kernel module checkpoints the current memory image of the calling process and returns control to user space. This event marks the beginning of a new checkpointing interval, terminated only by the next checkpoint (or restore) request. The data (and metadata) associated with every checkpoint is maintained in an in-kernel journal by the core *checkpointing component* (*CKPT*) of `ksmc`. The journal stores a maximum predetermined number of K checkpoints on a per-process basis, following a FIFO replacement strategy—currently $K = 1$ by default, a common assumption in traditional memory checkpointing applications [144; 152; 53; 83; 117; 125; 51; 68; 115]. When necessary, user programs can issue a memory *restore* request and allow `ksmc` to automatically revert the current memory image to the last checkpoint k, with $k \in [1; K]$.

To speculatively copy frequently accessed memory pages and reduce memory tracing costs, the checkpointing component relies on the *speculation component* (*SPEC*), which maintains fresh writable working set estimates to drive SMC's speculative copying strategy. In particular, at the beginning of every checkpointing interval, the speculation component informs the checkpointing component of all the *hot* memory pages that should be *eagerly* copied before returning control to user space. A copy of these pages is immediately stored in the current checkpoint, eliminating the need for explicit memory tracing mechanisms in the forthcoming checkpointing interval. All the other (*cold*) memory pages, in turn, are explicitly tracked and their data copied *lazily* at first modification.

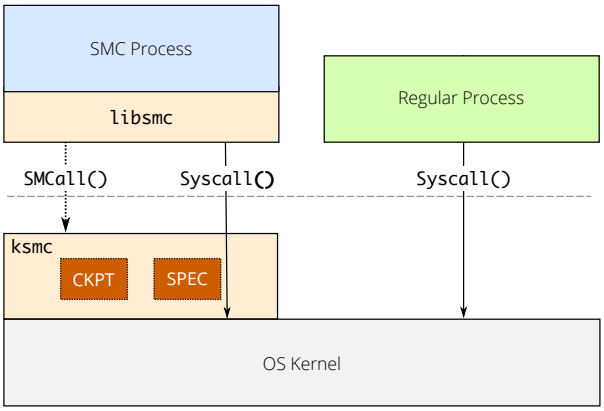


Figure 3.1: High-level architecture of SMC

3.4.1 Checkpointing Component

The checkpointing component implements the core memory checkpointing functionalities in the `ksmc` kernel module. Its operations and interface are deliberately decoupled from the main kernel as much as possible. Its internal structure is fully event-driven with a number of well-defined entry points. The main entry point provides user programs with access to a simple control interface via the `libsmc` library. Each user process can register itself with the checkpointing component—that is enter "*SMC mode*"—and specify the desired SMC configuration, including the speculation strategy to adopt and the memory regions to checkpoint. By default, the entire memory image is considered for checkpointing, but user programs may limit checkpointing operations to specific memory areas—for example, to implement an SMC-managed heap for a specialized memory allocator that supports application-level backtracking. The control interface also allows primitives to checkpoint/restore the predetermined memory areas or reset/-collect SMC statistics—for example, average number of pages copied eagerly/lazily per checkpointing interval.

For each process in SMC mode, `ksmc` maintains a process descriptor—with process-specific configurations—a set of memory area descriptors, and a journal of checkpoint descriptors. Each checkpoint descriptor maintains a number of page entries with the address and a copy of the original page to restore the saved memory image starting from the next checkpoint in the journal—or

the current memory image in case of the most recent checkpoint.

When a process enters SMC mode, `ksmc` creates new process and memory area descriptors as well as an implicit first checkpoint using a full-coverage memory tracing strategy akin to incremental checkpointing. This is done by write-protecting the page table entries associated with all the memory pages in the virtual address space of the calling process and intercepting all related page faults to save a copy of the soon-to-be modified pages.

Page fault events represent the second important entry point in `ksmc`, allowing SMC's memory tracing strategy to create new page entries in the current checkpoint descriptor, notify the speculation component of the event, and allow the kernel to simply copy and unprotect the faulting page and resume user execution. To avoid slowing down the normal execution of the main kernel's page fault handler, `ksmc` supports efficient lookups of process and memory area descriptors to quickly return control to the main kernel if the last faulting page is not currently being tracked by SMC. A similar strategy is used when intercepting process termination events—the third entry point in `ksmc`—which the checkpointing component tracks to automatically garbage collect all the descriptors and page entries associated with each terminating SMC process.

When a new checkpoint operation is requested, `ksmc` marks the current checkpoint descriptor as completed—note that this is always possible even at the first application-requested checkpoint by construction—and creates a new checkpoint descriptor for the forthcoming checkpointing interval. It subsequently iterates over the page entries in the last checkpoint descriptor and requests the speculation component to determine the optimal copying strategy for each page. For each memory page subject to an *eager* copying strategy, `ksmc` immediately creates a new page entry in the new checkpoint descriptor. For other pages, `ksmc` write-protects the page and delegates the checkpointing operations to page fault time.

When a new restore operation is requested, `ksmc` walks the checkpoint descriptors in reverse order—starting from the current one and ending with the one requested by the user—and incrementally restores all the contained page entries. It subsequently evicts all the visited checkpoint descriptors (and associated entries) from the journal and notifies the speculation component of the event.

3.4.2 Speculation Component

The speculation component enhances the basic incremental checkpointing strategy implemented by the standalone checkpointing part with a working set estimation-driven speculative checkpointing technique at the beginning of every checkpointing interval. While currently integrated in `ksmc`, the speculation component is strictly decoupled from the checkpointing component and provides a generic speculation framework suitable for both user-level and kernel-level checkpointing solutions. The speculation component requires the external checkpointing solution to provide a number of platform-specific callbacks, including memory allocation, debugging, and configuration primitives. In SMC, our kernel module implements all the relevant callbacks suitable for kernel-level execution.

Internally, our speculation component shadows many of the data structures described in the previous subsection—descriptors and page entries—but also supports writable working set *contexts* for the benefit of the individual speculation strategies implemented in our framework. Each context stores all page entries associated with the current writable working set, which our speculation component uses to determine the memory pages subject to our *eager* copying strategy when initializing a new checkpoint descriptor. The current working set context is established at the beginning of every checkpointing interval based on user-defined policies.

Each speculation strategy has unrestricted read and write access to the current writable working set context and can register hooks to manipulate the context for all the events controlled by our checkpointing module: page fault, checkpoint, restore, etc. The most conservative speculation strategy would simply produce empty writable working sets never populated with any page entries, an approach that would effectively degrade SMC to traditional incremental checkpointing. More effective speculation strategies, including our genetic speculation and other more traditional working set estimation strategies, are discussed in the following sections.

3.5 Speculation Strategies

In the course of this work, we have considered a number of speculation strategies for SMC, drawing from classic working set tracking techniques and black box optimization algorithms. We now discuss these strategies in more detail.

3.5.1 Classic WSE Strategies

Scanning-based techniques Scanning-based WSE strategies periodically scan *all* the memory pages of a running process and determine the current writable working set from the recently modified pages. These strategies are generally too expensive for short scanning intervals—strategies involving lightweight dirty page sampling have suggested using intervals of around 30 seconds [142]—due to high costs associated with frequent reference bit manipulation. The latter also suffers from the deployability limitations evidenced in Section 3.2. These shortcomings hinder the applicability of scanning-based strategies to high-frequency SMC.

Active-list-based techniques Active-list-based techniques divide all memory pages into two lists: *active* and *inactive*. On first access, pages are put on the active list, which are considered hot, that is eagerly copied at the beginning of a new checkpoint interval. On the contrary, inactive pages are copied on demand triggering a COW event. We implemented two active-list-based techniques, *Active-RND* and *Active-CKS*, which mainly differ in their active list eviction strategy.

Active-RND depends on dynamically determining the size of the WWS through periodic sampling. *Active-RND* achieves this by write-protecting the whole address space during the sampling runs, whereas the WWS size is calculated as the running average of the number of pages accessed during these runs. Whenever the active list has reached the estimated size and a new page faults in, *Active-RND* randomly evicts a page from the list. We chose a random page replacement strategy over other well known page replacement algorithms, like FIFO or the LRU-like CLOCK algorithm [35] and its variations [25; 64], because the latter either performed significantly worse in early experiments (FIFO), or require dirty page tracking or page-table entry reference-bit manipulation.

Active-CKS relies on the observation that copying and calculating a checksum is still significantly cheaper than copying a page in COW fashion. While pages also enter the active list when first accessed, *Active-CKS* will only evict a page when its checksum did not change during the last N checkpoint intervals, with $N = 5$ (the top performer in our experiments).

Oracle The *Oracle* strategy considers all the pages that will be accessed during the next interval as *hot*. Since this strategy is directly based on knowledge of the future (and due to the lack of a time machine), SMC implements

only an optimistic approximation of this algorithm based on profiling data. For each checkpoint c, it logs the number of modified pages N_c offline and pre-copies N_c dummy pages online. While this strategy lacks correctness, it gives a good estimate of what performance improvements can be expected by SMC given an ideal speculation strategy.

3.5.2 Genetic Speculation

Our genetic speculation strategy—or *GSpec*—aims to estimate the current writable working set using a methodology inspired by genetic algorithms [96]. Such algorithms provide a widely employed blackbox optimization method for problems with a large set of possible solutions. Genetic algorithms are inherently *self-tuning* and *self-adapting*, matching the stringent accuracy and adaptivity requirements of high-frequency memory checkpointing. Inspired by biological evolution, such algorithms allow candidate solutions, also called *individuals*, to compete against each other. In our case an individual represents a set of *hot* pages, whereas the information of which pages are considered to be *hot* is encoded in the individuals' chromosomes—typically represented by a bit string. All current individuals form a population. They are periodically evaluated using a *cost function*, which measures their respective *fitness*. After each evaluation period, a new generation of the population is formed by selecting the most fit individuals (*selection*) and recombining their chromosomes (*crossover*). Over time the population's solutions are meant to converge to a minimum of the cost function.

Chromosome representation and cost function *GSpec* maintains a global list of all the memory pages currently known by the algorithm, ordered by page appearance. Each individual's chromosomes represent a set of candidate memory pages, stored in a *WWS bitmap*—a generic bit string. If a bit in the WWS bitmap is set, the corresponding page is marked as *hot*, that is, part of the writable working set—otherwise the page is considered *cold*. Whenever a memory page is marked as cold by all the individuals, the page is removed from the global page list, that is, the algorithm forgets about the page.

GSpec models its *cost function* based on the memory copying costs caused by a given individual. Each memory page copied during a checkpointing interval contributes to the total cost associated with the current individual. Memory pages copied *lazily* are weighted more to reflect the memory tracing costs associated with the COW semantics. Although weighted less, pages

copied *eagerly* are still assigned a nonzero cost, preventing *GSpec* from greedily copying all the known memory pages. The cost values are directly derived from our analysis in Table 3.1, with a value of 1 and 8 accounted for every page copied eagerly and lazily, respectively.

Speculation phase The population has a predetermined size of $N = 5$ individuals, a standard value adopted in prior work on micro-genetic algorithms to ensure an efficient and fast-converging implementation [75]. For each checkpointing interval, *GSpec* selects one individual from the population in a round-robin fashion and requests the checkpointing component to copy all the hot pages eagerly. The costs for the eagerly copied pages (1) are attributed to the current individual. For each page that faults in during the current interval, the respective cost (8) is assigned to the current individual. If a faulting page is currently not in *GSpec*'s global list, it is added to the WWS bitmap of the current individual with unbiased probability $p=0.5$.

Forming a new generation After every individual had its turn, *GSpec* computes a new generation of individuals to evolve the current population. Each new individual thereby inherits the combined genetic information from selected parent individuals of the current population. Common *selection strategies* adopted by traditional genetic algorithms are *tournament selection* [94] and *roulette wheel selection* [86].

Both strategies select two parent individuals P_1 and P_2 to generate each individual in the new generation. *GSpec* implements a roulette wheel selection strategy, which yields a simpler implementation and is known to accurately model many real-world problems [49]. This strategy stochastically selects individuals with a higher probability for lower cost values. *GSpec*, achieves that by keeping track of the lowest cost C_{min} in the population and selecting a random individual I_R with a cost C_R as parent with a probability of $p = C_{min}/C_R$. This process is repeated until two parents are assigned to each individual of the new generation.

Once the parent individuals for the next generation have been selected, *GSpec* mixes the writable working sets of each parent pair P_1 and P_2 to generate each new individual. This operation is commonly referred to as *crossover*, with two dominant strategies used in the literature: *n-point crossover* and *uniform crossover* [96].

GSpec opts for a uniform crossover strategy, which generally yields an unbiased and more efficient exploration of the search space in practice [127]. This strategy selects each chromosome bit from P_1 (instead of P_2) with a

predetermined probability p. *GSpec* selects the individual chromosome bits with the standard probability $p = 0.5$ commonly adopted in prior work in the area [127].

To avoid local minima, genetic algorithms occasionally *mutate* the recombined chromosomes after the crossover phase. *GSpec* implements a simple bit-flip mutation strategy, flipping the individual chromosome bits with a predetermined probability p. In the current implementation, *GSpec* opts for a bit-flip mutation probability $p = 0.01$, again, a value commonly adopted in the literature [96].

3.6 Implementation

We implemented SMC in an architecture-independent loadable kernel module for the Linux kernel. Our implementation initially targeted Linux 3.2, comprising a total of 2227 LOC [1] for the *checkpointing component* and 1466 LOC for the *speculation component*—implementing our genetic speculation strategy and the alternatives (*Active-RND*, *Active-CKS*, and *Oracle*) considered in the paper. We subsequently tracked all the mainline Linux kernel changes until the recent 3.19 kernel release and, despite the fast-paced evolution of the Linux kernel interfaces, we added a total of only 20 extra LOC to our original implementation. This acknowledges our efforts into decoupling SMC from the mainline kernel, relying on a minimal and stable set of kernel APIs—currently a total of 45 common kernel routines for memory allocation, page table manipulation, interfacing, and synchronization.

Driven by the same principles, we implemented SMC's page fault interception mechanism using *kernel probes* [18], the standard Linux kernel instrumentation facility which allows modules to dynamically break into any kernel routine—`handle_mm_fault`, for our purposes—in a safe and nondisruptive fashion. We adopted the same mechanism to intercept process termination events—the `do_exit` and `do_execve` kernel routines—and automatically perform all the necessary process-specific cleanup operations. To implement SMC's dedicated `SMCall` interface, in turn, we allowed our kernel module to export a new kernel parameter accessible via the `sysctl` system call from user space. Our user-level `libsmc` library—implemented in one header file of 114 LOC—hides the internals of the `sysctl`-based communication protocol with the kernel module to user programs.

[1] Source lines of code reported by David A. Wheeler's SLOCCount.

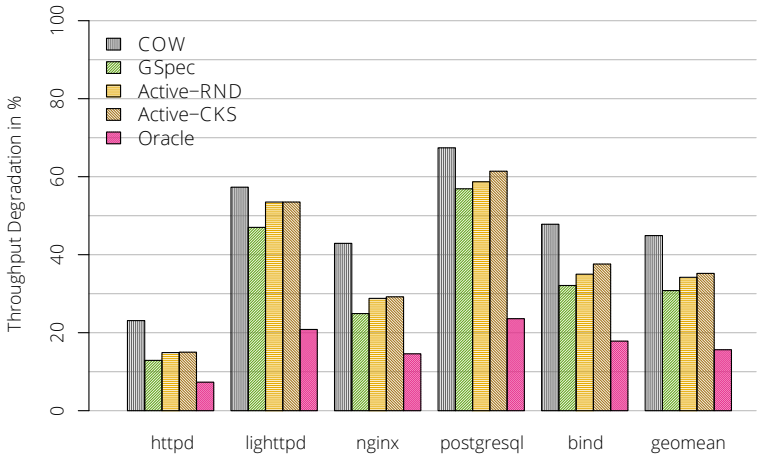


Figure 3.2: Throughput degradation induced by different SMC speculation strategies (program only).

To support common request-oriented recovery models with minimal user effort [53; 83], SMC is also equipped with a profiler that automatically identifies suitable checkpointing locations at the top of long-running request loops and a transformation module that subsequently prepares the program for speculative memory checkpointing using the identified locations. The profiler and the transformation module rely on link-time instrumentation implemented using the LLVM compiler framework [79], for a total of 728 LOC. The profiling instrumentation tracks all the loops in the program for the benefit of our profiler—coped with an interposition library to track all the processes in the target program—implemented in 3,476 LOC. The latter allows the user to instrument the target program, run it using a given test workload, and receive a complete report on all the process classes identified in the program and their long-running request loops—loops that never terminate during the test workload. The report is subsequently used by the transformation module to produce the final SMC-ready binary.

3.7 Evaluation

We evaluated our SMC framework implementation on a workstation running Linux v3.12.36 (x64) and equipped with a dual-core Intel Pentium G6950 2.80 GHz processor and 16 GB RAM. To evaluate the real-world impact of SMC, we selected five popular server programs—a common target for memory checkpointing applications in prior work in the area [152; 117; 125; 51; 115]—and allowed our deployed SMC framework to checkpoint the memory image of their worker processes at *every* client request, following the common request-oriented checkpointing model [53; 83]. For our analysis, we considered the three most popular open-source web servers—Apache httpd (version 2.2.23), nginx (version 0.8.54), and lighttpd (version 1.4.28)—a popular RDBMS server—PostgreSQL (v9.0.10)—and a widely used DNS server—BIND (version 9.9.3). To evaluate the impact of SMC on our server programs, we performed tests using the Apache benchmark (AB) [12] (web server programs), the Sysbench benchmark [5] (PostgreSQL), and the queryperf tool [10] (BIND). To investigate the SMC-induced performance impact in memory-intensive application scenarios and its sensitivity to the checkpointing frequency, we further evaluated our solution on `hmmer`, a popular scientific benchmark. Finally, in order to directly compare SMC with recent instrumentation-based memory checkpointing techniques [140] that naturally do not cover uninstrumented shared libraries, we focus our evaluation on a program-only analysis and briefly report on the performance impact of shared libraries when extending the checkpointing surface to the entire address space.

To prepare our test programs for request-oriented memory checkpointing, we allowed our dynamic profiler to automatically identify all the long-running request loops in preliminary test runs and instrument the top of each loop with a *checkpoint* call into the `libsmc` library. We configured all of our test programs with their default settings and instructed the Apache httpd web server to serve requests with the prefork module with 10 parallel worker processes. We repeated all our experiments 11 times (with negligible variations) for each of the speculation strategies presented in Section 3.5 and report the median.

Our evaluation focuses on five key questions: (i) *Performance*: Does SMC yield low run-time overhead in high-frequency memory checkpointing scenarios? (ii) *Checkpointing frequency impact*: How sensitive is SMC performance to the memory checkpointing frequency? (iii) *Accuracy*: What is the accuracy of our WSE-based speculation strategies? (iv) *Memory usage*: How much

Strategy	Throughput degradation
COW	59.8 %
GSpec	42.8 %
Active-RND	46.4 %
Active-CKS	48.6 %
Oracle	18.9 %

Table 3.2: Throughput degradation (geomean) induced by different SMC speculation strategies (program and shared libraries).

Server	Requests per second
Apache httpd	20,887
lighttpd	28,002
nginx	22,602
PostgreSQL	20,089
BIND	30,848

Table 3.3: Number of requests per second handled by our server programs (baseline, no checkpointing).

memory does SMC use? (iv) *Restore time*: Does SMC yield low restore time increase?

3.7.1 Performance

To evaluate the run-time performance overhead of SMC on real-world applications, we tested our server programs running in "*SMC mode*" and compared the resulting throughput against the baseline. To benchmark our web server programs, we configured the Apache benchmark to issue 25,000 requests through the loopback device, using 10 parallel connections, 10 requests per connection, and a 1KB file. To benchmark BIND, we configured the queryperf tool to issue 500,000 requests for a local resource using 20 parallel threads. To benchmark PostgreSQL, we configured the Sysbench benchmark to issue 10,000 OLTP requests using 10 parallel threads and a read/write workload. In all our experiments, we verified that our programs were fully saturated by the benchmarks.

Figure 3.2 shows the SMC-induced throughput degradation for our server programs, as observed during the execution of our macrobenchmarks. The

absolute number of requests handled by the individual servers without checkpointing can be found in Table 3.3. As expected, our speculation strategies generally yield a lower run-time performance overhead than traditional COW-style incremental checkpointing (*COW* in Figure 3.2) implemented by our checkpointing component in absence of any speculation strategy—note that our COW-based implementation is already much faster than traditional *fork*-based implementations used in much prior work. Compared to *COW*, our speculation strategies reported an average (geometric mean) overhead reduction of 9.6-14.24 percentage points (p.p.). *GSpec*, in particular, was consistently the top performer across all our server programs (14.24 p.p. average overhead reduction compared to *COW*, geometric mean). In some scenarios, the *GSpec*-reported improvements over traditional memory checkpointing are more significant—for example, 18 p.p. overhead reduction for nginx—due to higher checkpointing frequency and a more stable working set.

Active-RND is the second best-performing strategy—with an average performance overhead of 34.2% compared to *GSpec*'s 30.8% and *COW*'s 44.9% (geometric mean)—but we experienced its performance rapidly dropping as we deviated from the best-performing *RND-N* value. We found that altering *GSpec*'s core parameters from the values commonly adopted in the genetic algorithms literature, in contrast, had only marginal (if any) performance impact. Furthermore, *Active-CKS* reported the worst speculation performance, with an average overhead of 35.02% across all our server programs. Finally, the *Oracle* strategy reported, as expected, a consistently lower overhead compared to all our speculation strategies (15.63% geometric mean), providing a promising theoretical lower bound for the performance overhead of any future SMC strategy. Encouragingly, *GSpec* consistently follows the *Oracle* strategy across all our server programs and its overhead even comes relatively close to the *Oracle* for programs with a fairly stable writable working set—for example, 32.1% compared to 17.83% on BIND.

We now compare our results with recent compiler-based memory checkpointing techniques (LMC) [140]. For servers with good speculation performance, SMC performance is comparable or better than that of compiler-based techniques (e.g., *GSpec*'s 12.9% vs. LMC's 15.3% on Apache httpd). When speculation is less effective, compiler-based techniques tend to outperform SMC (e.g., *GSpec*'s 56.9% vs. LMC's 32.2% on PostgreSQL). On average, SMC induces an extra performance impact of 10-15 p.p. across programs. Nevertheless, we found our results very encouraging, given that unlike compiler-based techniques, SMC's checkpointing strategy is source

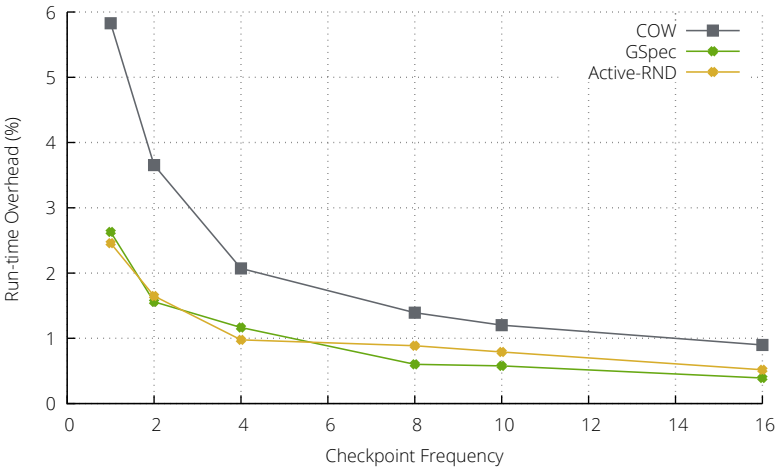


Figure 3.3: Run-time overhead induced by the different SMC speculation strategies on `hmmer`.

code-agnostic and can thus operate on legacy binaries.

Finally, Table 3.2 shows that, when extending the checkpointing surface to the entire address space, we observed an additional performance impact (due to shared library checkpointing) in the range of 12-15 p.p. We also note that the general trend is consistent and speculation equally effective, e.g., 17 p.p. average performance improvement with *GSpec*.

3.7.2 Checkpointing Frequency Impact

In the previous subsection, we investigated the SMC-induced performance impact on server request-oriented memory checkpointing, a scenario which, in our experiments, yielded a checkpointing frequency of 9K-26K checkpoints/sec across all our server programs.

To investigate the frequency impact, we evaluated our best-performing (*GSpec* and *Active-RND*) speculation strategies on `hmmer`, a memory-intensive scientific benchmark. For our purposes, we instrumented `hmmer` to invoke the *checkpoint* call into the `libsmc` library at each task loop iteration, and forced our library to forward the calls to `ksmc` only every F predetermined invocations. This allowed us to emulate different checkpointing frequencies, ranging from roughly 700 checkpoints/sec—when checkpointing at every

	GSpec				Active-RND			
	OP	**UP**	**MP**	**WMP**	**OP**	**UP**	**MP**	**WMP**
Apache httpd	7.9	1.5	9.4	19.7	6.1	2.9	9.0	29.5
nginx	10.1	0.6	10.8	6.8	2.5	0.8	3.3	8.7
lighttpd	22.7	3.2	25.9	48.3	10.5	6.1	16.6	59.6
PostgreSQL	30.0	4.7	34.8	68.0	19.2	6.3	25.6	70.4
BIND	2.4	0.9	3.3	9.8	3.3	0.6	3.9	7.8
geomean	10.5	1.6	12.5	21.2	6.3	2.2	8.7	24.3

	Active-CKS			
	OP	**UP**	**MP**	**WMP**
Apache httpd	9.4	0.4	9.9	26.9
nginx	2.1	0.3	2.4	7.8
lighttpd	19.1	2.1	21.2	64.5
PostgreSQL	29.7	0.9	30.6	81.6
BIND	2.9	0.4	3.3	10.2
geomean	8.0	0.6	8.7	25.7

Table 3.4: Accuracy of the different SMC speculation strategies, with the average numbers of overcopied pages (OP), undercopied pages (UP), mispredicted pages (MP), and weighted mispredicted pages (WMP).

iteration—to 40 checkpoints/sec—when checkpointing every 16 iterations.

Figure 3.3 depicts the SMC-induced run-time overhead on `hmmer` across all the checkpoint frequencies considered. The results shown in the figure provide a number of interesting insights. First, checkpointing at every loop iteration yields comparable results to our performance experiments on servers program, with *GSpec* (2.6%) and *Active-RND* (2.4%) improving over *COW* (5.8%). Finally, as expected, for lower memory checkpointing frequencies, the memory tracing costs incurred by traditional *COW* become more amortized throughout the execution and the performance benefits of SMC become less evident—e.g., less than 1 p.p. overhead reduction when checkpointing every 16 iterations.

3.7.3 Accuracy

To evaluate the accuracy of our speculation strategies, we implemented support for a *"meta speculation"* strategy in SMC. The meta speculation strat-

egy relies only on standard COW-style incremental checkpointing, but also transparently exposes each observed page-fault event *only* to the other speculation strategies that have assumed the faulting page not to be in the current writable working set. This allows all strategies to operate normally, while the meta speculation strategy gathers accuracy statistics based on the number of memory pages dirtied by the running program.

Table 3.4 reports the accuracy statistics produced by the meta speculation strategy when analyzing our server programs. Statistics are gathered on a per-checkpoint interval basis during the execution of our macrobenchmarks and averaged using the mean. The number of mispredicted pages (MP), that is, the sum of overcopied pages (OP) and undercopied pages (UP), represents the total number of dirty memory pages that a given speculation strategy failed to predict according to its internal writable working set estimates. The weighted mispredicted pages (WMP), in turn, weigh undercopied pages—inducing COW events—more than overcopied pages, also taking into account the additional costs for computing the checksums in *Active-CKS*. WMP is computed as $WMP = C_{OC} * N_{OC} + C_{UC} * N_{UC}$, with N_{OC}/N_{UC} and C_{OC}/C_{UC} being the number and cost factor of overcopied/undercopied pages (respectively). Based on the numbers in Table 3.1, we assume $C_{OC} = 1$ for *GSpec* and *Active-RND*, and $C_{OC} = 2.5$ for *Active-CKS*. We also assume $C_{UC} = 8$ for all our strategies.

The number of unweighted mispredictions (MP) alone seems to suggest that *Active-CKS*, with 8.7 mispredicted pages on average, is together with *Active-RND* the best speculation strategy. However, its high accuracy is overshadowed by the checksumming costs ($WMP = 25.7$), especially as *Active-CKS* tends to overcopy ($OP = 8.0$) nearly as much as *GSpec* (10.5).

GSpec, in turn, reported 21.2 weighted mispredicted pages on average, outperforming the runner-up *Active-RND*—that is, 24.3 WMP on average—with a similarly efficient working set estimation implementation. This result acknowledges the effectiveness of *GSpec*'s cost-driven speculation strategy empowered by genetic algorithms compared to the random strategy provided by *Active-RND*. This is also reflected in the lower WMP values reported by *GSpec* across all our server programs.

Overall, we can observe that the WMP predicts the performance results of the respective speculation mechanisms well and further shows the importance of carefully balancing accuracy and efficiency of the underlying working set estimation algorithm when designing a speculation strategy.

3.7.4 Memory Usage

As checkpoints also include overcopied pages, the accuracy of a speculation strategy has a direct impact on checkpoint size and overall memory usage. In our experiments, we observed our speculation strategies introducing an average checkpoint size increase compared to COW of 44%-66% across all our server programs (geometric mean). Programs with a larger writable working set—for example, PostgreSQL—or more diverse memory access patterns across checkpointing intervals—for example, lighttpd—yield the highest checkpoint size compared to traditional incremental checkpointing across all our speculation strategies, with a maximum increase of 133% and 107% (respectively). Programs with more rigorous memory usage, in turn—that is, Apache httpd and BIND—yield a more limited amount of overcopying, with a maximum increase of only 19% and 12% across all our speculation strategies. *GSpec*'s checkpoint size increases are comparable to the other speculation strategies, only occasionally yielding higher increases that reflect a more aggressive overcopying strategy—for example, for Apache httpd. Even nontrivial increases in checkpoint sizes (e.g., 133% for PostgreSQL), however, do not typically result in significant increases in physical memory usage overhead compared to *COW*. To quantify the latter, we computed the average overhead induced by memory checkpointing on the *Resident Set Size* (RSS).

Using *COW*, we reported a worst-case RSS overhead induced by memory checkpointing of only 3.6% (lighttpd). The same scenario resulted in a maximum RSS overhead of 7.6% across all our speculation strategies. This, thereby, translates to a maximum RSS increase of only 4 p.p. induced by SMC.

3.7.5 Restore Time

Overcopying errors introduce an excessive number of pages in a checkpoint, thus also increasing the restore time. For the program with the largest checkpoint size increase (PostgreSQL with 30 pages for *GSpec*) and the second largest average checkpoint size (28 pages), overcopying results in roughly doubling the number of pages to be restored (58 pages). The worst-case relative increase across our server programs is, thus, small, with only 558 extra CPU cycles required to restore 58 pages (2840 cycles) instead of 28 pages (2282 cycles)—measured using a synthetic microbenchmark. As the total time is still small and restore operations are generally much less frequent than checkpoint operations (e.g., at error recovery time), we believe this ad-

ditional cost to be negligible in practice.

3.8 Related Work

Incremental checkpointing techniques Several incremental checkpointing variations and applications are described in literature, with implementations at the user level [42; 118; 13; 119; 51; 128; 112; 117], kernel level [52; 125; 77; 56; 136; 76; 2; 106], or virtual machine monitor level [143; 72; 108; 33]. User-level techniques can be easier to deploy, but incur significant runtime overhead because memory management at the application-level is more costly than from within the kernel [28]. Other user-level approaches, rely on compiler-based program instrumentation [32; 84; 38; 159; 83; 139; 140], which require source-code and recompilation of the target programs and all used libraries. Using dynamic instrumentation at the binary level [115; 152] can provide checkpointing for unmodified binaries but incurs even higher performance overheads. Finally, approaches that require hardware support are not practical on commodity systems [44]. For this reason, SMC adopts a kernel-only checkpointing strategy implemented in a small kernel module, allowing for easier deployment compared to prior kernel-level work relying on dedicated kernel patches [52; 76; 2] or complex modules implementing fully-blown memory containers [106; 125]. Furthermore, in stark contrast to SMC, these techniques make no attempt to eliminate direct and indirect memory tracing costs in high-frequency memory checkpointing scenarios.

Checkpointing optimizations A common trend in prior work is to explore strategies to reduce the amount of checkpointed data. Some approaches propose checkpoint compression [84; 62], others rely on block-level checksumming [50; 100] to improve the granularity of incremental checkpointing techniques [136; 13; 119; 112; 50; 100], or, seek to discard redundant memory pages from the checkpointed data [108; 57; 104]. These approaches are well-suited to space-efficient process checkpointing on persistent storage, but are generally less useful to improve the memory checkpointing performance. SMC demonstrates that, in high-frequency memory checkpointing scenarios, memory overcopying can actually be beneficial to minimize the impact of indirect costs on the run-time performance.

Researchers also have explored program analysis techniques to select optimal checkpointing locations [84] or checkpointed data [38; 68; 53]. While complementary to our work, these techniques may help select checkpoint-

ing intervals with minimal working set size or provide useful heuristics to improve the accuracy of our working set estimation algorithms. We plan to explore the impact and the synergies between program analysis techniques and SMC in our future work.

Finally, other researchers have considered prediction-based strategies to improve memory checkpointing techniques. Nicolae et. al [105] propose predicting the order of memory pages modified within the next checkpointing interval to prioritize data to save on persistent storage in an asynchronous fashion. Also their prediction strategy is tailored to reducing the number of copy-on-write events—each memory page is write-protected until asynchronously flushed to persistent storage. Unlike SMC, however, their focus is on reducing copy-on-write events to minimize memory usage and their prediction strategy is only effective in asynchronous checkpointing scenarios. Other researchers have proposed combining copy-on-write semantics with dirty page tracking—using dirty bits [143] or memory diffing [89]—to predict (and precopy) the pages modified at the next checkpointing interval. Their prediction strategy, however, is limited to consecutive checkpointing intervals—which reduces the overall prediction accuracy—and relies on expensive tracking mechanisms in high-frequency checkpointing scenarios—which reduces the overall performance. SMC, in contrast, generalizes these simple prediction strategies to the writable working set model, with a larger window of observation and stronger performance-accuracy guarantees.

Working set estimation Researchers have investigated working set estimation algorithms for a broad range of application domains, ranging from garbage collection [148; 154; 58], dynamic memory balancing [37; 164; 66; 90; 142; 89], and efficient memory management in general [165], to fast program startup [67], VM migration [153], and page coloring problems [158]. To our knowledge, however, SMC represents the first application of working set estimation algorithms to the memory checkpointing domain. Prior work on working set-driven restore of checkpointed virtual machines [156; 155] comes conceptually close, but, in such context, the working set estimation is performed relatively infrequently and offline—at checkpointing time—and the information gathered only later used to efficiently prefetch data from persistent storage—at restore time. SMC, in contrast, relies on online WSE algorithms that assist and exploit synergies with high-frequency memory checkpointing techniques in real time.

Working set size estimation techniques rely either on dirty page sampling [142; 158], monitoring memory statistics exported by the operating sys-

tem [90; 154; 58; 37], or incrementally constructing LRU-based miss ratio curves (MRC) [148; 89; 164; 165; 66; 153; 163]. The latter generally provide the most accurate working set estimation method, but their most natural implementation requires expensive memory tracing mechanisms. More efficient implementations adopt an intermittent MRC tracking strategy that closely follows the phase behavior of common real-world programs [163] or rely on working set tracking to avoid tracing frequently accessed pages [148; 164; 165], typically at the cost of reduced accuracy [24].

However, traditional working set tracking techniques impose important performance and deployability limitations when applied to high-frequency memory checkpointing. Our genetically-inspired blackbox optimization algorithm, in turn, seeks to minimize the ad-hoc tuning effort generally required by prior techniques, automatically adapting the estimates to different workloads and matching the high accuracy and responsiveness required in high-frequency memory checkpointing scenarios.

3.9 Conclusion

Traditional incremental memory checkpointing is generally perceived as sufficiently fast for several typical real-world programs. In this paper, we challenged this common perception in the context of high-frequency memory checkpointing, by demonstrating that *"hidden"* costs generally deemed marginal in periodic checkpointing solutions significantly increase the run-time overhead when checkpoints are frequent. To substantiate our claims, we presented an in-depth analysis of the direct and indirect memory tracing costs associated with incremental checkpointing and uncovered limitations of prior frameworks in high-frequency checkpointing scenarios.

To address such limitations, we presented SMC, a new low-overhead technique suitable for high-frequency memory checkpointing. To minimize the direct costs associated with the checkpointing activity, our SMC framework relies on non-intrusive kernel-level specialization implemented in a loadable kernel module. In order to minimize the indirect costs associated with the checkpointing activity, our framework relies on algorithms for estimating the writable working set to copy speculatively those memory pages that are most likely to change in the next checkpointing interval, and in so doing reduce the memory tracing surface required by traditional incremental checkpointing.

We also demonstrated that our genetically-inspired blackbox optimization algorithm (GSpec) provides an effective working set estimation strategy for

SMC, continuously adapting the working set to the workload driven by only program-agnostic cost factors. This strategy provides better accuracy, performance, and self-tuning guarantees than traditional working set estimation techniques. Overall, our experimental results show that SMC is both time- and space-efficient in the practical cases of interest, demonstrating that low-overhead high-frequency memory checkpointing is a practical option and opening up opportunities for new programming abstractions empowered by fast checkpointing techniques.

CHAPTER 4

PEEKING INTO THE PAST: EFFICIENT CHECKPOINT-ASSISTED TIME-TRAVELING DEBUGGING

4.1 Introduction

Debugging sessions can be time-consuming, frustrating and expensive, especially for long-lived latent software bugs that require a long time to manifest themselves. Such latent bugs often result from particular *events* in the input. For example, consider a server receiving hundreds of thousands of requests, for which a single bad request corrupts its state. Unless the corruption leads to an immediate crash, rather than a failure or a misbehavior at a later time, it becomes hard to track down the exact request that caused it. We present a new solution that lets programmers efficiently create high-frequency checkpoints, store *millions* of them in memory, and efficiently search through them to look for the root cause of such event-driven latent bugs.

Our work fits under the general umbrella of targeted debugging solutions

for "hard" bugs that also includes recent work on reproducing rare concurrency bugs [71; 150]. In fact, from the wide range of approaches based on record/replay [59; 72; 109; 22; 27; 137] and execution synthesis [150], to the recently proposed failure sketches [71], most existing solutions focus solely on buggy data races. In contrast, we aim for the *other* major category of hard bugs: latent corruption that only manifests itself millions of events later.

Unfortunately, even the most advanced record/replay debuggers [128; 137; 78; 15] are a poor match for such bugs. In theory, it is simply a matter of replaying a recorded trace, using either forward or backward execution, with additional breakpoints and watchpoints, until we find the target condition, but in practice this is inefficient and cumbersome.

First, despite the progress in recent years in reducing the overhead [78; 137], record/replay systems are still *impractical.* This is true not just for the recording side (a popular optimization target for these systems), but especially for the replay side which is equally important in reducing the time to track down elusive bugs. Second, pure record/replay systems are *linear* by nature. Finding a bug typically requires at least one linear execution of a trace. In contrast, for many classes of bugs, we can examine a collection of snapshots much more efficiently, e.g., using binary search. Third, record/replay systems are mostly useful for understanding the problem once the trigger event is known, but offer limited help in *identifying* the trigger (e.g., by querying the history for specific events). Finally, and very pragmatically, the fastest record/replay systems today require *extensive kernel modifications*, making them hard to deploy (compared to, say, a self-contained kernel module).

In contrast, we believe that high-frequency memory checkpointing [41; 88; 143; 140; 139] offers an alternative and complementary solution that couples low run-time overhead with a searchable trail of snapshots of the program state. Of course, this is only possible if we can make both the recording and the querying of the history sufficiently lightweight.

Memory checkpointing has already proven useful in the domains of error recovery [51; 53; 54; 68; 83; 115; 116; 119; 125; 135; 152; 143], record/replay debugging [61; 128; 72; 137; 78], application-level backtracking [33; 159], and periodic memory rejuvenation [144], but these solutions are all too heavyweight for the high-frequency checkpoints we need for debugging latent software bugs. We require checkpointing at high speed and fine granularity, say for every incoming request of a loaded server program (to record all the potential inputs of interest). Moreover, to address very long-lived latent bugs, the checkpoint history should be as long as possible, which poses the challenge of storing memory checkpoints efficiently. Finally, as the num-

ber of checkpoints can be high (potentially millions), this vast amount of data has to be processed by the user in a manageable manner.

Based on these observations, we propose *DeLorean*, a new end-to-end solution for time-travelling debugging of event-driven latent bugs by means of high-frequency checkpointing. The idea consists in taking frequent checkpoints for all the events of interest and then quickly query the collected data for a given condition (e.g., using binary search). To implement both phases efficiently, *DeLorean* does not support full record/replay functionalities but instead focuses on automating and speeding up the process of diagnosing and determining the root-cause of event-driven latent software bugs. To maintain a sufficiently long checkpoint history, *DeLorean* employs memory *deduplication* and/or *compression*, increasing the history size up to 10 times compared to plain checkpointing. Moreover, *DeLorean* is fully integrated into `gdb` offering effective ways to introspect the entire checkpoint history efficiently. Finally, *DeLorean* is easy to deploy, since we implemented its core functionalities in a self-contained Linux kernel module.

Although we present fast checkpointing mainly as a stand-alone debugging solution, we emphasize that our work is complementary to many existing debugging approaches. For instance, existing record/replay solutions often also use checkpointing to implement reverse debugging. However, as the checkpointing in these systems is expensive, they can take snapshots only very infrequently—up to once per 10-25 seconds in some cases [72]. Our high-frequency checkpointing strategy brings this number down to milliseconds.

Contribution. Our contribution is threefold.

(a) We present *DeLorean*, a debugging solution prototype based on high-frequency memory checkpointing. *DeLorean* targets event-driven latent bugs and combines excellent run-time performance with an efficient memory footprint (allowing for millions of checkpoints).

(b) We propose a new mechanism to perform fast queries on the collected checkpoints called *time-travelling introspection*. With such mechanism in place, we show that checkpoint-assisted debugging is a viable option to diagnose event-driven latent software bugs.

(c) We evaluate *DeLorean* effectiveness, performance, and memory usage guarantees and compare our results to state-of-the-art debugging solutions.

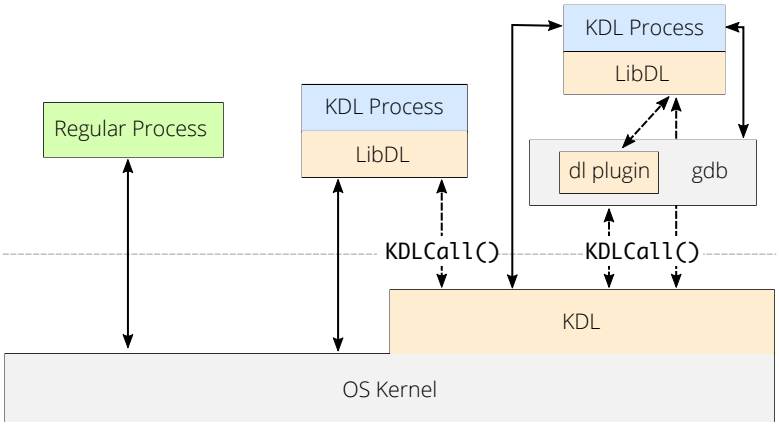


Figure 4.1: Architecture overview of *DeLorean*, with both its user- and kernel-level components and their interactions.

4.2 Architecture Overview

Figure 4.1 depicts the high-level system architecture of *DeLorean*. *DeLorean* consists of *KDL*, the kernel module, *LibDL*, the shared library, and *dl*, the `gdb` plugin. This simple architecture makes *DeLorean* easy to deploy, without the need to recompile the kernel or the target application.

The *KDL* kernel module enables *DeLorean* checkpointing features and offers them to the userland through the *KDLCall* interface. *KDL* can handle multiple checkpointed processes and keeps all the checkpointed data in memory. Further, it offers several configuration parameters, e.g., the maximum number M of most recent checkpoints in its history, and incorporates several mechanisms to reduce the amount of checkpointed data. While *KDL* is part of the *DeLorean* debugging framework, the exported checkpointing features are generally applicable to other use cases that require support for time- and space-efficient memory checkpointing.

The *LibDL* user library unburdens debugged applications to directly interact with the *KDLCall* interface, our API to the kernel module. This library can be preloaded in the target application, as done by *dl*, but can also be directly linked to applications so they can use *KDL* checkpointing facilities for other use cases, such as backtracking [33; 155] and error recovery [51; 53;

54; 68; 83; 119].

We implemented the *dl* debugger as a generic `gdb` plugin that complements all the standard debugging functionalities already available in `gdb` with support for checkpoint-assisted debugging. This is possible by exporting custom commands in `gdb`, as detailed later in Section 4.3.

To register an application with *DeLorean*, users simply start a `gdb` session with our *dl* plugin, which automatically sets up all the other *DeLorean* components. Note that running processes not being debugged by *DeLorean* are unaffected by the operation. By default, *DeLorean* monitors the entire address space of the targeted process. However, *DeLorean* allows users to limit the checkpoint surface to specific portions of the address space during the initialization (and configuration) phase. Each process in the application will be registered with *DeLorean* until it exits or is explicitly unregistered by the user.

Once an application is registered with *DeLorean*, the system provides users with a number of features:

(a) take a new checkpoint,

(b) rollback to a previous checkpoint,

(c) restore the current checkpoint to continue execution,

(d) and *query the checkpointed history*.

While the first three operations also feature in traditional checkpointing systems, querying the checkpointed history is a new feature offered by *DeLorean*. It allows the user to specify complex search queries and efficiently evaluate the given conditions through the checkpoint history. This is similar, in spirit, to temporal queries supported in modern databases with versioning support. Moreover, *DeLorean* may perform condition evaluation through the checkpointed history either *linearly* (i.e., through a linear search) or using a more efficient *bisect* approach (i.e., using a binary search, not unlike `git bisect` [17])—provided that the continuity of the condition throughout the collected history allows for it. This is typically the case for the long-lived effects caused by the event-driven latent bugs targeted by *DeLorean* (e.g., memory corruption).

Moreover, while memory checkpointing is fast, it generally incurs memory overhead which can grow significantly with the number of checkpoints in the history [139; 140; 143]. For this reason, we have implemented various modes of operation in *KDL* that allow users to tune the checkpoiting

feature. Specifically, users can enable memory *deduplication* and/or *compression* features in *KDL* to allow for the storage of a large checkpoint history but trading off run-time performance. In the following sections, we detail the design and the features of *DeLorean* debugger.

4.3 User-space Debugger

While *DeLorean* checkpointing functionalities reside in *KDL*, users do not have to interact with the kernel module directly. They only have to make sure that the kernel module is loaded, as our system exposes all the debugging functionalities through *dl*, our checkpoint-assisted time-traveling debugging tool. *dl* is implemented as a `gdb` plugin and allows to frequently checkpoint the target process while preserving checkpointed data for millions of checkpoints.

An extract of some of the commands exported by *dl* is shown in Table 4.1. The Table also contains the references for each command used throughout this section.

4.3.1 Initialization

A debugging session is initiated in the same way a user would normally start a standard `gdb` session but additionally specifying *dl* as a `gdb` plugin from the command line. The user needs to load the kernel module before starting the session. After that, *dl* preloads the *LibDL* shared library. As soon as the application starts running, *dl* puts the corresponding process into *DeLorean* mode transparently to the user. This ensures the entire process address space is by default registered with *KDL*, except for the memory regions reserved for *LibDL*. Furthermore, to rollback the stack area of the process safely, *dl* starts a dedicated worker thread owned by the target process. The worker thread owns a private memory area used for its stack and metadata, which is also excluded from the checkpointing surface. The initialization starts an implicit initial *checkpoint interval* (i.e., the execution interval between two consecutive checkpoints) by write-protecting all registered memory pages. This is necessary to checkpoint the target pages efficiently and incrementally, as detailed later in Section 4.4.

ref.	checkpointing
[a]	`dl cp take [LOCATION [if <CONDITION>]]` Starts a new checkpointing interval. When a location is specified, a breakpoint is set that triggers the new interval. If a condition is given, the interval starts only when the condition is satisfied.
[b]	`dl rb <CP ID>` Rolls back the memory state of the process to the checkpoint with the specified ID.
[c]	`dl restore` Restores the memory state to the "present" state.
	time-travelling introspection
[d]	`dl for <CP SPEC> if <CONDITION> [do <CMD>]` For all the specified checkpoints, *dl* rolls back and evaluates the provided condition. If a `do-statement` is specified, the list of commands are executed when the condition is satisfied.
[e]	`dl for <CP SPEC> do <CMD>` For all the specified checkpoints, *dl* rolls back and executes the provided commands.
[f]	`dl search <CONDITION>` Searches for the furthest checkpoint in time which satisfies the condition. The command uses either a linear or a binary search. If such a checkpoint exists, the state is ultimately rolled back to the selected checkpoint.

Table 4.1: Subset of *dl* commands. `CP ID` represents the numeric identifier of a checkpoint interval. `CP SPEC` indicates a set of checkpoints specified as arrays of IDs and/or intervals.

4.3.2 Checkpoint

The main features of *dl* are the capability to take checkpoints, rollback the memory to an older state and restore for further execution (Table 4.1 show an extract of the commands).

While the user can take new checkpoints explicitly (starting new checkpoint intervals) by issuing [a] without arguments, this is of limited use by itself, as it requires the user to interrupt the process. A more interesting option for our target domain is to checkpoint the memory image of the target process automatically when particular events occur during execution (e.g., a new request received by a server program). The user can associate *checkpoint requests* to the program events of interest by additionally supplying a location and, possibly, a condition to be evaluated. This is very similar to setting a new breakpoint in `gdb`.

We initially implemented our checkpoint request functionality on top of the `gdb` breakpoint mechanism. However, we observed that piggybacking

on gdb breakpoints was relatively expensive. Whenever the program hits a breakpoint associated to the checkpoint request, it first transfers CPU to gdb which, in turn, hands over control to *LibDL* to perform the checkpoint operation, and later resumes the execution. Unfortunately, the transfer is costly. Static instrumentation, i.e., instrumenting the points of interest with direct calls into *LibDL*, was one option we considered to reduce the overhead. Efficient static instrumentation, however, would typically require recompiling the target application. For this reason, we opted instead for a design based on hardware breakpoints. We extended *KDL* adding an interface to set hardware breakpoints and automatically take checkpoints whenever a breakpoint is hit. We ran a microbenchmark to analyze the overhead of using breakpoints to issue checkpoint requests. We estimated CPU cycles by averaging 1,000 executions of each test. While a checkpoint request directly issued by a process (*static instrumentation*) costs approximately $14K$ cycles, a software breakpoint is more than two orders of magnitude slower (ca. $2.7M$ cycles). Checkpoints issued by *DeLorean* hardware breakpoint handler, while being somewhat more expensive (ca. $19k$ cycles), are still in the same ballpark as instrumentation-based checkpointing.

Our debugger allows the use of hardware breakpoints via a configuration option that is enabled by default. When this option is set, *DeLorean* will always try to convert a *soft-checkpoint request* into a *hard-checkpoint request*. This will only fail if the system runs out of available hardware breakpoints. Further, hardware breakpoints do not support conditional checkpoint requests other than equality checking.

4.3.3 Time-traveling Introspection

DeLorean provides `rollback` [b] and `restore` [c] commands to navigate the memory history. Travelling to a specific checkpoint is certainly useful when the number of checkpoints is relatively low. However, it is much harder to find the right target for millions of checkpoints.

To deal with a large checkpoint history, *DeLorean* provides two commands to *query* the set of previously taken checkpoints: `for` [d][e] and `search` [f]. Commands [d] and [e] linearly iterate over all the specified checkpoints and for each of them evaluate conditionals, and execute commands (either conditionally or unconditionally). The user specifies the set of checkpoints as an interval (e.g., `(0,1000)` for the last 1000 checkpoints) a list of IDs, a mix of lists and intervals, or the keyword `all`, to include all the taken checkpoints.

The `search` [f] command, in turn, locates the first checkpoint satisfying

a provided condition and rolls back to that state. The condition can be anything ranging from a simple comparison to the result of a function call. The command can be configured to either search strategies `forward`, `backward` and `bisect`. The `forward` and `backward` strategies perform a linear scan over the given checkpoint in opposite direction. Differently, the `bisect` strategy locates a checkpoint using binary search, similar to `git bisect` [17]. This strategy can dramatically reduce the duration of the search operation, although it is only applicable when the condition is met continuously from a particular point onward (e.g., from the point when `sanity_test()` returns `false`).

4.4 Kernel Support

KDL, the *DeLorean* kernel module, is the core of our debugging system, providing all the necessary features to checkpoint a user process, roll it back to inspect past memory states, and restore the "present" memory state to resume execution. An overview of the module components and their interactions is depicted in Figure 4.2.

4.4.1 Taking Checkpoints

There are three entry points in *KDL* that cause a new checkpoint interval to begin. First, whenever a process registers with *KDL*, the initialization implicitly starts a checkpoint interval, as mentioned earlier. Second, it is possible to explicitly request a checkpoint via the *KDLCall* interface. This method is used by *dl* to implement targeted checkpoints on top of the `gdb` breakpoint mechanism. Finally, the kernel module triggers checkpoints directly using hardware breakpoints.

To efficiently satisfy checkpoint requests, *KDL* employs an incremental memory checkpointing strategy, shown as the best approach for high frequency memory checkpointing [141]. At the beginning of a new checkpoint interval, *KDL* write-protects all the memory pages to checkpoint. This strategy allows *KDL* to get notified whenever any of these pages is modified during a checkpoint interval to add a copy of the checkpointed page to the *checkpoint list* (i.e., using a *copy-on-write* strategy on a write *page fault*). This list is stored in the *process context*, a per-process data structure initialized whenever a process registers with *KDL*. As depicted in Figure 4.2, each *process context* also contains a *journal* of past checkpoints. Whenever the current checkpoint

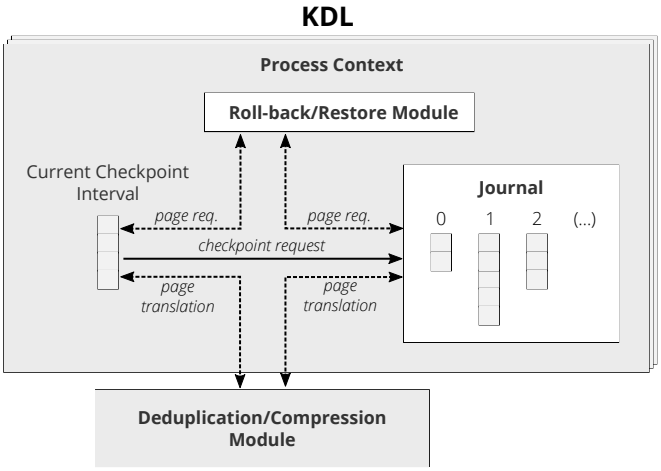


Figure 4.2: Overview of the core *KDL* kernel module components.

interval terminates (and a new interval starts), *KDL* re-protects all the dirty memory pages, i.e., pages with a copy in the current *checkpoint list*. Next, the current *checkpoint list* becomes an entry in the *journal* and is replaced by a new empty list. When the configurable journal size K is exhausted, *DeLorean* evicts the last checkpoint list from the journal to make room for new ones. Currently, our eviction strategy effectively deletes the oldest checkpointed pages, limiting the observable history.

h

4.4.2 Reducing Memory Overhead

As long-lived latent software bugs may trigger even hours after their root cause, they require a large checkpoint history to allow the user to locate the root cause of the bug. For this reason, our design seeks to retain as many checkpoints in the history as possible, trading off some run-time performance for a more space-efficient checkpoint list representation. Specifically, *KDL* supports optional *page deduplication* and *page compression* for all the checkpointed pages to reduce the memory footprint and scale to a larger number of checkpoints in the history.

Page Deduplication Given that programs, and in particular server processes, tend to frequently re-initialize long lived data structures with similar data, we can expect many checkpointed pages to be filled with the same content. Building on this intuition, *KDL* supports a page deduplication strategy that keeps the checkpoints in the history as compact as possible. Figure 4.2 shows that the deduplication module (and its tracking data structures) has global (rather than per-process) scope. This enables deduplication of pages shared across processes concurrently registered with *KDL*.

To support page deduplication, *KDL* relies on a red-black binary tree to keep the set of globally unique checkpointed pages ordered. To determine the order between any two given pages, *KDL* supports two possible strategies: (i) directly comparing the content of the pages using `memcmp` [1] or (ii) compute and compare a *checksum* based on the content of the pages. The first approach requires a `memcmp` operation between a given page and $O(log(n))$ other pages (where n is the number of pages stored in the deduplication tree). The second approach requires only $O(log(n))$ checksum comparisons and, in the best case, only a single additional `memcmp` operation, but this number increases with the number of colliding pages, for which the checksum is the same while the content is not.

Furthermore, we explored two possibilities to select the moment to deduplicate a page (i.e., merge two checkpointed pages with the same content), based on either a *greedy* or a *lazy* approach. The greedy approach deduplicates each page at copy-on-write time, which eliminates unnecessary copying of duplicated pages. The lazy approach, in turn, deduplicates all the checkpointed pages at the end of every checkpoint interval. This approach does not eliminate unnecessary copying of checkpointed pages, but has the advantage of batching deduplication operations and removing expensive deduplication tree lookups in the page fault handler at copy-on-write time.

The effectiveness of page deduplication is heavily subject to the memory usage patterns of the debugged application. A very unstable working set with random memory write patterns is unlikely to benefit from page deduplication. Luckily, real-world applications normally exhibit fairly stable working sets which benefit from checkpointed page deduplication, as the encouraging results in our evaluation demonstrate. In fact, Section 4.6 shows that greedy checksum-based approach deduplication is the best candidate for the applications we target.

[1] `https://www.kernel.org/doc/htmldocs/kernel-api/API-memcmp.html`

Page Compression To improve space-efficiency guarantees with programs exhibiting poor temporal working set similarity, *KDL* also supports a page compression strategy. While page compression is more robust than deduplication against random memory write patterns and consequently more effective in reducing the memory footprint of the checkpoint history, the compression algorithm imposes a higher run-time overhead during checkpoint and roll-back operations. Furthermore, page compression requires *KDL* to store the compressed data in memory efficiently, or fragmentation would eliminate most of the savings gained. To address this problem, *KDL* stores the compressed pages using the `zsmalloc` allocator [9], designed for optimal storage of compressed memory.

Another key challenge concerns the selection of the compression algorithm to guarantee a good tradeoff between data compression and run-time performance. Inspired by `zram` [91], *KDL* relies on the LZO algorithm and deals with special-cases (such as zero-filled pages and pages whose compressed content is larger than the original content) so to implement an effective and efficient compression strategy.

Similar to deduplication, our page compression strategy can, in principle, operate using either a *greedy* or a *lazy* approach. Our current *KDL* implementation, however, only supports a *lazy* approach, given that, due to a caveat in the implementation of `zsmalloc`, it is not possible to integrate compression support in our page-fault handling code path.

Finally, while page compression and deduplication are conceptually independent (and competing) alternatives, *KDL* can support both deduplication and compression at the same time. We evaluate all the possible deduplication/compression configurations of *KDL* in Section 4.6.

4.4.3 Rolling Back to a Checkpoint

The *KDL* rollback component is responsible for rolling back the memory state of the target process to a given checkpoint. As depicted in Figure 4.2, this component operates directly in the target process context. To implement the required functionalities, the rollback component identifies all the pages that have to be restored in order to rollback to a given checkpoint and stores them in a binary search tree—the *roll-back tree*. Furthermore, it replaces all the page mappings of that target process managed memory regions with the ones that are part of the specified checkpoint (by looking them up in the rollback tree created earlier). To protect the checkpointed pages, the new mappings are initially all read-only. Should the target process want to

write into an initially writeable page after rollback, *DeLorean* lazily creates a scratch copy of the page in question using a copy-on-write strategy. When the "current" process state is restored, *KDL* discards all the scratch pages created by any user operations while rolled-back.

For efficiency reasons, the rollback tree usually only contains entries for one specific checkpoint. *KDL* creates the tree by looking up the checkpoint list for the target checkpoint and adding all its pages to the rollback tree. Furthermore, *KDL* iterates through the younger journal entries to add all the pages referring to memory locations not already included in the tree but mapped at the time the checkpoint was taken.

Given that generating the rollback tree is expensive, *KDL* also supports the generation of a permanent and global version of the rollback tree, which includes data for all the checkpoints. The initial cost to create a permanent tree is higher, but the cost is amortized when a large number of rollback operations are requested, e.g., when the user performs a search over a large number of checkpoints.

The rollback approach described thus far is the most transparent and general possible: *KDL* reverts the *entire* memory state to a given checkpoint and the user may freely inspect the old state and implement arbitrarily complex operations over it. However, this approach may not be the most efficient if user operations over the checkpointed state are localized (e.g., a simple search over a given global variable), given that an excessive number of pages may need to be mapped (and remapped back). To address this problem, *KDL* supports an alternative. Whenever the userland is aware of the variables that need to be accessed beforehand, *KDL* supports an *on-demand* rollback mechanism. This approach only rolls back the pages that are actually required to access the target data structures. To implement this mechanism, *KDL* exports a user-level API which requires a list of memory addresses and the corresponding sizes (the targeted memory pages are identified automatically by *KDL*).

4.5 Implementation

We implemented our *DeLorean* prototype on Linux. *KDL* is implemented as a standard Linux *loadable kernel module* supporting recent kernel versions (3.2 until 3.19), allowing for easy deployability. The module is compatible with `x86` and `x86_64` architectures and supports applications already targeted by

gdb. The total module size is $5,493$ LOC[2], including all the modes of operation described in the paper.

To keep *KDL* as portable as possible, we relied on *Kernel Probes* [18] (the standard Linux instrumentation framework to non-disruptively hook into kernel routines) to implement two key features: (i) page fault handling (and copy-on-write) interposition by hooking into `handle_mm_fault` before a write page fault is handled; (ii) process exit interposition to clean up each process context by hooking into `do_exit` and `do_execve`. The *KDLCall* system call interface, in turn, is implemented using the Linux `sysctl` [6] mechanism, the standard Linux interface to configure kernel parameters at runtime. This interface is used by the userland to issue checkpoint/roll-back/restore requests, retrieve module statistics, and access configuration options.

The *dl* debugging component is implemented as a combination of a Python-based `gdb` plugin [16] and a `gdb` command file [1], accounting for a total of $1,075$ LOC. Other than implementing *dl*-specific options and commands, the scripts add hooks to allow for proper cleanup when the target process exits and to force a restore if the user resumes execution while the process is in rolled-back state. The *LibDL* shared library, finally, implemented in 417 lines of C code, is injected into the target process via the usual `LD_PRELOAD` interface.

4.6 Evaluation

We evaluated our *DeLorean* prototype, on an Ubuntu 14.04 workstation running the stock kernel Linux v3.16 (x86_64). The workstation mounts a dual-core Intel Pentium G6950 2.80GHz processor and 16GB of RAM. To evaluate the impact of our system on real-world applications, we selected five popular server programs, similarly to prior work in memory checkpointing and record/replay debugging [125; 78; 137; 128; 51; 152; 115; 116; 140]. In particular, we focused our evaluation on three popular open-source web servers, Apache `httpd` [12] (version 2.2.23), `nginx` [132] (version 0.8.54) and `lighttpd` [74] (version 1.4.28), a widely used database server, PostgreSQL [3] (version 9.0.10), and the vastly deployed `bind` DNS server [10] (version 9.9.3).

We evaluated the impact of *DeLorean* on our server programs using well-known benchmark suites: Apache benchmark (`ab`) [11] for our web servers,

[2]Source lines of code as reported by the CLOC tool.

SysBench [20] for PostgreSQL, and queryperf [7] for `bind`. To evaluate the worst-case performance with intensive CPU pressure, we have run our performance tests issuing requests through the loopback device (also a common latent bug debugging scenario), which fully saturated our test programs. In addition, we evaluated additional aspects of our system using dedicated microbenchmarks.

We configured our programs and benchmarks suites with their default options. We instrumented our programs with our *LibDL* shared library and allowed them to take a checkpoint for each request (the finest granularity to debug latent event-driven bugs). We repeated our experiments 11 times (with negligible variations) and report the median values.

Before continuing, data resulting for the pure usage of checkpoint without any memory footprint reduction method is interpolated due to the lack of RAM in our experimental settings. However, based on the linearity shown by our evaluation, we can safely compute the expected runtime and space measures in such situations.

Our evaluation answers the following questions:

(a) *Deduplication and compression performance*: How efficient are the various deduplication and compression strategies supported by *DeLorean*?

(b) *Deduplication and compression effectiveness*: How effective are the various deduplication and compression strategies supported by *DeLorean* in reducing the memory footprint and scaling to a large checkpoint history?

(c) *Time-travelling introspection performance*: How efficiently can we query the checkpointed memory pages through the history?

(d) *Comparison with existing solutions*: How do our results compare to existing debugging solutions?

(e) *Case studies*: How can we use *DeLorean* to debug real-world software bugs?

4.6.1 Deduplication Performance

Deduplicating a page at record time can be done by directly *comparing* any two given pages, or by first computing (and comparing) a *checksum* of the two pages. Both solutions incur two costs: (i) maintaining a data structure to store the unique instance of each page and (ii) actually checking for duplicate pages. The distribution of such costs depends on whether we opt of a

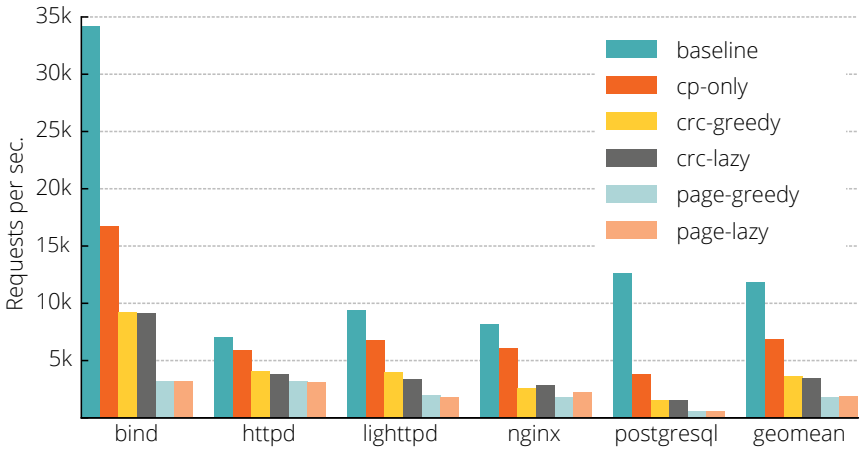


Figure 4.3: Server throughput for our dedup. strategies ($1M$ checkpoints). `cp-only` results are interpolated.

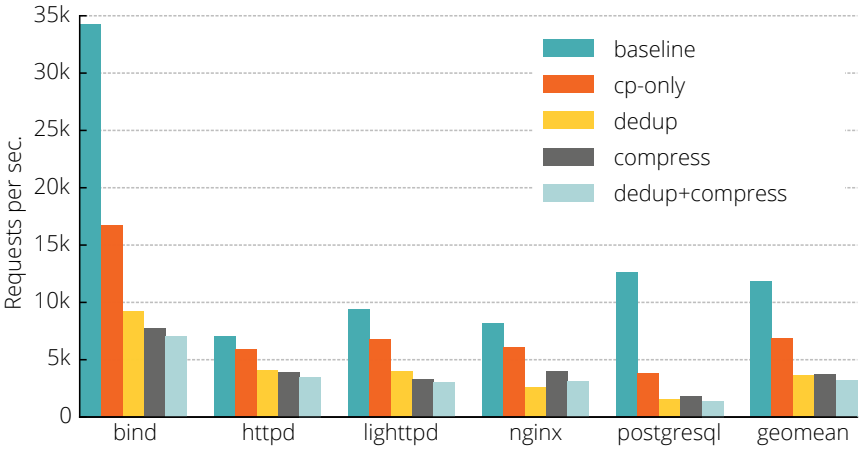


Figure 4.4: Server throughput with compression and/or dedup. ($1M$ checkpoints). `cp-only` results are interpolated.

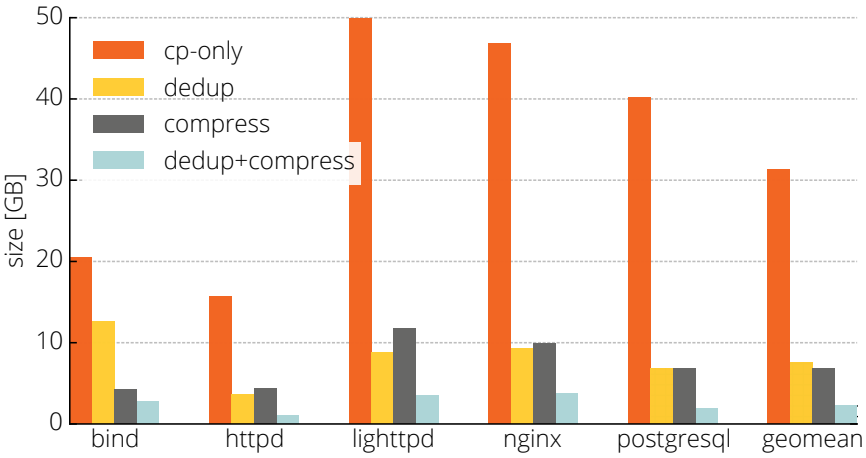


Figure 4.5: Memory footprint with compression and/or dedup. ($1M$ checkpoints). `cp-only` results are interpolated.

greedy (page fault handling-time) deduplication strategy or *lazy* (checkpoint finalization-time) deduplication strategy.

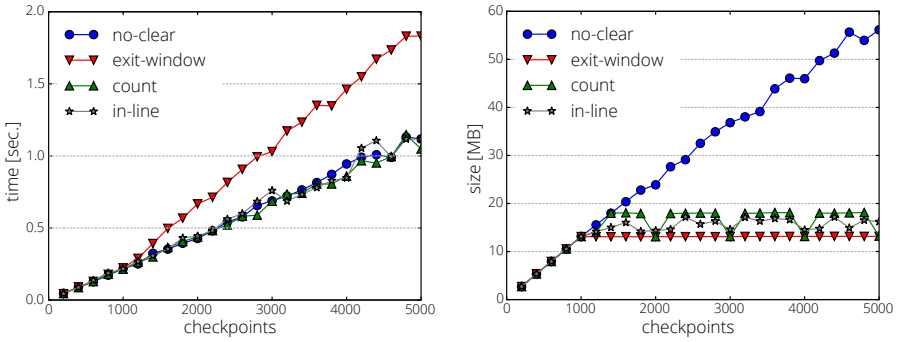
Figure 4.3 shows the throughput of the *baseline* (program running without checkpointing), plain checkpointing (*cp-only*), the comparison-based deduplication strategies (*page-greedy* and *page-lazy*), and the checksum-based deduplication strategies (*crc-greedy* and *crc-lazy*). We can see in the figure that the throughput degradation imposed by comparison-based deduplication is higher than that of checksumming across all the servers. We further observed 78.24 % and 77.76 % throughput degradation for *page-greedy* and *page-lazy*, compared to 65.83 % and 63.96 % for *crc-greedy* and *crc-lazy* (geometric mean).

This demonstrates that page comparison costs are significantly higher than checksumming costs. Furthermore, on average, the greedy approach performs slightly better than the lazy approach independently of the comparison strategy. This result is in line with page deduplication significantly reducing the number of checkpointed pages and the cost of greedily copying pages being thus amortized by the benefits of maintaining less unique page instances on tracking data structures.

This trend is reversed for the comparison-based strategy on `nginx`, which

causes *page-greedy* to produce a higher throughput degradation compared to *page-lazy*. A closer inspection revealed that, while both comparison-based strategies are less effective in deduplicating pages compared to the checksum-based strategies, *page-greedy* deduplicates even less than *page-lazy* on `nginx`. We attribute this behavior to the run-time overhead imposed by page deduplication, which slows down the server and causes it to behave less deterministically with additional data accounting. This, in turn, results in reduced deduplication effectiveness. This effect is further amplified by the *page-greedy* strategy, which introduces extra page fault-time latency throughout the execution.

While *crc-greedy* performs sliglhtly better than *crc-lazy*, we will focus on *crc-lazy* deduplication (or simply *dedup*, hereafter). Due to implementation limitations, *crc-greedy* can only be combined with compression by uncompressing all the checkpointed pages to determine duplicates, while *crc-lazy* can deduplicate based on compressed data.



(a): runtime overhead (crc-based).

(b): size overhead (crc-based).

Figure 4.6: Impact of orphan cleanup mechanisms vs. number of checkpoints (`lighttpd`).

4.6.2 Orphans Cleanup Impact

Deduplication requires maintaining a dedicated tracking data structure (the deduplication tree) and relinquishing pages that are no longer needed. In particular, when all the references to a given deduplicated page are dropped (e.g., when a particular checkpoint list is deleted), the corresponding now *orphaned* page remains in the deduplication tree. Without further intervention, the number of orphan pages would grow over time, resulting in increas-

ing performance overhead (i.e., the cost to walk the deduplication tree increases) and memory overhead (i.e., the number of tracked pages increases). To address this problem, we implemented *orphan cleanup* mechanisms that triggered at various points during the checkpointing process.

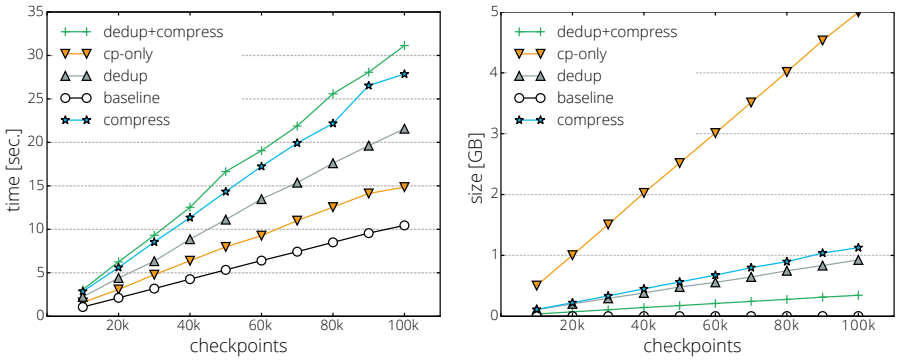
Figure 4.6 shows the effect on the benchmark run time and memory usage incurred by such mechanisms on `lighttpd` (for clarity, we omit the very similar results on the other servers). For this experiment, we set the journal size to $1,000$ checkpoints and linearly increased the number of checkpoints taken. As shown in the figure, the naive no-op (*cp-only*) mechanism rapidly increases the increases the memory footprint of the system. Cleaning up orphans every time a checkpoint is deleted (*exit-window* mechanism), in turn, does not impose additional memory overhead, but imposes a higher performance overhead. Performing orphan cleanup operations after taking a certain number of checkpoints (*count*, i.e., every $1,000$ checkpoints in our case) or at deduplication time (*in-line*) reduces the impact of orphans cleanup, while retaining reasonable memory usage guarantees. To isolate the effect of orphan deletion, the rest of our analysis assumes that the journal size coincides with the observation window.

4.6.3 Deduplication vs. Compression

While deduplication is an effective solution to reduce memory usage at record time in the common case, compression represents a more general (but slower) alternative. *DeLorean* can also support both deduplication and compression to minimize the overall memory footprint. Figure 4.7 illustrates the impact of different deduplication/compression configurations for an increasing number of checkpoints on `lighttpd`. As expected, both the benchmark run time (Figure 4.7a) and memory usage (Figure 4.7b) increase linearly with the number of checkpoints taken. We provide a detailed comparison in the following.

Memory Overhead To measure the effectiveness of deduplication (*dedup*) and compression (*compress*), we measured the amount of memory (the page data and the metadata required for accounting) used by each approach and compared the results with plain checkpointing (*cp-only*).

Figure 4.5 shows that most of our server programs' write patterns allow for effective deduplication of the checkpoint history, reducing the memory footprint up to 82.33% (`lighttpd`). For `bind`'s less stable working set, deduplication is less effective, resulting in only 38.40% footprint reduction. Com-



(a): runtime **(b):** memory usage

Figure 4.7: Impact of deduplication and compression vs. number of checkpoints (`lighttpd`).

pression, on the other hand, was highly effective in reducing `bind`'s memory footprint. Without this exception, compression and deduplication reported similar reduction across our server programs. The combination of both strategies (*dedup+compress* is, as expected, even more effective, as it allows for memory footprint reductions of up to 95.23% (`postgresql`).

Run-time Overhead While reducing the memory footprint, enabling deduplication and/or compression also increases the run-time overhead. Figure 4.4 shows the throughput (requests per second) reported by our server programs when enabling deduplication, compression, or both.

As shown in the figure, the throughput degradation induced by plain checkpointing on the server programs ranges between 16 % for `httpd` and 70 % for `postgresql` with an average of 33.44 % (geometric mean). Server programs with shorter request-processing loops yield a higher checkpointing frequency, naturally increasing the performance impact.

We observed that compression and deduplication impose a similar throughput degradation on the server programs (63.19 % vs. 63.96 %, geometric mean). Furthermore the cost of deduplication and compression are not additive when combining the two approaches, but instead tend to get amortized, resulting in an average overall degradation of 68.44 %. Since the combination of both approaches incurs only acceptably higher overhead (+4.5 % compared to deduplication or compression, geometric mean) while enabling the most space-efficient (and scalable) checkpointing strategy, *DeLorean* enables both memory saving techniques by default. However, considering the signif-

Test	Linear [sec]			Binary [sec]		
	10k	100k	1M	10k	100k	1M
Full (compression)	0.960	9.851	99.065	0.061	0.724	8.544
Full	0.155	1.728	18.655	0.061	0.724	8.605
On-demand	0.088	0.999	11.414	0.061	0.739	8.592

Table 4.2: Search tests with 13 pages working sets.

icant difference with plain checkpoint (+35 %), the user is able to disable the space-efficent methods to allow for better runtime-efficient checkpointing.

4.6.4 Time-travelling Introspection Performance

In this section, we evaluate the performance of *DeLorean*'s *time travelling introspection* strategy, which relies on efficient queries over the checkpoint history accumulated during execution. Table 4.2 reports the time to complete a query through the full checkpoint history with an average of 13 pages per checkpoint—corresponding to the highest average checkpoint size observed across our server programs during the execution of our benchmarks. For this experiment, we selected a simple query condition comparing the content of a known integer global variable against an expected value. Moreover, we filled the checkpoint history with a constant number of pages (13) per checkpoint. We employed deduplication do make sure that the pages would fit the memory for $1M$ checkpoints. However, this does not impact the rollback performance, hence the results of the experiment proposed. We ran this microbenchmark 11 times (with negligible variations) and report the median.

To thoroughly evaluate the query run time, we have analysed the impact of different factors:

(a) the rollback mechanisms, namely the *full* mechanism (rolling back all the pages), and the *on-demand* mechanism (rolling back only the target page);

(b) the query strategy, either *linear* or *bisect*,

(c) and whether compression is used.

As expected, Table 4.2 shows that the query run time linearly increases with the number of checkpoints independently of the particular factors considered. We now examine the impact of each factor in more detail.

Rollback mechanism The *full* rollback mechanism seeks to support arbitrary search queries, but it also requires building the permanent rollback tree first and issuing a full rollback operation for each condition evaluation. Table 4.2 depicts the impact of these costs on the query run time. For example, for $1M$ checkpoints, the query takes approximately 99 seconds to inspect all the 13 million pages in the checkpoint history (*linear* query strategy). While this is relatively efficient, the other rollback mechanisms provide better performance. Finally, the *on-demand* rollback mechanism, applicable to our example test case since the location and the size of the inspected variables are known (e.g., no pointers are used), reduces the time to process all the checkpoints by around 88%.

Query strategy We now compare the *linear*-based results examined thus far with the results reported by our *bisect* query strategy. The latter strategy improves the query performance, since bisecting searches only through $O(log(n))$ checkpoints (where n is the size of the checkpoint history). Table 4.2 details the improvements, showing that, for most rollback mechanisms, the much faster *bisect* performance is still bottlenecked by the constant cost of building the permanent rollback tree (around 8 seconds for $1M$ checkpoints).

Compression Unlike deduplication, the use of compression has a negative impact on query performance, given that every checkpoint examined during the search operations needs to be decompressed at rollback time. Table 4.2 depicts this cost when using the `full` rollback mechanism (providing worst-case results). As shown in the table, in this scenario, a query becomes approximately 5.5 times slower than without compression. When bisecting is possible, however, the same cost is essentially not noticeable.

4.6.5 Comparison with Existing Solutions

We compared our checkpointing and introspection overhead to two other systems that offer comparable (and state-of-the-art) functionality: the `gdb` general-purpose debugger and the `rr` record-and replay debugger [19].

We implemented checkpointing-based introspection, including linear and bisect search on top of `gdb`'s own fork-based checkpointing as a `gdb` extension. We configured `rr`'s replay strategy to re-execute the recorded trace and perform search using a conditional breakpoint set at the point where

DeLorean would take a checkpoint. *DeLorean* itself is configured to use compression and deduplication.

We report the results of a synthetic experiment on `lighttpd` (but we observed similar results on the other server programs), which we exercised with the Apache Benchmark (`ab`) issuing 10,000 requests. We used this number of requests since `gdb` could not scale to a much larger checkpoint history, but we remark that our results can linearly extend to even millions of requests. Table 4.3 shows that *DeLorean* has clearly the smallest recording overhead (3 seconds). The runner-up is `rr`, whose recording time is roughly 3.5 times slower then *DeLorean* (11 seconds). `gdb` recording time reported the worst recording performance (464 seconds), which corresponds to a 422x slowdown compared to the 1.1 seconds execution time without recording. Not surprisingly, `gdb`-based checkpointing also consumed the largest amount of memory (over 2 GB), followed by *DeLorean* and `rr`, which reported the smallest (3.6 MB) memory footprint.

Moreover, when searching linearly through the checkpoints in the worst case scenario—search for an expression that is never satisfied—`gdb` also takes the longest time (590 seconds). When operating bisect search with *DeLorean*, in contrast, the search time is drastically reduced to under 1 second. `rr` and *DeLorean* are in the same ballpark in terms of linear search performance (30 and 28 seconds, respectively). `rr`'s design cannot, however, support the more efficient bisect search, which would thus allow *DeLorean* to complete the example query introduced earlier with a $\sim$30x speedup compared to `rr`'s linear search strategy. In summary, *DeLorean* has better recording performance (as a result of trading off on memory usage), comparable search performance when not bisecting, and better search performance when bisecting.

We also compared our results with Scribe [78], a transparent low-overhead record-and-replay system. For practicality reasons, we used the virtual machine image offered for download by its authors (thus reporting even optimistic overheads for Scribe, given the generally slower baseline). The virtual machine was equipped with Ubuntu 12.04 and the custom Scribe Linux kernel v2.6.35. Due to kernel failures, we were only able to test two of our servers, namely `lighttpd` and Apache `httpd`. While the memory used to log the events of the two servers was relatively small ($83MB$ and $90MB$, respectively), we reported significant run-time overhead. We measured 682.74% run-time overhead for `lighttpd` and 255.31% for Apache httpd. *DeLorean* in a similar configuration—same virtualization techniques, but newer kernel—only reported roughly half the of the overhead when run in a virtual machine on the same physical machine.

	Recording		Introspection	
	time [sec]	space [MB]	linear [sec]	binary [sec]
DeLorean	3.2	42.4	30.2	0.2
GDB	464.1	2,298.0	590.1	0.9
RR	11.1	3.6	28.1	—

Table 4.3: Comparison of DeLorean, RR, and GDB on `lighttpd` for $10k$ server requests.

4.6.6 Case Studies

To demonstrate the effectiveness of *DeLorean*, we present two case studies, which show that *DeLorean* can prove useful in debugging arbitrary memory corruption and memory leaks.

Arbitrary Memory Corruption

an infamous bug in the FFmpeg library [14], which drives popular servers such as FFserver, causes erroneous handling of 4X videos under particular inputs and results in wild writes to random memory locations [8]. Servers that use the buggy FFmpeg library are susceptible to arbitrary state corruption when receiving particular requests. For example, a request-triggered wild memory write in FFserver can corrupt one of the `FFServerStreams` contained in `FFserverConfig` used to handle client feeds information. Since such streams are stored as a linked list, the `next` pointer referencing the next entry in the list may be overwritten with, say, `0xDEADBEEF`. The program may only crash several requests later, when the corrupted pointer is used while processing a completely unrelated event. Reproducing the crash alone does not necessarily help the user to identify the root cause, since the crash and trigger event are temporally distant, making difficult to relate the pointer corruption to the crash.

With *DeLorean*, the user can configure the debugger to take a checkpoint for each server request. This is possible by issuing the command `dl checkpoint take` and specifying the end of the server request loop as a target location. The user then allows the program to continue the execution and activate the test workload. Once the server crashes, *DeLorean* allows the user to efficiently query all the checkpoints taken during the execution in search of the trigger event.

In this example, we search for the first checkpoint containing the pointer with the corrupted value. This query can be performed by means of bisection. For instance, if the corrupted pointer contains the value `0xDEADBEEF`),

the developer can issue `dl search stream->next == 0xDEADBEEF` search command.

If successful, the `search` command automatically rolls the program state back to the first checkpoint that satisfies the given condition, or the present state otherwise. After restoration, the user can check for irregularities in the trigger request and the program state to identify the root cause quickly.

Memory Leak

some applications (for example PostgreSQL) implement their own memory garbage collection strategy by employing approaches like reference counting or context management. In the case of PostgreSQL, most of the backend operations rely on a context which keeps track of allocated memory and, each time a context is freed, so is all the referenced memory in that context. As a result, mismanagement of contexts can cause memory leaks. A typical bug is for memory allocated for each request to be incorrectly assigned to a long-lived context which is not freed until the server is reloaded. In such a situation, the server will slowly go out of memory (after a certain number of requests), making it difficult for users to determine the root cause of the bug.

With *DeLorean*, a user can inspect the context and, using either `dl search` or `dl for`, can determine the number of referenced objects as well as their location and content in memory. Using this strategy, the cause of a memory leak becomes very quickly apparent. Similarly, in the case of other systems employing reference counting, *DeLorean* can allow users to determine where and when a reference cycle was introduced by analyzing counter management data structures for a target object.

4.7 Discussion

Firstly, *DeLorean* is mainly aimed at aiding debugging for applications that have a medium-sized impact in terms of memory footprint (i.e., server applications). However, applications with bigger memory footprint can also be debugged while inevitably increasing the memory requirements for storing a high number of checkpoints.

Additionally, since *DeLorean* makes a significant use of the features and the environment provided by `gdb`, we share the same limitations that already affect `gdb`. This is also the case for the applicability of the tool.

While our efforts have focused on improving the memory footprint of our checkpoint-based system, we also aim at trading as little run-time performance as possible. For this reason, we explored the possibility to improve performance by extending *DeLorean* with *working set estimation* (WSE) strategies. WSE [155; 156] can help predict pages that are part of the writable working set in the next checkpoint interval, hence applying a WSE-driven precopying strategy rather than a fully copy-on-write strategy may reduce page fault handling costs [141]. However, when applying WSE to our *DeLorean* prototype we have observed little improvement. We attribute this outcome to the significant overhead imposed by the deduplication, which, when enabled, overshadows the benefits introduced by WSE. To improve the effectiveness of WSE and the overall system performance, an option is to consider an asynchronous deduplication and compression strategy. However, asynchronously processing the checkpointed data also introduces other concerns. In particular, if the speed with which new pages are copied is higher than that of asynchronous page processing operations, the memory reduction may become insufficient to support a large checkpoint history.

Another aspect that we did not explore in *DeLorean* is the effect of swapping excessive checkpoint data to the disk, rather than simply evicting them from memory. This strategy can allow users to maintain even larger checkpoint histories, but the impact on query performance may be significant.

Finally, while we consider *DeLorean* on its own a powerful debugging tool, we do not necessarily perceive checkpoint-assisted time-travelling introspection as an alternative to record-replay-based systems. We instead see potential in combining other debugging systems with *DeLorean* introspection capabilities, for example to assist the user in identifying the root cause of a bug after a record-and-replay system has helped her reproduce the bug on a lengthy test workload.

4.8 Related Work

Record/Replay systems Research in record/replay techniques has led to a broad range of software-based [122; 72; 39; 137; 78; 134] and hardware-based [97; 98; 59; 27] solutions, which improve debuggability for both operating systems [72; 39] and user applications [137; 78; 134; 59]. Nonetheless, the capabilities and focus of these systems vary considerably, as some focus on replay speed [72; 59; 30], others on live execution after replay [78], and yet others on supporting replay sessions on modified versions of the debugged

applications [137]. While hardware-based solutions are appealing, dedicated hardware is not usually available on commodity systems. Software-based solutions are more convenient, but incur much higher overheads on recording and replaying. As a trade-off, OS-level solutions [137; 78; 134] and VMM-level solutions [39; 72] yield better performance, although they still incur significant record-time overheads. *DeLorean* does not offer replay capabilities but focuses on *better* debugging instead, by explicitly supporting queries over the execution history for root-cause analysis purpose, and explores high-speed checkpointing as a convenient and efficient way to search through such history. Moreover, our approach is orthogonal to record/replay systems and it is trivial to employ checkpointing during a replay phase and add search functionalities during off-line debugging sessions.

Checkpointing Over the past 15 years, the research community has proposed many checkpointing solutions. User-level implementations [13; 42; 51; 112; 116; 118; 119; 128] are generally easier to deploy, but they also typically incur significant run-time overhead, making them ill-suited for high-frequency memory checkpointing. Compiler-based solutions [140] are more efficient, but they also require recompilation, reducing deployability. Other approaches improve performance by adding checkpointing support at the virtual machine monitor level [72; 33; 108; 143]. A disadvantage of such methods is the necessity to checkpoint the whole virtual machine rather specific applications of interest. Approaches using dedicated hardware [44; 126] are more efficient but impractical for commodity systems. Finally, kernel-level checkpointing solutions have been explored before [2; 52; 56; 76; 77; 106; 125; 136]. Unlike *DeLorean*, however, they either rely on kernel patches [2; 52; 76] compromising their general applicability, or operate as kernel modules that implement full-blown memory containers [106; 125]. The recent SMC [141] implements a more deployable checkpointing solution in a self-contained kernel module. However, SMC speculative checkpointing strategy tends to generate duplicates—trading memory usage for better performance—resulting in a nonscalable solution when maintaining a large checkpoint history. *DeLorean*, in contrast, relies on page deduplication and compression to keep a large history and provide even better scalability than regular checkpointing solutions.

Deduplication Deduplication is a common strategy to reduce memory usage in virtualized environments [23]. Prior work on disk-based checkpointing also used deduplication to reduce the size of on-disk checkpoints [57;

104; 108]. *DeLorean* follows a similar intuition to reduce the footprint of in-memory checkpoints for debugging. Beside deduplicating across applications, *DeLorean* also deduplicates pages across checkpoints for a given process, exploiting the intuition that programs often exhibit recurring steady states and thus yield a growing number of duplicated pages in the checkpoint history.

Compression Page compression is another common technique to reduce memory usage [47; 91; 45]. Prior work in the area, however, is mostly concerned with reducing the impact of I/O operations and improving system, rather than debugging, performance. Similar to deduplication, compression has also been used in prior disk-based checkpointing solutions to improve disk space utilization [63; 62; 114]. *DeLorean*, in contrast, draws inspiration from memory compression solutions such as `zram` [91], to aggressively compress checkpointed data in memory and support a very large checkpoint history despite deduplication low effectiveness. Finally, in contrast to previous work in the general area of disk-based checkpointing, *DeLorean* combines compression and deduplication to further reduce its memory footprint.

4.9 Conclusion

Debugging is hard and time-consuming and effort has been put in improving this task. Recent research has paid attention to record/replay debugging systems to aid cyclic debugging of nondeterministic applications and operating systems. However, such systems result in slower applications runtime due to the overhead imposed by the recording phase. Further, replay still requires lengthy debugging iterations over the collected traces.

In this paper, we have proposed *DeLorean*, a checkpoint-based time-travelling debugging system which trades replay capabilities for more efficient debugging and root-cause analysis. Our high-frequency checkpointing strategy reduces the run-time overhead of the recording phase, while minimizing the footprint of checkpointed memory by means of page deduplication and compression. We have further proposed a new time-travelling introspection mechanism that quickly iterates over the checkpointed memory to pinpoint the root cause of long-lived event-driven software bugs.

Our analysis on real-world server applications demonstrates that *DeLorean* imposes a reasonable run time overhead in recording phase and achieves up to 94.25% memory footprint reduction compared to incremental check-

pointing. Moreover, our results demonstrate that checkpoint-assisted time-travelling introspection is efficient in quickly searching over the checkpoint history for a large window of observation, requiring, in the best case, less than few seconds to complete a search.

CHAPTER 5

GENERAL CONCLUSION

Checkpointing is an important technique that has many applications inside the reliability domain, such as automated error recovery [152; 135; 53; 83; 54; 121; 117; 125; 51; 68; 115] and debugging [129; 128; 61; 72; 48]. An integral part of checkpointing is taking a snapshot of a process' memory, also known as memory checkpointing. For applications that require a high checkpointing frequency, the costs for memory checkpointing have a significant effect on their run-time performance overhead. Further, applications that require high checkpointing frequencies and a long checkpoint history additionally face the challenge of storing checkpoints efficiently lest they lose precious application memory. As we showed in Chapter 2 and Chapter 3, current checkpointing solutions are not well suited for high checkpointing frequencies, as they (1) incur high run time overhead, have high deployment costs and fail to offer memory guarantees and, (2) do not deal with the problem of storing the large number of checkpoints produced by high checkpointing frequencies. This thesis provides contributions towards making checkpointing viable for high checkpoint frequencies.

In Chapter 2 we investigated pure user-level high frequency checkpointing. Limiting our investigation to a pure userland implementation addresses the issue that, when deploying checkpointing solutions in the real world, changes to the kernel are often not possible. We evaluated three page granular check-

pointing mechanisms, as well as one instrumentation-based checkpointing technique on a set of server applications. To achieve a high checkpoint frequency, we took a checkpoint for each request, a common use-case, for example, in request oriented recovery. Our experiments confirmed that with higher checkpointing frequency, the page granular user-level techniques suffer from a high performance overhead. The instrumentation-based undolog, while offering better performance for high frequency applications, suffers from unbounded memory usage. We then presented LMC, a system for efficient memory-bound high frequency user-level checkpointing. Instead of relying on inefficient kernel primitives that offer page granular copy-on-write semantics, LMC utilizes compiler instrumentation to implement copy-on-write on byte-granularity. In contrast to the undolog approach, LMC stores checkpoint data in a shadow-state organization, thereby limiting memory usage.

In Chapter 3, we revisited page granular checkpointing, however, this time relaxing our requirement of not allowing kernel modifications. We started our investigation with an analysis of the costs of page granular checkpointing, showing that saving a page using Linux CoW mechanism—by forking—is circa eight times as expensive as just copying the page, ignoring the costs for the fork system call itself, which are substantial. The results of this investigation motivated the design of SMC. While SMC allows applications to efficiently access the kernel's CoW mechanism, it also features a speculation component, which by speculatively copying hot pages aims to avoid the eight times higher cost of kernel CoW over plain copying. As the speculation essentially consists of writable working set estimation, we evaluated two established WWSE techniques—Active-RND and Active-CKS—and our novel GSpec WWSE algorithm. In contrast to the traditional WWSE techniques, GSpec's approach can automatically tune to different workloads.

Our experiments showed a hypothetically perfect and overhead-free speculation can reduce the overhead of CoW checkpointing by 29.3 percentage points from 44.9% to 15.6% in the tested scenario. This confirms that speculation is a useful tool to reduce COW induced checkpointing overhead. Further, we observed that all tested speculation strategies lead to an improvement in terms of performance overhead while leading to a modest increase of memory consumption and recovery time. GSpec showed the highest speculation accuracy across the tested server applications and reduced checkpointing overhead by 14.1 percentage points from 44.9% to 30.8%, compared to plain COW.

In Chapter 4, we presented DeLorean, a debugging system leveraging check-

point-assisted time-traveling introspection, as a real-world use case of high frequency checkpointing. DeLorean ephamsizes the importance of efficient checkpoint storage and exploration, two important problems that arise when many checkpoints are taken with high frequency. DeLorean uses the checkpointing component presented in Chapter 3 to achieve efficient high frequency checkpointing in order to reduce the run-time overhead during the recording phase. By compressing and deduplicating checkpointed pages, DeLorean is capable of reducing the memory footprint of the checkpointing storage by 95%, making it possible to store millions of checkpoints. Further, we explored efficient ways to inspect the checkpoint history. Knowing memory locations of interest partial on-demand rollback can save up to 88% of search time in our experiments. This can be further improved if certain properties of the search criteria are known, which in some cases allows us to use use a bisect search strategy effectively enabling us to search 1 million checkpoints in around 8.5 seconds.

5.1 Direction for Future Work

As we point out in Chapter 2, the required instrumentation causes run-time performance overhead when checkpointing is disabled. We propose using basic block cloning to run application without instrumentation when checkpointing is disabled to mitigate this cost. Adding this capability and exploring its trade-offs is an obvious first extension of the work presented in Chapter 2.

Another limitation of the checkpointing technique presented in Chapter 2 is due to its compile time instrumentation approach. This approach requires the targeted application and shared libraries to be available in source code. It is conceivable to apply the checkpointing instrumentation using static or dynamic binary instrumentation, although instrumentation on binary level usually involves a larger run-time performance overhead. Further, such a solution limits the use of some optimizations presented in Chapter 2 as these optimizations rely on information available in LLVM IR code. One possible workaround is to rely on binary instrumentation for the parts of the application that are not available as source, for example shared libraries, and use LMC's compile time instrumentation for the rest of the application. Another approach could be to manuallay annotate shared library functions, comparable to LLVM DataFlowsanitizer's ABI list [133], and use this information to add instrumentation that copies the modified state of these functions to the

checkpoint.

Also, in Chapter 2 we implicitly assume that the rate in which checkpoints are taken is constant throughout the whole execution time. In such cases the decision if a specific use qualifies as high-frequency checkpointing is straight forward. However, it is conceivable that the executed instructions per checkpoint ratio changes significantly throughout the execution of the application. An example for this are computationally-expensive requests that lead to long-lasting checkpointing intervals. This motivates extending LMC with the possibility to switch to page granular checkpointing for such long-lasting checkpointing periods and switch back to instrumentation-based checkpointing for short checkpointing intervals. To achieve this, one first would have to establish the break even point of the different checkpointing costs to define a the duration of long-lasting checkpointing intervals. Further, one needs to identify the code paths that lead to long-lasting checkpointing intervals. Ideally, these code-paths can be identified statically at compile time by an extension of LMC's instrumentation component. In case the checkpointing interval is heavily payload dependent, it might be more beneficial to switch checkpointing methods based on data gathered during the execution of the checkpointed application.

One possible extension of the work presented in Chapter 3 is to explore ways to improve SMC's speculation accuracy. A possible approach is to allow speculation mechanisms access to contextual data, such as the request type or client ID to maintain multiple speculation contexts. The motivation of this approach is that it is likely that different requests lead to different memory access patterns. For example, for two request types that lead to two distinct but constant writable working sets, a global speculation context will always lead to miss-speculation. If, on the other hand, one maintains a separate speculation context for each request type, even a simple greedy speculation algorithm can exhibit perfect speculation results. One of the key challenges would be finding out what kind of data is an efficient identifier for speculation contexts and if this approach would be generalizable.

Further it would be interesting to reevaluate pure user-level checkpointing with speculation. As we have shown in Chapter 2, `mprotect`-based checkpointing, for example, exhibits a significant run-time overhead due to the high signal handling costs. Through speculation, however, we expect to reduce this overhead by avoiding expensive CoW, ideally—with perfect speculation—up to match the performance results of the oracle speculation mechanism in Chapter 3.

5.2 Conclusion

In summary, this thesis shows the need for specialized high frequency memory checkpointing techniques and proposes enhancements to current checkpointing techniques to make them fit for use cases that require checkpoints to be taken with a high frequency. We explored the deployability trade-off of different checkpointing techniques and showed that if the target application can be recompiled, pure userland techniques relying on compiler instrumented techniques can offer a significantly better run-time performance than their page-granular counterparts. LMC's shadow state inspired checkpoint organization shows that it is not necessary to give up memory guarantees to achieve this. Further, we showed that speculation and exporting copy-on-write functionality as a first level kernel primitive to the userland lowers the overhead of page granular checkpointing significantly. In turn, the techniques presented in this thesis allow for efficiently storing and searching a large number of checkpointed data resulting from the high checkpoint frequency.

We hope that these techniques combined open up a whole new range of possible scenarios for high frequency memory checkpointing, such as the time traveling debugging system DeLorean presented in this thesis.

BIBLIOGRAPHY

[1] GDB command files. https://sourceware.org/gdb/onlinedocs/gdb/Command-Files.html.

[2] OpenVZ. http://openvz.org.

[3] PostgreSQL. http://www.postgresql.org.

[4] pyftpdlib. https://code.google.com/p/pyftpdlib.

[5] SysBench. http://sysbench.sourceforge.net.

[6] /proc/sys documentation. https://www.kernel.org/doc/Documentation/sysctl/README, 1998.

[7] queryperf. https://github.com/davidmmiller/bind9/tree/master/contrib/queryperf, 2001.

[8] CVE 2009-0385. http://www.trapkit.de/advisories/TKADV2009-004.txt, 2009.

[9] The zsmalloc allocator. https://lwn.net/Articles/477067/, 2012.

[10] BIND. https://www.isc.org/downloads/bind/, 2013.

[11] Apache benchmark. http://httpd.apache.org/docs/2.0/programs/ab.html, 2015.

[12] Apache HTTP server. http://httpd.apache.org/, 2015.

[13] CRIU. http://criu.org, 2015.

[14] FFmpeg. https://www.ffmpeg.org/, 2015.

[15] GDB. http://www.gnu.org/software/gdb/, 2015.

[16] GDB python API. https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html, 2015.

[17] git-bisect manual page. https://www.kernel.org/pub/software/scm/git/docs/v1.7.10/git-bisect.html, 2015.

[18] Kernel Probes. http://www.kernel.org/doc/Documentation/kprobes.txt, 2015.

[19] rr: lightweight recording and deterministic debugging. http://rr-project.org/, 2015.

[20] SysBench. https://github.com/akopytov/sysbench, 2015.

[21] Akritidis, P. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Security Symp.* (2010), p. 12.

[22] Altekar, G., and Stoica, I. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the Symposium on Operating Systems Principles* (2009).

[23] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using ksm. In *Proceedings of the Linux Symposium* (2009).

[24] Azimi, R., Soares, L., Stumm, M., Walsh, T., and Brown, A. D. Path: Page access tracking to improve memory management. In *Proceedings of the Sixth International Symp. on Memory Management* (2007), pp. 31–42.

[25] Bansal, S., and Modha, D. S. CAR: clock with adaptive replacement. In *Proceedings of the Third USENIX Conf. on File and Storage Technologies* (2004), pp. 187–200.

[26] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symp. on Operating Systems Principles* (2003), pp. 164–177.

[27] Basu, A., Bobba, J., and Hill, M. D. Karma: Scalable deterministic record-replay. In *Proceedings of the International Conference on Supercomputing* (2011).

[28] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Symp. on Operating Systems Design and Implementation* (2012), pp. 335–348.

[29] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.

[30] Bhansali, S., Chen, W.-K., De Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., and Chau, J. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the International Conference on Virtual Execution Environments* (2006).

[31] Bhat, K., Vogt, D., Kouwe, E., Gras, B., Sambuc, L., Tanenbaum, A., Bos, H., and Giuffrida, C. *OSIRIS: Efficient and consistent recovery of compartmentalized operating systems*. Institute of Electrical and Electronics Engineers, Inc., 9 2016, pp. 25–36.

[32] Bronevetsky, G., Marques, D., Pingali, K., Szwed, P., and Schulz, M. Application-level checkpointing for shared memory programs. In *Proceedings of the 11th International Conf. on Architectural Support for Programming Languages and Operating Systems* (2004), pp. 235–247.

[33] Bugnion, E., Chipounov, V., and Candea, G. Lightweight snapshots and system-level backtracking. In *Proceedings of the 14th Conf. on Hot Topics in Operating Systems* (2013), p. 23.

[34] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[35] Carr, R. W., and Hennessy, J. L. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symp. on Operating Systems Principles* (1981), pp. 87–95.

[36] Chen, L., and Avizienis, A. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on* (Jun 1995), pp. 113–.

[37] Chiang, J.-H., Li, H.-L., and Chiueh, T.-c. Working set-based physical memory ballooning. In *Proceedings of the 10th International Conf. on Autonomic Computing*, pp. 95–99.

[38] Choi, S.-E., and Deitz, S. Compiler support for automatic checkpointing. In *Proceedings of the 16th Annual International Symp. on High Performance Computing Systems and Applications* (2002), pp. 213–220.

[39] Chow, J., Lucchetti, D., Garfinkel, T., Lefebvre, G., Gardner, R., Mason, J., Small, S., and Chen, P. M. Multi-stage replay with crosscut. In *Proceedings of the International Conference on Virtual Execution Environments* (2010).

[40] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the Second Symp. on Networked Systems Design and Implementation* (2005), pp. 273–286.

[41] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2008).

[42] Dieter, W. R., and E., L. J. User-level checkpointing for LinuxThreads programs. In *Proceedings of the USENIX Annual Technical Conference* (2001), pp. 81–92.

[43] Döbel, B., and Härtig, H. Who watches the watchmen? – protecting operating system reliability mechanisms. In *HotDep* (2012).

[44] Doudalis, I., and Prvulovic, M. KIMA: Hybrid checkpointing for recovery from a wide range of errors and detection latencies. Tech. rep., Georgia Institute of Technology, 2010.

[45] Douglis, F. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the USENIX Annual Technical Conference* (1993).

[46] Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. A survey of fault tolerance mechanisms and Checkpoint/Restart implementations for high performance computing systems. *J. Supercomput. 65*, 3 (Sept. 2013), 1302–1326.

[47] Ekman, M., and Stenstrom, P. A robust main-memory compression scheme. In *Proceedings of the International Symposium on Computer Architecture* (2005).

[48] Feldman, S. I., and Brown, C. B. Igor: A system for program debugging via reversible execution. *SIGPLAN Not. 24*, 1 (Nov. 1988), 112–123.

[49] Ferreira, C. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*. 2006.

[50] Ferreira, K. B., Riesen, R., Brighwell, R., Bridges, P., and Arnold, D. libhashckpt: Hash-based incremental checkpointing using GPU's. In *Proceedings of the 18th European Conf. on Recent Advances in the Message Passing Interface* (2011), pp. 272–281.

[51] Gao, Q., Zhang, W., Tang, Y., and Qin, F. First-aid: Surviving and preventing memory management bugs during production runs. In *Proceedings of the Fourth ACM European Conf. on Computer Systems* (2009), pp. 159–172.

[52] Gioiosa, R., Sancho, J. C., Jiang, S., Petrini, F., and Davis, K. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE Conf. on Supercomputing* (2005), p. 9.

[53] Giuffrida, C., Cavallaro, L., and Tanenbaum, A. S. We crashed, now what? In *Proceedings of the Sixth Workshop on Hot Topics in System Dependability* (2010), pp. 1–8. ACM ID: 1924912.

[54] Giuffrida, C., Iorgulescu, C., and Tanenbaum, A. S. Mutable checkpoint-restart: automating live update for generic server programs. In *Proceedings of the 15th International Middleware Conference* (2014), pp. 133–144.

[55] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Edfi: A dependable fault injection tool for dependability benchmarking experiments. In *Proceedings of the Pacific Rim International Symp. on Dependable Computing* (2013), pp. 1–10.

[56] Hargrove, P. H., and Duell, J. C. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series 46*, 1 (Sept. 2006), 494.

[57] Heo, J., Yi, S., Cho, Y., Hong, J., and Shin, S. Y. Space-efficient page-level incremental checkpointing. In *Proceedings of the ACM Symp. on Applied Computing* (2005), pp. 1558–1562.

[58] Hertz, M., Kane, S., Keudel, E., Bai, T., Ding, C., Gu, X., and Bard, J. E. Waste not, want not: Resource-based garbage collection in a shared environment. In *Proceedings of the International Symp. on Memory Management* (2011), pp. 65–76.

[59] Honarmand, N., Dautenhahn, N., Torrellas, J., King, S. T., Pokam, G., and Pereira, C. Cyrus: Unintrusive application-level record-replay for replay parallelism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[60] Hruby, T., Vogt, D., Bos, H., and Tanenbaum, A. S. Keep net working - on a dependable and fast networking stack. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2012), DSN '12, IEEE Computer Society, pp. 1–12.

[61] Hursey, J., January, C., O'Connor, M., Hargrove, P. H., Lecomber, D., Squyres, J. M., and Lumsdaine, A. Checkpoint/restart-enabled parallel debugging. In *Proceedings of the 19th European Message Passing Interface Conference* (2010), pp. 219–228.

[62] Ibtesham, D., Arnold, D., Bridges, P., Ferreira, K., and Brightwell, R. On the viability of compression for reducing the overheads of Checkpoint/Restart-Based fault tolerance. In *Proceedings of the 41st International Conf. on Parallel Processing* (2012), pp. 148–157.

[63] Islam, T. Z., Mohror, K., Bagchi, S., Moody, A., De Supinski, B. R., and Eigenmann, R. McrEngine: a scalable checkpointing system using data-aware aggregation and compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2012).

[64] Jiang, S., Chen, F., and Zhang, X. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Tech. Conf.* (2005), pp. 323–336.

[65] Jimborean, A., Loechner, V., and Clauss, P. Handling multi-versioning in LLVM: Code tracking and cloning. In *Workshop on Intermediate Representations* (Apr. 2011).

[66] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conf. on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 14–24.

[67] Joo, Y., Ryu, J., Park, S., and Shin, K. G. FAST: quick application launch on solid-state drives. In *Proceedings of the Ninth USENIX Conf. on File and Stroage Technologies* (2011), pp. 19–32.

[68] Kadav, A., Renzelmann, M. J., and Swift, M. M. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the 18th International Conf. on Architectural Support for Programming Languages and Operating Systems* (2013), vol. 41, ACM, pp. 473–484.

[69] Kannan, S., Gavrilovska, A., Schwan, K., and Milojicic, D. Optimizing checkpoints using nvm as virtual memory. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symp.* (2013), pp. 29–40.

[70] Kashyap, S., Min, C., Lee, B., Kim, T., and Emelyanov, P. Instant os updates via userspace checkpoint-and-restart. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2016), USENIX ATC '16, USENIX Association, pp. 605–619.

[71] Kasikci, B., Schubert, B., Pereira, C., Pokam, G., and Candea, G. Failure sketching: a technique for automated root cause diagnosis of in-production failures. In *Proceedings of the Symposium on Operating Systems Principles* (2015).

[72] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (2005), p. 1.

[73] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on*

Operating Systems Principles (New York, NY, USA, 2009), SOSP '09, ACM, pp. 207–220.

[74] Kneschke, J., Ruusamae, E., Ruckert, M., mOo, Jakabosky, R., and Buhler, S. lighttpd. `http://www.lighttpd.net/`.

[75] Krishnakumar, K. Micro-genetic algorithms for stationary and non-stationary function optimization. In *Proceedings of the Intelligent Control and Adaptive Systems Conf.* (1990), pp. 289–296.

[76] Laadan, O., and Hallyn, S. E. Linux-CR: Transparent application checkpoint-restart in linux. In *Proceedings of the Linux Symposium* (2010), pp. 159–172.

[77] Laadan, O., and Nieh, J. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *Proceedings of the USENIX Annual Technical Conference* (2007), pp. 1–14.

[78] Laadan, O., Viennot, N., and Nieh, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* (2010).

[79] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symp. on Code Generation and Optimization* (2004), p. 75.

[80] Lattner, C., and Adve, V. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI* (Chigago, Illinois, June 2005).

[81] Lattner, C., Lenharth, A., and Adve, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2007), pp. 278–289.

[82] Lawall, J. L., and Muller, G. Efficient incremental checkpointing of Java programs. In *DSN* (2000), pp. 61–70.

[83] Lenharth, A., Adve, V. S., and King, S. T. Recovery domains: An organizing principle for recoverable operating systems. In *Proceedings of the 14th International Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), vol. 37, ACM, pp. 49–60.

[84] Li, C.-C., and Fuchs, W. CATCH: Compiler-assisted techniques for checkpointing. In *Proceedings of the 20th International Symp. on Fault-tolerant computing* (1990), pp. 74–81.

[85] Li, Y., and Lan, Z. FREM: A fast restart mechanism for general Checkpoint/Restart. *IEEE Trans. Comput. 60*, 5 (May 2011), 639–652.

[86] Lipowski, A., and Lipowska, D. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications 391*, 6 (2012), 2193–2196.

[87] Lohr, S., and Markoff, J. Windows is so slow, but why? `https://www.nytimes.com/2006/03/27/technology/windows-is-so-slow-but-why.html`.

[88] Lu, P., Ravindran, B., and Kim, C. VPC: scalable, low downtime checkpointing for virtual clusters. In *Proceedings of the 24th International Symp. on Computer Architecture and High Performance Computing* (Oct. 2012), pp. 203–210.

[89] Lu, P., and Shen, K. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of the USENIX Annual Tech. Conf.* (2007), pp. 1–15.

[90] Magenheimer, D. Memory overcommit... without the commitment. In *Xen Summit* (2008), pp. 1–3.

[91] Magenheimer, D. In-kernel memory compression. `https://lwn.net/Articles/545244/`, 2013.

[92] Matsakis, N. D., and Klock, II, F. S. The rust language. *Ada Lett. 34*, 3 (Oct. 2014), 103–104.

[93] McConnell, S. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[94] Miller, B. L., Miller, B. L., Goldberg, D. E., and Goldberg, D. E. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems 9* (1995), 193–212.

[95] Miraglia, A., Vogt, D., Bos, H., Tanenbaum, A., and Giuffrida, C. Peeking into the past: Efficient checkpoint-assisted time-traveling debugging. In *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE)* (Oct 2016), pp. 455–466.

[96] Mitchell, M. *An introduction to genetic algorithms*. MIT press, 1998.

[97] Montesinos, P., Ceze, L., and Torrellas, J. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture* (2008).

[98] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2009).

[99] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Producing wrong data without doing anything obviously wrong! In *ASPLOS* (2009), pp. 265–276.

[100] Nam, H.-c., Kim, J., Hong, S., and Lee, S. Probabilistic checkpointing. In *Proceedings of the 27th International Symp. on Fault-Tolerant Computing* (1997), p. 48.

[101] Necula, G. C., McPeak, S., and Weimer, W. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 128–139.

[102] Nethercote, N., and Seward, J. How to shadow every byte of memory used by a program. In *Proceedings of the Third International Conf. on Virtual Execution Environments* (2007), pp. 65–74.

[103] Nethercote, N., and Seward, J. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2007), pp. 89–100.

[104] Nicolae, B. Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal. In *Proceedings of the International Symp. on Parallel Distributed Processing* (2013), pp. 19–28.

[105] Nicolae, B., and Cappello, F. AI-Ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd International Symp. on High-performance Parallel and Distributed Computing* (2013), pp. 155–166.

[106] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review 36*, SI (Dec. 2002), 361–376.

[107] Ostrand, T. J., and Weyuker, E. J. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes 27*, 4 (July 2002), 55–64.

[108] Park, E., Egger, B., and Lee, J. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conf. on Virtual Execution Environments* (2011), pp. 75–86.

[109] Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K. H., and Lu, S. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles* (2009).

[110] Payer, M. Too much PIE is bad for performance. Tech. rep., 2012.

[111] Payer, M., Kravina, E., and Gross, T. R. Lightweight memory tracing. In *Proceedings of the USENIX Annual Tech. Conf.* (2013).

[112] Plank, J. S., Beck, M., Kingsley, G., and Li, K. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the USENIX Annual Technical Conference* (1995), p. 18.

[113] Plank, J. S., Li, K., and Puening, M. A. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst. 9*, 10 (Oct. 1998), 972–986.

[114] Plank, J. S., Xu, J., and Netzer, R. H. B. Compressed differences: An algorithm for fast incremental checkpointing. Tech. rep., University of Tennessee, 1995.

[115] Portokalidis, G., and Keromytis, A. D. REASSURE: A self-contained mechanism for healing software using rescue points. In *Proceedings of the 6th International Conf. on Advances in Information and Computer Security* (2011), pp. 16–32.

[116] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: treating bugs as allergies—a safe method to survive software failures. *ACM SIGOPS Operating Systems Review 39*, 5 (2005).

[117] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symp. on Operating Systems Principles* (2005), vol. 39, pp. 235–248.

[118] Rieker, M., Ansel, J., and Cooperman, G. Transparent user-level checkpointing for the native POSIX thread library for Linux. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (2006), vol. 6, pp. 492–498.

[119] Ruscio, J. F., Heffner, M. A., and Varadarajan, S. DejaVu: Transparent user-level checkpointing, migration and recovery for distributed systems. In *Proceedings of the ACM/IEEE Conf. on Supercomputing* (2007).

[120] Sade, Y., Sagiv, M., and Shaham, R. Optimizing C multithreaded memory management using thread-local storage. In *Compiler Construction*, pp. 137–155.

[121] Saini, A., Rezaei, A., Mueller, F., Hargrove, P., and Roman, E. Affinity-aware checkpoint restart. In *Proceedings of the 15th International Middleware Conference* (2014), pp. 121–132.

[122] Saito, Y. Jockey: a user-space library for record-replay debugging. In *Proceedings of the International Symposium on Automated Analysis-driven Debugging* (2005).

[123] Sancho, J. C., Petrini, F., Davis, K., Gioiosa, R., and Jiang, S. Current practice and a direction forward in Checkpoint/Restart implementations for fault tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symp.* (2005), p. 300.

[124] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: a fast address sanity checker. In *Proceedings of the of the USENIX Annual Tech. Conf.* (2012), p. 28.

[125] Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., and Keromytis, A. D. ASSURE: Automatic software self-healing using rescue points. In *Proceedings of the 14th International Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 37–48.

[126] Sorin, D. J., Martin, M. M., Hill, M. D., Wood, D., et al. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the International Symposium on Computer Architecture* (2002), pp. 123–134.

[127] Spears, V. M., and Jong, K. A. D. On the virtues of parameterized uniform crossover. In *In Proceedings of the Fourth International Conf. on Genetic Algorithms* (1991), pp. 230–236.

[128] Srinivasan, S. M., Kandula, S., Andrews, C. R., and Zhou, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference* (2004), p. 3.

[129] Subhraveti, D., and Nieh, J. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *Proceedings of the International Conf. on Measurement and Modeling of Computer Systems* (2011), pp. 109–120.

[130] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS) 24*, 4 (2006), 333–360.

[131] Synopsys, Inc. Coverity Scan Open Source Report 2014. Tech. rep., Synopsys, Inc., 2014.

[132] Sysoev, I. nginx. `http://nginx.net`, 2010.

[133] Team, T. C. Dataflowsanitizer design document. `https://clang.llvm.org/docs/DataFlowSanitizerDesign.html`.

[134] Thane, H., and Hansson, H. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems* (2000).

[135] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Triage: diagnosing production run failures at the user's site. *ACM SIGOPS Operating Systems Review 41*, 6 (2007).

[136] Vasavada, M., Mueller, F., Hargrove, P. H., and Roman, E. Comparing different approaches for incremental checkpointing: The showdown. In *Proceedings of the Linux Symposium* (2011), pp. 69–79.

[137] Viennot, N., Nair, S., and Nieh, J. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[138] Vogt, D., Döbel, B., and Lackorzynski, A. Stay strong, stay safe: Enhancing reliability of a secure operating system. In *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS 2010), Paris, France, April 2010* (New York, NY, USA, 2010), ACM.

[139] Vogt, D., Giuffrida, C., Bos, H., and Tanenbaum, A. S. Techniques for efficient in-memory checkpointing. In *Proceedings of the Ninth Workshop on Hot Topics in System Dependability* (2013).

[140] Vogt, D., Giuffrida, C., Bos, H., and Tanenbaum, A. S. Lightweight memory checkpointing. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2015), pp. 474–484.

[141] Vogt, D., Miraglia, A., Portokalidis, G., Bos, H., Tanenbaum, A., and Giuffrida, C. Speculative memory checkpointing. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware '15, ACM, pp. 197–209.

[142] Waldspurger, C. A. Memory resource management in VMware ESX server. In *Proceedings of the Fifth USENIX Symp. on Operating Systems Design and Implementation* (2002), pp. 181–194.

[143] Wang, L., Kalbarczyk, Z., Iyer, R. K., and Iyengar, A. Checkpointing virtual machines against transient errors. In *Proceedings of the 16th International On-Line Testing Symp.* (2010), pp. 97–102.

[144] Wang, Y.-M., Huang, Y., Vo, K.-P., Chung, P.-Y., and Kintala, C. Checkpointing and its applications. In *Proceedings of the 25th International Symp. on Fault-Tolerant Computing* (1995), p. 22.

[145] Woude, J. V. D., and Hicks, M. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 17–32.

[146] Wu, J., Cui, H., and Yang, J. Bypassing races in live applications with execution filters. In *OSDI* (2010), vol. 10, pp. 1–13.

[147] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symp.* (2006), pp. 121–136.

[148] Yang, T., Berger, E. D., Kaplan, S. F., and Moss, J. E. B. CRAMM: virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symp. on Operating Systems Design and Implementation* (2006), pp. 103–116.

[149] Ye, D., Ray, J., Harle, C., and Kaeli, D. R. Performance characterization of SPEC CPU2006 integer benchmarks on x86-64 architecture. In *IISWC* (2006), pp. 120–127.

[150] Zamfir, C., and Candea, G. Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems* (2010).

[151] Zarrabi, A., Samsudin, K., and Wan Adnan, W. A. Linux support for fast transparent general purpose Checkpoint/Restart of multithreaded processes in loadable kernel module. *J. Grid Comput. 11*, 2 (June 2013), 187–210.

[152] Zavou, A., Portokalidis, G., and Keromytis, A. D. Self-healing multitier architectures using cascading rescue points. In *Proceedings of the 28th Annual Computer Security Applications Conf.* (2012), pp. 379–388.

[153] Zaw, E. P., and Thein, N. L. Live virtual machine migration with efficient working set prediction. In *Proc. of First International Conf. on Network and Electronics Engineering* (2011).

[154] Zhang, C., Kelsey, K., Shen, X., Ding, C., Hertz, M., and Ogihara, M. Program-level adaptive memory management. In *Proceedings of the Fifth International Symp. on Memory Management* (2006), pp. 174–183.

[155] Zhang, I., Denniston, T., Baskakov, Y., and Garthwaite, A. Optimizing VM checkpointing for restore performance in VMware ESXi. In *Proceedings of the USENIX Annual Tech. Conf.* (2013), pp. 1–12.

[156] Zhang, I., Garthwaite, A., Baskakov, Y., and Barr, K. C. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th International Conf. on Virtual Execution Environments* (2011), pp. 87–98.

[157] Zhang, W., de Kruijf, M., Li, A., Lu, S., and Sankaralingam, K. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *Proceedings of the 18th International Conf. on Architectural Support for Programming Languages and Operating Systems* (2013), pp. 113–126.

[158] Zhang, X., Dwarkadas, S., and Shen, K. Towards practical page coloring-based multicore cache management. In *Proceedings of the Fourth ACM European Conf. on Computer Systems* (2009), pp. 89–102.

[159] Zhao, C. C., Steffan, J. G., Amza, C., and Kielstra, A. Compiler support for fine-grain software-only checkpointing. In *Proceedings of the 21st International Conf. on Compiler Construction* (2012), pp. 200–219.

[160] Zhao, Q., Bruening, D., and Amarasinghe, S. Efficient memory shadowing for 64-bit architectures. In *Proceedings of the 2010 International Symp. on Memory Management* (2010), pp. 93–102.

[161] Zhao, Q., Bruening, D., and Amarasinghe, S. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the Eighth Annual IEEE/ACM International Symp. on Code Generation and Optimization* (2010), pp. 22–31.

[162] Zhao, Q., Rabbah, R., Amarasinghe, S., Rudolph, L., and Wong, W.-F. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the 17th International Conf. on Compiler Construction* (2008), pp. 147–162.

[163] Zhao, W., Jin, X., Wang, Z., Wang, X., Luo, Y., and Li, X. Low cost working set size tracking. In *Proceedings of the USENIX Annual Tech. Conf.* (2011), pp. 17–28.

[164] Zhao, W., and Wang, Z. Dynamic memory balancing for virtual machines. In *Proceedings of the International Conf. on Virtual Execution Environments* (2009), pp. 21–30.

[165] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conf. on Architectural Support for Programming Languages and Operating Systems* (2004), pp. 177–188.

SUMMARY

Computers and their software play an ever increasing role in our daily life—software runs on our computers, phones, TVs and around our arms in the form of smart watches. While one can argue that software has improved our quality of life in many ways, it is plagued by a problem, which is as old as software itself: Reliable software is hard to build. "Have you tried turning it off and on again", has become our pop-culture's iconic manifestation of this problem. So it does not come as a surprise that the research of methodologies and technologies to make software *reliable* originated at the same time in which digital computers first hit the industry and universities.

Checkpointing is one important technique that originating from this research and has many applications inside the reliability domain, such as automated error recovery and debugging. An integral part of checkpointing is taking a snapshot of a process' memory, also known as *memory checkpointing*, which is the main subject of this thesis.

In particular this thesis concentrates on scenarios that require *high checkpointing frequencies*. Examples of such use cases are automatic error recovery techniques that require checkpoints on every client request or on carefully selected rescue points. Further, in debugging scenarios the ability take checkpoints with very high frequency allows the inspection of arbitrary memory states throughout the execution.

For these high-frequency checkpointing scenarios memory checkpointing is an important cost factor, which makes it an interesting and worthwhile target for optimization. Applications that require high checkpointing frequencies and a long checkpoint history also face the challenge of storing checkpoints efficiently lest they lose precious application memory.

In Chapter 2, we investigate pure user-level high frequency checkpointing. Limiting our investigation to a pure userland implementation addresses the issue that, when deploying checkpointing solutions in the real world, changes to the kernel are often not possible. We examine three page granular check-

pointing mechanisms, as well as one instrumentation-based checkpointing technique on a set of server applications. To achieve a high checkpoint frequency, we take a checkpoint for each request, a common use-case, for example, in request oriented recovery. Our experiments show that with higher checkpointing frequency, the page granular user-level techniques suffer from a high performance overhead. The instrumentation-based undolog, while offering better performance for high frequency applications, suffers from unbounded memory usage. We then present *Lightweight Memory Checkpointing* (LMC), a system for efficient memory-bound high frequency user-level checkpointing. Instead of relying on inefficient kernel primitives that offer page granular copy-on-write semantics, LMC utilizes compiler instrumentation to implement copy-on-write on byte-granularity. In contrast to the undolog approach, LMC stores checkpoint data in a shadow-state organization, thereby limiting memory usage.

In Chapter 3, we revisit page granular checkpointing, however, this time relaxing our requirement of not allowing kernel modifications. We start our investigation with an analysis of the costs of page granular checkpointing, showing that saving a page using Linux CoW mechanism—by forking—is circa eight times as expensive as just copying the page, ignoring the costs for the fork system call itself, which are substantial. The results of this investigation motivate the design of the second memory checkpointing technique that we present in this thesis, *Speculative Memory Checkointing* (SMC). While SMC allows applications to efficiently access the kernel's CoW mechanism, it also features a speculation component, which by speculatively copying hot pages aims to avoid the eight times higher cost of kernel CoW over plain copying. As the speculation essentially consists of Writable Working Set Estimation (WWSE), we evaluated two established WWSE techniques—Active-RND and Active-CKS—and our novel GSpec WWSE algorithm. In contrast to the traditional WWSE techniques, GSpec's approach can automatically tune to different workloads.

Our experiments show that a hypothetically perfect and overhead-free speculation can reduce the overhead of CoW checkpointing by 29.3 percentage points from 44.9% to 15.6% in the tested scenario. This confirms the usefulness of speculation to reduce COW induced checkpointing overhead. Further, we show that all tested speculation strategies improve the performance overhead while leading to a modest increase of memory consumption and recovery time. GSpec exhibits the highest speculation accuracy across the tested applications and reduced checkpointing overhead by 14.1 percentage points from 44.9% to 30.8%, compared to plain COW.

In Chapter 4, we presented *DeLorean*, a debugging system leveraging checkpoint-assisted time-traveling introspection, as a real-world use case of high frequency checkpointing. DeLorean emphasizes the importance of efficient checkpoint storage and exploration, two important problems that arise when many checkpoints are taken with high frequency. DeLorean uses the checkpointing component presented in Chapter 3 to achieve efficient high frequency checkpointing to reduce the run-time overhead during the recording phase. By compressing and deduplicating checkpointed pages, DeLorean is capable of reducing the memory footprint of the checkpointing storage by 95%, making it possible to store millions of checkpoints. Further, we explore different ways to inspect the checkpoint history. We show that when knowing memory locations of interest, partial on-demand rollback can save up to 88% of search time in our experiments. This can be further improved if certain properties of the search criteria are known, which in some cases allows us to use use a bisect search strategy, effectively enabling us to search one million checkpoints in around 8.5 seconds.

In summary, this thesis shows the need for specialized high frequency memory checkpointing techniques and proposes enhancements to current checkpointing techniques to make them fit for use cases that require checkpoints to be taken with a high frequency. We explored the deployability trade-off of different checkpointing techniques and showed that if the target application can be recompiled, pure userland techniques relying on compiler instrumentation can offer a significantly better run-time performance than their page-granular counterparts. LMC's shadow state inspired checkpoint organization shows that it is not necessary to give up memory guarantees to achieve this. Further, we showed that speculation and exporting copy-on-write functionality as a first level kernel primitive to the userland lowers the overhead of page granular checkpointing significantly. Finally, we explore techniques that allow for efficiently storing and searching a the large number of checkpointed data resulting from the high checkpoint frequency.

We hope that these techniques combined open up a whole new range of possible scenarios for high frequency memory checkpointing, such as the time traveling debugging system DeLorean presented in this thesis.

SAMENVATTING

Efficiënte Hoogfrequente Momentopnames ten bate van Foutherstel en Foutanalyse

Computers en de programmatuur die er op draait spelen in ons dagelijkse leven een immer toenemende rol. Deze programmatuur, ook software genaamd, wordt continue uitgevoerd op computers, telefoons, TV's, en zelfs om onze polsen in slimme horloges. Deze software heeft weliswaar onze levensstandaard op meerdere manieren verhoogd, maar wordt ook al sinds het eerste uur geplaagd door een probleem: betrouwbare software is moeilijk te schrijven. Dit blijkt mede uit de gevleugelde uitdrukking *"Have you tried turning it off and on again"*, oftewel, heb je hem al uit- en aangezet?, de zogenaamd universele oplossing voor allerlei computerproblemen, die dus allemaal aan de software moeten liggen.

Het is dus niet verrassend dat onderzoek naar methodes en technieken om software *betrouwbaar* te maken in dezelfde periode een vlucht nam dat computers voor het eerst werden toegepast in het bedrijfsleven en in universiteiten.

Checkpointing, is een algemene systeem momentopname, is een belangrijke techniek die uit dit onderzoek is voortgekomen, en veel toepassingen heeft t.b.v. betrouwbaarheid, zoals het automatisch herstellen van fouten die opgetreden hebben, en het vinden van fouten in de software. Een belangrijk onderdeel van deze momentopname is een momentopname van het geheugen van een programma maken. Dat is het hoofdonderwerp van dit proefschrift.

Dit proefschrift legt zich met name toe op scenarios waarbij *frequente momentopnames* nodig zijn. Voorbeelden van toepassingen zijn automatische herstel technieken die momentopnames voor elke cliënt verzoek vereisen, of op precies uitgekiende herstel-tijdstippen. Verder maken frequente opnames het ook mogelijk om, bij het opsporen van fouten, de toestand van een programma op elk gewenst moment tijdens het verloop van het programma op

te vragen.

Voor het hoogfrequente uitvoeren van deze opnames is het vastleggen van de toestand van het geheugen een belangrijke factor in de kosten. Daarom is het aantrekkelijk dit te optimaliseren. Toepassingen die zowel hoogfrequente opnames nodig hebben als een lange opname geschiedenis krijgen ook te maken met het efficiënt opslaan van al die opnames zonder dat waardevolle applicatie geheugen kwijt te raken.

In Hoofdstuk 2 onderzoeken we pure applicatieniveau hoogfrequente opnames. Omdat we ons tot het applicatieniveau beperken, omzeilen we het feit dat bij het in gebruik nemen van een dergelijke oplossing in de echte wereld, een verandering aan het onderliggende besturingssysteem vaak niet mogelijk is. We onderzoeken drie momentopname-systemen die op pagina-resolutie werken, en verder een systeem waar instrumentatie voor nodig is, toegepast op een verzameling server applicaties. Om een hoogfrequente opname aan te tonen, nemen we voor elk verzoek een opname. Dit is een veel voorkomende modus wanneer we verzoek-gebaseerd foutherstel willen doen. Onze experimenten tonen aan dat wanneer een hogere opnamefrequentie wordt toegepast, de opnamesystemen die een paginaresolutie gebruiken een hoger snelheidsverlies laten zien. De instrumentatie-gebaseerde *undolog*, die weliswaar efficiënter is in gebruik, heeft als nadeel dat het geheugengebruik onbegrensd is. We presenteren *Lightweight Memory Checkpointing* (LMC), een systeem dat op efficiënte wijze hoogfrequente, applicatieniveau momentopnames met wél begrensd geheugengebruik mogelijk maakt. LMC maakt gebruik van instrumentatie die bij het bouwen (compileren) van de applicatie automatisch door de compiler wordt aangebracht, in plaats van afhankelijk te zijn van relatief inefficiënte systeemniveau primitieven die met paginaresolutie werkt, en met *copy-on-write* semantiek werkt. In tegenstelling tot de *undolog* aanpak slaat LMC opname gegevens op in een schaduwbestand, waardoor geheugengebruik begrensd wordt.

In Hoofdstuk 3 komen we terug op paginaresolutie momentopnames. Maar deze keer staan we er niet op dat systeemwijzigingen niet kunnen. Eerst analyseren we wat de kosten zijn voor deze momentopnames met pagina resolutie. We tonen aan dat als we gebruik maken van het Kopiëer-met-Schrijven (*copy-on-write*, oftewel CoW) mechanisme in Linux, dat met `fork` automatisch wordt ingezet, het maar liefst acht maal zoveel tijd kost ten opzichte van simpelweg de pagina kopiëren. En daarmee negeren we nog de kosten van `fork` zelf, die ook aanmerkelijk zijn. De resultaten van deze analyse motiveren het ontwerp van de tweede techniek om momentopnames mee te maken die in dit proefschrift wordt gepresenteerd, Speculatieve Geheugen-

opnames, in het Engels genaamd *Speculative Memory Checkpointing* (SMC). SMC maakt een combinatie van CoW en simpelweg direct kopiëren mogelijk. SMC kan ook er notie van nemen welke pagina's vaak veranderen, oftewel 'heet' zijn, en deze speculatief kopiëren, om een veel duurdere CoW gebeurtenis te snel af te zijn. De speculatie bestaat uit het inschatten van wat telkens de set pagina's is waarmee gewerkt wordt, en ook nog eens naartoe geschreven wordt. Dit noemen we *Writable Working Set Estimation*, oftewel WWSE. We hebben zowel twee gevestigde WWSE technieken geëvalueerd, namelijk Active-RND en Active-CKS, als ons eigen WWSE algoritme, namelijk GSPec WWSE. In tegenstelling tot traditionele WWSE technieken, kan GSpec zich automatisch op verschillende soorten taken instellen. Onze experimenten tonen aan dat een (hypothetische) ideale en kosteloze speculatie de kosten van CoW momentopnames met 29,3 procentpunten kan verminderen - van 44,9% tot 15,6%, in het geteste scenario. Dit bevestigt het nut van speculatie om de kosten van het gebruik van CoW bij momentopnames te verminderen. Verder laten we zien dat alle geteste speculatie strategieën de kosten drukken, en ook een beperkte toename in het geheugengebruik en hersteltijd met zich mee brengen. GSPec laat de grootste speculatie nauwkeurigheid zien bij de geteste toepassingen, en vermindert de opname kosten met 14,1 procentpunten, van 44,9% naar 30,8%, in vergelijking tot standaard CoW.

In Hoofdstuk 4 presenteren we *DeLorean*, en systeem om fouten in software te helpen opsporen (*debugging*). Dit systeem maakt gebruik van momentopnames waardoor 'tijdreizen' mogelijk wordt. Hiermee demonstreren we een toepassing in de echte wereld voor hoogfrequente momentopnames. DeLorean benadrukt het belang van efficiënt onderzoek en opslag van momentopnames, twee belangrijke vraagstukken die zich voordoen wanneer veel momentopnames genomen worden. DeLorean gebruikt het systeem dat in Hoofdstuk 3 is gepresenteerd om efficiënte, hoogfrequente momentopnames te realiseren, en zo de vertraging tijdens de opnamefase te verminderen. Door het comprimeren en ontdubbelen van de pagina's in de momentopnames, is DeLorean in staat om het geheugengebruik van de opslag van de momentopnames met 95% te verminderen. Hierdoor is de opslag van miljoenen momentopnames mogelijk. Verder onderzoeken we verschillende manieren om de inhoud te onderzoeken. We laten zien dat, in onze experimenten, als interessante geheugenlocaties bekend zijn, gedeeltelijke terugrol tot 88% van de zoektijd kan schelen. Dit kan verder verbeterd worden als bepaalde eigenschappen van de zoekcriteria bekend zijn. In sommige gevallen kunnen we daardoor een zoekstrategie gebruiken door telkens middendoor te delen –

hierdoor kunnen we miljoenen opnames in rond de 8,4 seconden doorzoeken.

Samengevat laat dit proefschrift de noodzaak zien voor hoogfrequente geheugen momentopnames, en worden er verbeteringen voorgesteld aan gevestigde opname technieken. Hierdoor worden ze bruikbaar voor hoogfrequente toepassingen. We hebben het praktische inzetbaarheids spanningsveld onderzocht van verschillende opnametechnieken, en hebben laten zien dat als de doel-applicatie opnieuw gecompileerd kan worden, technieken die puur op applicatieniveau werken een veel betere doorvoersnelheid van de applicatie mogelijk maken dan varianten die van paginaresolutie technieken afhankelijk zijn. Het schaduwbestand van LMC laat zien dat het niet noodzakelijk is om geheugen garanties te laten varen om dit te realiseren. Verder meer laten we zien dat speculatie en het ontsluiten van de CoW functionaliteit als een systeem functie aan de applicatie de extra kosten van hoogfrequente opnames aanzienlijk verkleint. De technieken die in dit proefschrift worden gepresenteerd maken het efficiënte opslaan en doorzoeken van het grote aantal opnames dat van een hoogfrequente opname afkomt mogelijk.

We hopen dat deze technieken gezamenlijk een heel nieuw scala aan mogelijke toepassingsscenario's mogelijk maakt voor hoogfrequente geheugen moment opnames, zoals het 'tijd-reizende' DeLorean systeem dat ook in dit proefschrift gepresenteerd wordt.