Compiler Support for Fine-grain Software-only Checkpointing

by

Chuck (Cheng Yan) Zhao

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Compiler Support for Fine-grain Software-only Checkpointing

Chuck (Cheng Yan) Zhao

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2013

Checkpointing support allows program execution to roll-back to an earlier program position, discarding any modifications made since that point. Existing software-based checkpointing methods are mainly library-based, snapshot a program's entire working memory, and hence have prohibitive overhead for many potential applications. This dissertation proposes a lightweight and fine-grain checkpointing framework implemented entirely in software, through compiler transformations and optimizations. In our framework, the programmer can specify arbitrary checkpoint regions via a simple API, and the compiler automatically transforms the code to enable checkpointing and optimizes for checkpointing overhead reduction.

Our fine-grain software-only checkpointing is based on compiler instrumentation to *back-up* program-state changes on a per-store basis within user-annotated program regions. An individual backup action is performed for a given memory location only if there is a corresponding write to that memory location, thus our checkpointing scheme naturally adapts to the program's behavior. This scheme respects user-inserted program annotations for program partitioning into checkpoint regions, and supports regions containing complex program constructs such as function callsites, function pointers, and recursions. Our comprehensive compiler optimization framework employs aggressive compiler optimization algorithms and conducts program transformations to minimize checkpointing overhead. The compiler transformation and optimization infrastructure is sufficiently robust to enable user-level checkpointing over large real-world applications and preserves the correctness of the original program even under pathological corner cases.

We explore three application areas for our fine-grain checkpointing support. First, we utilize our efficient software-only checkpointing to support overlapping execution with delinquent loads through

the prototyping and evaluation of both control-speculation and data-speculation transformations. We further propose a theoretical timing model and confirm its effectiveness with a real-machine workload. Second, we investigate its application to debugging, in particular by providing the ability for a debugger to rewind to an arbitrarily-placed point within the execution of a buggy program. A study using BugBench applications shows that our compiler-based fine-grain checkpointing has more than a factor of 100 less overhead than full-process, unoptimized checkpointing. Finally, we demonstrate that compiler-based checkpointing support can be leveraged to free the programmer from manually implementing and maintaining detailed software rollback mechanisms when coding a backtracking algorithm for a realistic CAD application, with runtime overhead of only 15% compared to the manual implementation.

Acknowledgements

I would like to deeply thank my advisors—Professor Greg Steffan and Professor Cristiana Amza for all the guidance to me over the years of my Ph.D. journey. Our weekly meetings are a critical component in directing and guiding me over my research. I appreciate the time and occasions we meet, think hard and deep into the difficult questions, looking for potential solutions from the seemingly impossibles. Thanks for also the numerous paper draft writings and revisions, practice talks and rehearsals.

My committee including Professor Abdelrahman and Professor Moshovos provided useful insights, suggestions and feedbacks for filling out this work.

I want to deeply thank my parents—my mom and dad. Although they live in a different continent that is 12 hours ahead of my time in Toronto, we have daily email/phone/chat communications. It is their love and support that makes me persistent. It is their encouragement that helps me sustain and successfully reach the end of the extremely long and seemingly for-ever lasting journey. It is especially valuable and important when I was facing the darkest period of my life—going through divorce.

Throughout the 9 years at University of Toronto, I spend 6 in EA-305 with the PACRAT group. I want to thank all my colleagues and friends for helping me through the difficult process: deep discussions of both academic and non-academic issues, friendship and couragement to face both daily challenge and dynamic future. My colleagues and friends: Mihai Burcea, Diego Huang, Yi Wang, Alex Chooong, Nick Ni, Zefu Dai, Xin Tong, Davor Capalija. Too many names that I can't fully list here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Checkpointing is the process of taking a program snapshot so that program state can be safely and precisely restored to this snapshot should any error, failure, or misspeculation occur during future execution. Depending on available hardware support, conventional checkpointing methods include both software and hardware solutions, with hardware support being rare and only available on high-end enterprise-class proprietary systems. Most existing software checkpointing solutions back-up program changes at a coarse granularity by copying either the entire process's memory or at least many objects. While software instrumentation for program checkpointing is not new, the applicability of such checkpointing techniques is limited by prohibitive software overheads.

In this dissertation we design, implement, and experimentally evaluate novel compiler-driven techniques for program instrumentation and overhead optimization of fine-grain software checkpointing. We show that we can significantly reduce software checkpointing overhead by conducting checkpointing at a fine granularity—i.e., on a per-memory-store-operation basis. We show that our fine-grain checkpointing scheme provides ample opportunities for program optimizations and that our checkpointing optimization framework can effectively eliminate up to 98% of software checkpointing overhead. With the support of our efficient and compiler-driven checkpointing framework, we further explore the possibility and feasibility of enabling checkpointing on performance-sensitive applications where checkpointing was previously considered impossible and impractical. We show that checkpointing-enabled applications enjoy unique features and properties that can improve programmer productivity.

1.1 Checkpointing

Checkpointing [20, 35, 39, 48, 56, 61, 62] is a technique to back-up program state such that execution can later revert to the back-up and recover from unpredictable program failures. While proposed hardware-based checkpointing solutions [5, 31] show promising performance, they are limited by the available hardware buffer space. More importantly, hardware support for checkpointing is normally not available in commodity systems. Software-based checkpointing solutions [35, 39, 50, 62] can be used on commodity hardware, thus they are immediately more applicable and affordable. However, they normally come with prohibitive software overhead as they are typically coarse-grained, meaning that they back-up large ranges of memory or even the entire process image.

In this dissertation, we present a software-only method for checkpointing program execution that is implemented in a compiler. In particular, our transformations implement checkpointing at the finest level of granularity—individual variables, as opposed to previous work that checkpoints entire ranges of memory or entire objects [5, 20, 35, 50]. The intuition is that such fine-grain checkpointing can (i) be a better fit for the nature of many applications since it checkpoints *only* the memory that is changed, (ii) provide many opportunities for optimizations that reduce redundancy and increase efficiency, and (iii) facilitate uses of checkpointing that demand minimal overhead. We present a complete checkpointing framework and optimization infrastructure that can (i) enable software-only checkpointing over arbitrarily large and complex program regions and (ii) leverage compiler optimizations to reduce overhead. We show that our fine-grain scheme is more efficient than coarse-grain approaches, and that up to 98% of checkpoint buffer space and up to 95% of `backup` memory service calls can be eliminated.

1.2 Applications of Checkpointing

We demonstrate the usability of our compiler-based checkpointing infrastructure via three key applications that leverage this support. The first application is to support speculative execution overlapping with delinquent loads, where a compiler helps to schedule work that can potentially overlap with the long cycles of last-level cache misses. We conduct an in-depth study using a delinquent-load intensive application (`MCF`). Depending on different locations where delinquent loads may reside, we present both control speculation and data speculation as compiler transformation schemes to tolerate the long cache-miss latency caused by the delinquent load. As part of a complete software speculation model, we study various types of value predictions and propose a mathematical timing model that matches well

with timing behaviors of real-machine applications. This application is a limit-test for the applicability of our lightweight checkpointing technique. Unfortunately we find that for this case the opportunities for speculative optimization are too rare and fine-grained to amortize the checkpointing and prediction overhead, and hence cannot demonstrate speedup.

The second application is support for debugging, in particular by giving a programmer the ability to roll-back execution to repeatedly examine the state of a program prior to the manifestation of a bug. We study several flawed applications from the BugBench [41] suite and demonstrate the low overhead of checkpointing support for rollback.

The third application is support for backtracking algorithms, where a programmer can avoid implementing manual support for rewinding data-structures, by leveraging compiler-based checkpointing to provide it automatically instead. We study VPR [9, 54], in particular the simulated-annealing-based place-and-route algorithm for FPGAs, which optimistically swaps blocks and either keeps or discards the swap depending on whether a cost function improves. By comparing the original manual implementation of backtracking support to our automatic compiler-based approach, we demonstrate the benefits that our automatic checkpointing infrastructure can provide.

1.3 Research Goals

In this dissertation, we plan to explore the techniques to enable fine-grain software-only checkpointing and aggressively reduce checkpointing overhead by exploring and exercising compiler techniques. In particular, we will focus on introducing compiler transformations to enable checkpointing, adopting existing compiler optimizations to make it effective for checkpointing, and inventing new checkpoint-specific compiler optimizations. We strive for the following research goals.

1. **Enable checkpointing over arbitrary program regions.**

   We will investigate the process that enables fine-grain and software-only checkpointing over programmer-annotated regions of arbitrary size and complexity. In addition to introducing general schemes to enable checkpointing on common scenarios from a region with potentially unbounded size, we will also discuss detailed steps to handle pathological corner cases to make our checkpointing scheme complete.

2. **Improve checkpointing performance through compiler optimizations.**

   We will explore software-only techniques to improve checkpointing performance by employing

compiler optimizations to aggressively reduce checkpointing overhead. In particular, we will focus on evaluating the effectiveness of existing compiler optimizations in the software-only checkpointing domain, and inventing new compiler optimizations that are specially designed to harness the new optimization opportunities that are only exposed through the software-only process of enabling fine-grain checkpointing. Since we have the complete knowledge of the entire application (including the checkpoint-enabled program region), we expect significant performance improvements over alternative coarse-grain solutions.

3. **Support key applications.**

   Leveraging our efficient software-only checkpointing framework, we attempt to enable checkpointing service on three key applications. A checkpoint-enabled application will either require new functionality that is otherwise impossible or impractical, or obtain improvements that will ultimately benefit programmers who choose to adopt them. We plan to explore two coarse-grain applications: checkpoint-enabled debugger, and checkpoint-support for automatic software backtracking using `VPR`. Even more aggressively, we will attempt a high-risk case that uses an application-level technique (software-only checkpointing) to explore speculative optimization related to a micro architecture-level feature (delinquent loads).

4. **Enable future research.**

   Our compiler-based checkpointing infrastructure has become stable, mature and is capable of handling large user applications. We are publicly releasing the entire source code package (all compiler passes, compiler driver, complete sanity testing harness, complete source codes of checkpointing-enabled applications, tips for effectively building compiler passes for transforming and optimizing large-scale applications, suggestions on solid and bug-free software development practice, etc.), to encourage future benefit from the framework.

1.4 Thesis Limitations

Any academic work has limitations and our work is not an exception. In this thesis, we explore a user-level checkpointing approach that can only checkpoint programs whose source codes are available—for situations where source code is unavailable, such as operating system services and library calls, we provide a mechanism for the user to describe the impact on checkpointing of the service/call. When such outlining is not possible, checkpointing cannot be supported. We design our checkpointing buffer

to favour run-time performance over storage efficiency, and hence do not directly pursue methods of purely reducing storage requirements at the cost of performance. Finally, our checkpointing method can only support single-threaded applications having a single outstanding checkpoint at any given time; we have not yet considered support checkpointing within a multi-threaded environment, checkpoints with multiple rollback points, nor support for nested checkpoints. We do not support rollback to a checkpoint that has already been committed.

1.5 Thesis Organization

The rest of the thesis is organized as following. In chapter 2, we conduct a thorough review of existing background work related to software-only checkpointing, deterministic re-execution, compiler optimization, and other closely related areas. In chapter 3, we discuss the detailed analysis and transformations to enable software-only checkpointing over any user-annotated program regions. In chapter 4, we present full details of existing checkpointing-specific compiler optimizations that target maximum overhead reduction through aggressive compile-time analysis and transformations. In chapter 5, we show detailed analysis of the checkpointing's buffer schemes: the selection between undo-log and write-buffer schemes, various undo-log buffer implementations, as well as performance-storage tradeoff comparisons. In chapter 6, we introduce the *first* key checkpoint-enabled application that attempts to overlap program execution with delinquent loads (DLs). In addition to identifying DL locations, we conduct both theoretical modeling and practical evaluation for checkpointing over fine-grain DL's granularity. In chapter 7, we illustrate *two* additional key applications that benefit directly from our low-overhead software-only checkpointing infrastructure. These applications either gain novel functionality that won't exist otherwise, or ultimately benefit users through improved programmer's productivity. We finally conclude in chapter 8 by summarizing the entire thesis and give suggestions on potential future research directions.

Chapter 2

Background

Checkpointing [21, 35, 48, 56, 62] is the process of taking a snapshot of program execution, so that if there is a subsequent program failure, error, misprediction, or other unwanted event, the program can be rewound to the snapshot to recover and start over. Checkpointing is often provided as a low-level service in high-end computer systems, supported by hardware or the operating system itself—to ensure high reliability and availability, providing security enhancement and improving fail-over support.

The techniques presented in this dissertation leverage prior work in related areas that are summarized in this chapter, including program checkpointing, compiler optimizations for checkpointing, thread-level speculation (*TLS*), transactional memory (*TM*), deterministic re-execution, and program backtracking—especially when used for software debugging.

2.1 Checkpointing Basics

We provide an example to illustrate the checkpointing process in Figure 2.1. A program's sequential execution is partitioned to have pre-defined checkpoints at designated program locations (from `P1` to `P5`). At each of these locations, the program can either commit the current checkpoint and immediately start a new one upon successfully completing the current stage without any error, or rollback the existing checkpoint and restart it after recovering an abnormal condition. The program starts its normal execution at program location `P1` along the path of `1.normal execution`. An unexpected error occurs at runtime. At checkpoint `P3`, this error is detected and the program needs to recover from it. The checkpointing recovery process is activated to safely rewind execution back to its latest start-checkpoint location along the path `2.checkpoint recovery`. This process restores the memory changes captured in the checkpointing buffer. Upon completion of the checkpointing recovery, the program resumes its normal

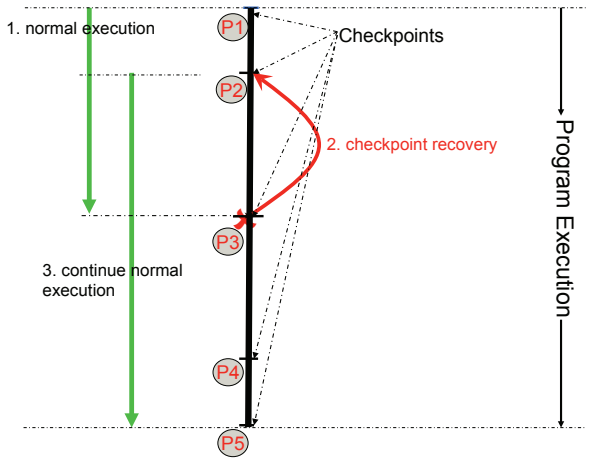


Figure 2.1: Overview of Checkpointing

execution from position `P2` as if it arrives at this location for the first time without any error. Execution continues as normal after checkpoint recovery along the path of `3.continue normal execution`. The same checkpointing process repeats for each additional program failure captured throughout the rest of the execution. Between consecutive checkpoints, checkpointing needs to preserve program states in preparation for potential program failures and recovery. This involves copying memory at runtime that results in overhead if the program behaves normally, and implementing checkpointing optimization methods that minimize this overhead.

Prior research has investigated hardware-based checkpointing [5, 31, 46] and software-only checkpointing [11, 12, 35, 39, 50, 62]. Hardware-based schemes rely on hardware support to buffer checkpointing states and facilitate recovery. However, checkpointing hardware has limited resources (usually limited buffer size) and is often not available in commodity platforms. This not only restricts the applicable range of code that can be effectively checkpointed, but also puts the long term prospects of hardware schemes under question. Software-only checkpointing schemes exhibit flexibility by removing the dependence on checkpointing hardware, and thus immediately becomes more applicable on a wider range of systems that demand checkpointing service. However, existing software-only checkpointing schemes suffer from prohibitive overhead by copying a large amount of memory at runtime. These schemes often employ heavy-weight checkpointing processes that operate at the application level [11, 12], operating system level [34, 39, 50] or virtual-machine level [62]. They focus on providing software checkpointing service to enhance system robustness and reliability, thus

performance of checkpoint-enabled applications is not the primary concern. Coarse-grain software-only checkpointing introduces prohibitive overhead. The range of quantified overhead is between 15 times slower to over $10,000$ times slower [40]. As a result, coarse-grain software-only checkpointing is not suitable for performance-critical applications that have real-time responsiveness requirements.

In contrast to existing approaches, we propose a lightweight and software-only checkpointing scheme that will work on fine granularity. It heavily leverages the compiler to aggressively optimize checkpointing performance, resulting in a checkpointing implementation that is more suitable for performance-sensitive applications.

2.2 Hardware-based Checkpointing

Hardware-based checkpointing implementations provide native hardware support for basic checkpointing services. There are various schemes available under this category, ranging from using checkpointing to repair computing engines [31], using checkpointing to reduce ReOrder Buffer (*ROB*) overhead [5], and using hardware checkpointing to improve system efficiency [46].

Hwu et al. [31] proposed a hardware solution to checkpoint computing engines to known good states, and use it later to recover from a program's ill-behaviors caused by exceptions and branch mispredictions. Their approach presented theoretical predictions and proved numerous theorems dealing with maximum estimated checkpoint buffer size, algorithms to perform the exact actions, and even high-level hardware design diagrams ready for implementation. However, this approach did not estimate the extra hardware cost as well as its related implementation complexity. Although the authors mentioned that the performance impact cannot be precisely proved, it still lacks performance analysis and comparison when their reference designs were to be implemented either in real hardware or inside a simulator.

Modern CPU designs employ two critical techniques to maintain high performance: (i) out-of-order execution to exploit instruction-level parallelism (ILP), and (ii) reorder buffer (ROB) to guarantee correctness. ROB requires a linear FIFO structure to guarantee its instruction's commit sequence. This puts challenges on various design concerns, including rename table entry, instruction retirement control, and register reclamation. *Akkary et al.* [5] proposed a hardware-based checkpoint scheme to alleviate major ROB overhead. Their scheme only generates checkpoints at branch instructions that have high likelihood of misprediction. Every instruction in the program will need to map to a particular checkpoint, increase its corresponding counter when it is being executed and decrease the counter after

it retires. This guarantees a global total ordering for committing checkpoints in the right sequence. The checkpoint will not commit until all instructions registered with it have committed. In addition, each memory instruction needs to have extra counters that refer to the instruction's uses. This poses a relatively large ISA change and hardware complexity. This scheme also proposed various hardware changes for increasing ROB performance through checkpointing. However, many of the proposed features need to significantly modify existing architecture, which makes it prohibitive. They did not address the design complexity, nor potential cost and performance impact when implementing the design.

When an individual checkpoint needs to cover a huge number of instructions (e.g., hundreds millions or even billions of instructions), the size of the checkpoint becomes a major storage and efficiency bottleneck. *Moshovos et al.* [46] proposed a novel checkpoint compression scheme for giga-scale, coarse-grain checkpoint/restore. They observe value locality from both in-memory data and address streams and the effectiveness of their compression is based on the exploitation of such locality. Based on previously proposed dictionary-based hardware compressors that are both expensive and slow, they observe that simpler mechanisms can offer similar compression benefits with a much smaller and simpler direct mapped structure. The newly proposed compressor requires few resources and can easily be pipelined to achieve one full block compression per processor cycle. When used alone, it can reduce checkpoint storage to 52% of its original size. When combined with previously proposed hardware-based compression, it improves overall compression rates while significantly reducing on-chip buffer requirements. Even with hardware's assistance to compress checkpointing, the checkpointing process is nevertheless coarse grain with significant overhead.

All hardware-based checkpointing schemes rely on available checkpointing hardware to conduct their designed activity. On one hand, this improves necessary execution efficiency by providing native hardware support. On the other hand, the amount of available support is strictly limited by the dedicated checkpointing hardware that might not be available for commodity servers. This restricts the applicable domains that the checkpointing service can apply. Overcoming such constraints leads to the design of software-only checkpointing.

2.3 Software-Only Checkpointing

Due to problems of checkpointing hardware's limitations and availability, software-only checkpointing schemes are much more popular than hardware-based checkpointing. *Plank et al.* [34, 50] presented a

user-directed checkpointing approach for manually inserting checkpointing directives at programmer-designated source-code locations. The compiler processes the annotated code, and converts directives into checkpointing library calls. The checkpointing library interface includes calls to identify inclusive memory regions, exclusive memory regions, and directives to force a checkpoint at certain program locations. They developed techniques using static data-flow analysis to optimize sizes of the checkpointing regions. Their contribution is a detailed backward all-path data-flow algorithm, which incorporates a normal control-flow analysis to build the programs call graph (*CG*), partitions the CG into sub CGs (*CG'*) where each *CG'* includes only one single force-checkpointing directive at its beginning, builds relevant sets on each statement inside the CG, inserts memory-inclusive and memory-exclusive calls, and iterates until the process finally stabilizes. This approach presents a compiler analysis leading to refining coarse-grain checkpoint region partitions, so that each incremental checkpoint will only `backup` chunks of memory that are changed between two consecutive checkpoints. However, their scheme is still coarse grain whose granularity largely depends on the programmer's intuitive understanding of the program's checkpointing region. Although optimizations can reduce overhead, this scheme does not provide performance comparison that measures checkpointing efficiency against previous approaches. In addition, this approach only operates on FORTRAN programs, which are known to have more regularity and less un-predictable dependencies.

Li et al. [39] proposed Compiler Assisted Techniques for CHeckpointing (*CATCH*)—a full program checkpointing solution that can automatically identify checkpoint locations. The CATCH scheme includes a CATCH filter (compiler passes) and a CATCH runtime library, both are built into a modified version of GCC. CATCH identifies potential checkpoint locations at either the beginning of a procedure or the first statement inside a loop. For each potential location, the system calculates its current checkpointing interval and evaluates costs. When a set of pre-defined conditions hold on a potential checkpointing location, an actual checkpoint is marked at the location, where the entire memory footprint of the running process is written into a disk file. Not satisfied with poor initial performance, the authors then aggressively apply optimization techniques. This includes checkpoint file compression, adaptive checkpointing, profile-guided optimization, etc. The optimizations effectively reduce checkpoint time from 212 sec/ckpt to 30 sec/ckpt, and reduce checkpointing overhead from 78.52% to 20.74%, respectively. The CATCH approach shows that compiler optimization is an effective tool to reduce checkponting overhead. CATCH works efficiently for coarse-grained checkpointing that focuses on reliability enhancement and fault tolerance support. However, due to the long latency and large amount of memory that needs to checkpoint, this approach is not suitable for applications with

critical real-time performance requirements.

Whaley [62] introduced a complete virtual-machine level checkpointing scheme using Java reflection and program analysis. Java reflection marks all necessary fields inside each relevant object. When the object is removed from memory, all fields marked necessary are written into permanent storage on disk, so that they can recover thoroughly and safely from disk when the object later needs to be reconstructed. The scheme performs a simple flow-insensitive and context-insensitive pointer analysis, so that every object in the virtual machine and each field in the object are marked necessary whenever there is a reaching path from the beginning of the program to the object or field. The checkpointing scheme stops the program when its execution reaches its main function, builds the objects reaching set using the program analysis, writes all relevant memory into a disk file, and then continues normal execution. Each subsequent program launching is equivalent to a checkpoint recovery which essentially reads the disk file and populates its contents to memory. After the initial overhead of conducting program analysis and checkpoint building, Whaley's checkpointing approach shows improved memory usage and speed up in programs launching (startup) time. It is especially useful for early JVMs that have significant application startup delays. However, there is no evaluation for the cost of checkpointing on both the checkpoint file size and checkpointing's performance impact on the original application. This approach concentrates on virtual-machine level coarse-grain single-application checkpointing. It lacks support for multiple checkpoints and checkpointing under finer granularity.

Due to changes in data set size and improvements in algorithms, a program's average running time has become longer than normal hardware's Mean Time Between Failures (*MTBF*). This implies that certain hardware components are guaranteed to fail before a program runs to its completion. A solution needs to protect such long-running programs from hardware failures and minimize service interruptions. *Bronevetsky et al.* [11, 12] proposed using compiler techniques to implement an application-level, non-blocking, coordinated global checkpointing solution for MPI-based programs on distributed-memory parallel computing clusters. The source program is preprocessed by a compiler where manually annotated checkpoint macros are converted into checkpoint library calls. This separates the implementation of the checkpointing library from the MPI library. A redesigned protocol exchanges messages on request to build checkpoints for each individual MPI process and collaborates with the rest of the MPI protocol. The solution focuses on the design and implementation of an enhanced protocol on top of the MPI layer, guarantees consistency, and handles different types of MPI messages properly. However, the major problem is the excessive checkpointing overhead. In addition, this approach works over coarse-grain process isolation through manual annotation of program's checkpointing locations.

In contrast to existing software checkpointing approaches that work on coarse granularity and incur prohibitive overhead, we propose a lightweight checkpointing scheme that will work on fine granularity. For performance-sensitive applications, tracking program changes on page, object, cache-line or array-level of granularity is still overly coarse grain. We will explore finer checkpointing granularity by tracking on a per-store basis through source-code instrumentation. We believe this is relatively the finest possible tracking granularity that a user-guided programming tool can perform on source-program level and is a field that has not been fully explored previously. The implications, difficulties and benefits of such fine-grain tracking is largely unknown to the research community.

2.4 Compiler Optimizations for Checkpointing

A compiler [3, 4, 6] is a computer program that converts program descriptions between different levels of abstractions. Often it translates a program from a high-level source-code format into a low-level machine-code description suitable for executing on a particular target platform. A compiler's traditional focus [3, 4] is automata-based parsing and semantic analysis (`front end`) that handles different language syntax and completes the translation process. More recently, compilers [6] have become more focused on improving code quality and application performance by exploring, identifying and reducing redundancy in application codes through program restructuring (`back end`).

Recent work [11, 12, 14, 51, 52] show that compiler optimizations are important in reducing checkpointing overhead. *Plank et al.* [51, 52] pioneered using a compiler to identify memory regions that are not changed between consecutive checkpoints and excluding such regions from incremental checkpointing. *Choi et al.* [14] noticed that normally only a small number of memory pages are modified (*DIRTY* pages) between nearby checkpoints and checkpointing can instead save only the DIRTY pages. This reduces full-process memory checkpointing into checkpoint only the memory pages that are modified between consecutive checkpoints. *Bronevetsky et al.* [11, 12] used a compiler to analyze ranges of an array that are modified between consecutive checkpoints. They developed compiler optimizations that identify first-write and last-write array indexes, and checkpoint only the modified range of the array.

Comparing with existing work, we plan to expand the use of compilers as the critical tool to further advance software-only checkpointing. We plan to explore finer checkpointing granularity by lowering the memory tracking unit from page, memory object or array level to per-store level—a degree of fine-grain checkpointing that hasn't been explored. A compiler will continue play a key role to both enable

checkpointing through compiler transformations and reduce checkpointing overhead through compiler optimizations. We will design, implement and evaluate novel compiler optimizations that particularly target overhead costs and opportunities brought by the fine-grain checkpointing scheme.

2.5 Systems and Mechanisms that Leverage Checkpointing

Other checkpointing-related systems and mechanisms exist. They leverage the checkpointing principle for different design goals or functionalities. They include deterministic re-execution, speculation, and software backtracking.

2.5.1 Deterministic Re-Execution

Deterministic re-execution allows a multi-threaded program to restore to a previous program location and re-execute it under a deterministic order. One of the frequent uses of deterministic re-execution is to debug or profile parallel applications. A complete deterministic re-execution scheme includes two components: (i) checkpointing of a parallel application and (ii) deterministic replay. We focus our discussion more on the parallel checkpointing portion of deterministic re-execution.

Feldman et al. [21] present *IGOR*—a system capable of conducting full-process checkpointing, and optimize it to incrementally checkpoint only dirty pages. Its deterministic re-execution is coarse-grain and only precise up to the nearest program location where a checkpoint is taken. IGOR has high software overhead: checkpointing causes applications to slowdown within the range of 40% to 370%, depending on checkpointing frequency.

King's time-traveling virtual machine (*TTVM*) [33] discusses an OS-level debugging facility by checkpointing entire OS states into disk files. This include CPU state, complete virtual memory and virtual disks, multiple user applications, and all system services. Coupled with light-weight fine-grain forward replays, TTVM's deterministic re-execution can reach any precise program location between two consecutive full checkpoints. Designed as an OS-level debugging tool, its overhead is also high. TTVM reports 7% runtime overhead with 25 MB/sec disk-write throughput when conducting one full VM checkpoint every 10 seconds.

Xu et al. [64] demonstrate a re-tracing tool (*ReTrace*) that is built on top of VMWare's deterministic replay technique to collect only non-deterministic events during program execution and later expanding the event collection into full program traces using replay. VMware's deterministic replay provides per-instruction precision, but ReTrace comes with prohibitive software overhead especially in the expansion

stage. Thus ReTrace is normally used as an offline profiling tool to analyze trace-based program behaviors or generate various types of precise traces.

Comparing with existing deterministic re-execution approaches that checkpoint entire VM or application to disk, we will instead checkpoint on a per-store granularity to memory within a single application only. Checkpointing overhead seems to correlate closely with the checkpointing granularity—the coarser grain of checkpointing, the higher overhead it causes. Thus we decide to checkpoint over the finest-possible granularity and expect much reduced overhead. We plan to provide a per-instruction level precision when deterministic re-execution takes place. In addition, we will guarantee that precise application context be restored for the deterministic replay stage. This includes CPU registers, the application's complete memory contents, precise exception state, and any application-level global variable that may have been modified due to side effects of the checkpoint-enabled application's execution. However, since we will conduct in-memory user-level checkpointing, we cannot precisely restore states in I/O devices, as well as states that are changed in system codes. In contrast to existing work, we strive to build a high-performance checkpointing framework with a heavy emphasis on maximium overhead reduction.

2.5.2 Speculation

Thread-level speculation [16, 27, 59] (*TLS*) and optimistic implementations of Transactional Memory [28, 29, 57] (*TM*) are optimistic program executions whose result might not be needed (and thus discarded). Both TLS and TM provide for each optimistic thread support to checkpoint before execution and rollback when an error occurs. Dependency tracking and conflict detection are necessary to make the optimistic parallel execution model complete.

Depending on available hardware support, there exists a large exploration space for implementing this speculative parallel model. Hardware-based solutions (*HTM*) [28,30,65] often provide enhancement on cache protocols and thus can partially reuse L1 or L2 cache as the speculative buffer. Conflict detection and rollback are also implemented in hardware, thus hardware-based solutions can often provide higher performance. However, it is non trivial to design and implement a simple yet efficient cache-protocol extension that guarantees both correctness and efficiency. These schemes also face the challenge of overflowed buffers that ultimately limit the granularity of speculative work. Software-only solutions [2, 29, 57] (*STM*) remove the hardware dependency by implementing the entire logic in software. They are thus not limited in buffer capacity, but suffer from prohibitive software instrumentation overhead. In between there exist hybrid solutions [8, 18, 44] (*HyTM*) that attempt to

combine the best of both sides—make the common case (small transactions) fast by using HTM, and make the uncommon case (large transactions) safe by default to STM.

In contrast to many TM or TLS solutions that use hardware buffering for multi-thread workloads, we instead focus on using software buffering for single-thread applications. This allows us to focus entirely on doing the best-possible work for checkpointing alone and remove any dependency on checkpointing hardware that doesn't exist. Existing software-only checkpointing solutions suffer from prohibitive checkpointing overhead, thus we strive to reduce the overhead by conducting checkpointing at the finest-possible granularity. We exploit the compiler as the key tool to enable software-only checkpointing by instrumenting on a per-store granularity. We further explore aggressive compiler analysis and optimizations over the distinct opportunities provided only by the compiler-driven program instrumentation. Build upon our efficient software-only checkpointing infrastructure, we plan to support a few important single-thread speculative applications—a domain that needs further and thorough investigation.

2.5.3 Software Backtracking

Software Backtracking [24, 25, 36] allows a program to execute backwards from the current location. Common uses of backtracking include reverse execution in a debugger or a virtual-machine environment, and individual applications that exercise backtracking algorithms.

Program debugging often demands frequently revisiting passed program locations and states when attempting to analyze and isolate the root cause of a bug. A backtracking-enabled debugger can greatly simplify the reverse-execution debugging process by eliminating the need for program restart, as well as avoiding all problems associated with reproducing the precise bug-trigger environment.

Agrawal et al. [24, 25] presented a prototype debugging tool that is based on dynamic program slicing and execution back-tracking. It provides a structured view of dynamic events through run-time traces, but is constrained by storage limitations.

Ding et al. [19, 66] pioneered behavior-oriented parallelization (*BOP*) that attempts to parallelize applications with unpredictable control flow, indirect data access, and input-dependent parallelism. *BOP* allows a program to be parallelized based on partial information about a program's behavior and user hints. *BOP* is based on programmable software speculation, where a user or an analysis tool marks possibly parallel regions in the code, and the run-time system executes these regions speculatively in parallel. *BOP* uses process consistency where the operating system starts speculative execution by forking the current non-speculative process and making copies of each existing memory pages. This

imposes high software overhead by conducting process forking and thus forces implicit coarse-grain speculative regions.

Recent versions of the gnu debugger (`GDB`) [22] allow reverse program execution by conducting instruction-wise program replay. It generates the replay log by conducting a per-instruction checkpointing inside its replay buffer for each individual instruction once the debugger is in the reverse-execution mode. For each instruction, the checkpointing process copies the memory contents that the instruction is about to overwrite, and also selectively saves sufficient CPU registers that are necessary to reconstruct loss-free CPU state upon a rollack request. Thus, for each instruction, the replay will restore the saved per-instruction checkpoint memory contents and the CPU register states. This per-instruction checkpointing comes with prohibitive overhead because the debugger has to preserve sufficiently large machine states and original memory contents in order to precisely reconstruct each instruction-wise machine state after a per-instruction restore. This scheme also allocates a fixed-size replay buffer and limits the replay distance for up to 200K instructions,[1] placing an implicit constraint on the maximum size of code region that can be checkpointed and replayed.

In contrast to existing backtracking applications, our thread-based checkpoint-enabled debugger will not have such limitations. In addition to offering features including unlimited retries, our checkpointing scheme allocates its buffer in heap memory so that it can grow the buffer dynamically when needed. This allows our checkpoint-enabled debugger to support reverse execution over regions with potentially unbounded size. The key feature that differentiates our design from the rest is the ability to conduct code optimizations after performing aggressive program analysis when implementing the support for backtracking inside a compiler. Our compiler has a complete view of the entire application's source code, thus it is capable of making better optimization decisions. We do not quantitatively compare with a reverse-execution enabled debugger because such debuggers only become available towards the end of this research.

We further extend support of program backtracking on source-code level. We expose the checkpointing functionality to the user via a set of simple APIs, so that programmers can have explicit control of program backtracking by specifying the precise checkpoint region location through a pair of region delimiters at the source-program level. Rather than supporting all tedious and error-prone details of conducting manual backtracking on each individual stores, our compiler-based scheme automates the entire enable-backtracking process and frees the programmer to instead focus more on other important

[1] $200K$ instructions is the default replay buffer count on GDB 7.0

issues. This will help to reduce the develop-run-debug cycle time and convert the improvement made on the development process into improved programmer's productivity.

Chapter 3

Enabling Checkpointing

As previously introduced in chapter 2, checkpointing [35, 48, 56, 62] is the process of taking program snapshots to facilitate failure recovery on potential future program errors. In contrast with previous work on hardware-support for checkpointing or coarse-grain software-only checkpointing based on copying large memory regions or cloning entire objects in software, in this chapter we discuss a lightweight compiler-based software-only solution to checkpointing that operates at the level of individual variables.

We plan to leverage compiler techniques to enable checkpointing on its finest granularity and aggressively optimize the checkpointing-enabled code for maximum efficiency. By choosing a software-only scheme, we avoid all problems related to the checkpointing hardware that may not exist in a given system. By proposing a lightweight and fine-grain scheme, we target performance-critical applications with rapid recovery from potentially frequent ill-behaviors. We plan to further explore checkpointing overhead reduction by aggressively applying compile-time analysis and optimizations. To our best knowledge, this is the first such attempt on fine-grain software-only schemes. We plan to enable software-only checkpointing over individual stores—the finest source-code level checkpointing granularity that naturally adapts to program behavior. Previous work [12, 14, 50] shows that a compiler can play a key role in building an efficient checkpointing solution by eliminating checkpointing overhead. We rely on a compiler to provide precise data type and size information, and conduct compile-time analysis to extract critical information that exhibits potential optimization opportunities that are normally not available at runtime. In addition, compilers can automate program transformations that will free users from tedious and error-prone manual annotations especially over large program regions with complex constructs. We plan to leverage compiler techniques to automate the checkpointing-enabling process and aggressively optimize our proposed software-only checkpointing solution targeting

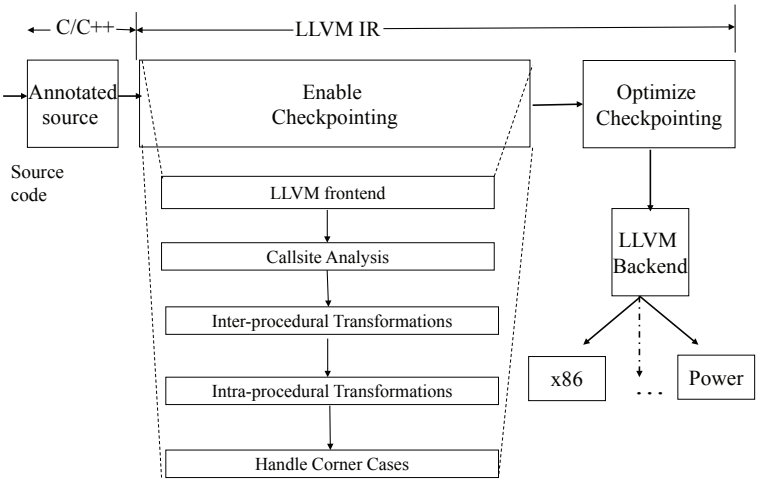


Figure 3.1: Overview of enabling checkpointing.

maximum overhead reduction.

We plan to relax the constraints on applicable regions that can potentially be checkpointed. We grant the user the ability to indicate a region of arbitrary size and complexity. A checkpointable region can be as small as one line of source code, or as big as the entire application. A user only needs to identify the region by marking both boundaries, with the rest of the process fully automated by the compiler. We will further provide a robust checkpointing infrastructure to enable novel features on key applications that can benefit directly from the efficient software-only checkpointing support.

3.1 Overview

Figure 3.1 presents an overview of our checkpointing system (*CKPT*), with detailed steps on compiler analysis and transformations to enable checkpointing. The system takes as input a source program, with annotations that indicate the locations where a checkpoint region begins and ends, as well as code that decides whether the checkpoint should be committed or rewound when the current checkpointing process is about to complete. The only required user action is to identify the checkpointing program region by inserting a pair of region delimiters, then our compiler automates the rest of the process to enable checkpointing.

The entire enable-checkpointing process consists of four major steps:

Step-1 converts the C/C++ language source code to an intermediate representation (IR) that LLVM [37, 38] operates and manipulates on. We choose to work on the LLVM IR level to gain language, compiler and platform independence. In Figure 3.1, we show only C/C++ inputs because our test applications happen to be written in these languages. In reality, there is no limitations on the selections of programming languages a user can build applications from and enjoy the benefits offered by our checkpointing framework. Despite a large number of existing compilers that already support LLVM, any compiler that provides a frontend that can convert source language code into LLVM IR will be able to utilize our checkpointing framework.

Step-2 (*Callsite Analysis*) analyzes callsites that reside in the user-defined checkpointing region. Using a callsite analysis algorithm, it recursively visits all function callsites within the checkpointing region and discovers all user-defined functions that may be invoked directly or indirectly from the region. This information becomes a vital prerequisite that enables step-3 to complete the checkpointing enabling process for any user-annotated program region.

Step-3 (*Inter-procedural Transformations*) conducts program restructurings to enable checkpointing on all participating user-defined functions. For each function identified by the callsite-analysis phase, our compiler generates a checkpoint-enabled version that co-exists with its original version.

And finally, step-4 (*Intra-procedural Transformations*) conducts program transformations that enable checkpointing inside the annotated region only.

After these four steps of compiler analysis and transformations, we produce a program that is checkpoint-enabled with respect to the user-annotated program region and functionally equivalent to the original program. We will unveil further details on steps and actions of enabling checkpointing later in the chapter. Our checkpointing transformations and optimizations are implemented as custom LLVM [37, 38] passes, with each pass targeting a particular analysis or transformation action. We use an existing pass manager from LLVM to establish a pass-execution order that naturally resolves all inter-pass dependencies. This pass manager guarantees that each of our custom LLVM passes will be visited at least once while respecting all dependencies among our checkpointing-specific transformation and optimization passes.

When the checkpointing framework completes its transformations, it produces transformed LLVM IR that has checkpointing enabled over the user-annotated program region. This transformed LLVM IR also preserves the behavior and correctness of the original application. This code can then proceed to optimizations and further target multiple native platforms that LLVM infrastructure supports. LLVM's platform support includes X86, X64, and ARM where the native code generators are production quality.

LLVM includes a large number of platform targets (PowerPC, Solaris, MIPS, XCore, PTX, AMD, etc.) where the current support is still experimental. In addition, LLVM provides a `C` backend that can convert optimized `LLVM IR` back to `C` source code. This source-to-source approach improves code portability and allows us to further capitalize on all optimizations of native back-end compilers.

3.2 Compiler Infrastructure

Our development of checkpointing compiler analysis and optimizations is based on well-established open-source research compiler infrastructures. As we will describe in the following sections, we initially used SUIF because of its stability, good ANSI-C standard compliance and source-to-source compilation path. However, we were later motivated to migrate our work to LLVM.

3.2.1 SUIF

SUIF [7, 26] is a open-source compiler infrastructure originally developed at Stanford University. It provides a solid platform for compiler research. SUIF has two main components: a small, well-documented *kernel*, and a *toolkit* composed of compiler passes that operate on the kernel. The kernel defines SUIF's intermediate representation (*IR*) that specifies the details of the language and provides functions to access and manipulate the *IR*. The toolkit includes `C` and `Fortran` front ends that convert source codes written in these languages into SUIF IR, a loop-level parallelism and locality optimizer, a MIPS back end, and helper tools for compiler development. Later enhancements include a *SUIF-to-C* backend that converts `SUIF` IR to low-level `C` code, as well as work that improves language compliance of the produced `C` code to ANSI `C` standard.

We started our compiler-based checkpointing using SUIF (version 1.3.0.5)— an available version that is considered most stable and C-standard compliant. However, active development and support of SUIF ceased in 1997. While we developed a large amount of code using the SUIF infrastructure, the lack of support, existing expertise, and bug fixes prevented us from making timely progress—hence we migrated our compiler infrastructure from SUIF to LLVM.

3.2.2 LLVM

LLVM [37,38] is an open-source software infrastructure that provides modular and reusable components for building compilers and programming tools. LLVM's building-block components are ideally language and target independent. Using LLVM, one can construct a new compiler with a selection

of the right components that exist, glue code, plus any components that are not currently available (need to be written). This software development using library-based composition model reduces the time and cost to construct a particular compiler and allows existing components to be shared across different compilers over different platforms. Improvements made to one compiler can implicitly benefit all other compilers that use LLVM's building-block components.

LLVM includes a small, carefully-designed, easy-to-understand and well-documented intermediate representation (*IR*), and a large number of existing program analysis, transformation and optimization passes that manipulate the IR. Many frontends exist that can convert language source code (`C, C++, Java, Fortran, Python, etc.`) to LLVM IR. LLVM's middle-end includes a large number of robust program analysis and transformations that are popular in most modern compilers. This covers scalar optimizations, loop-based optimizations, instruction combining and simplifications, instrumentation for performance debugging, advanced inlining, various forms of alias analysis and inter-procedural optimizations (IPO) that rely on the precise results of alias analysis. Important recent development includes automatic vectorization on both basic-block level and loop level, as well as support for OpenMP and PolyHedral framework [10, 23]. Platform backends are available that generate architecture-specific machine code, including x86, x64, ARM, PowerPC, MIPS, etc. Many of these platforms' native code generators are considered robust and production-quality. LLVM also provides solid just-in-time (JIT) compilation for many of the supported platforms.

LLVM is extremely well documented and makes it easy to discover certain design features, coding patterns and tips that can avoid errors. Online discussions are archived daily and are available for easy search. LLVM is sufficiently mature and is released as commercial product and being actively serviced by Apple (Clang, XCode, etc). LLVM is currently under active development with a 6-month release cycle. There is a large, friendly and responsive LLVM community that is helpful in leveraging existing expertise and dealing with project-specific problems.

We migrated our checkpointing compiler development from SUIF to LLVM. It turned out to be a vital step that not only eliminated all problems SUIF brings, but also greatly enhanced development experience and improved productivity. Although we could not reuse most of the code that we originally developed for SUIF, the ideas and algorithms are well leveraged and lead to a rapid transition.

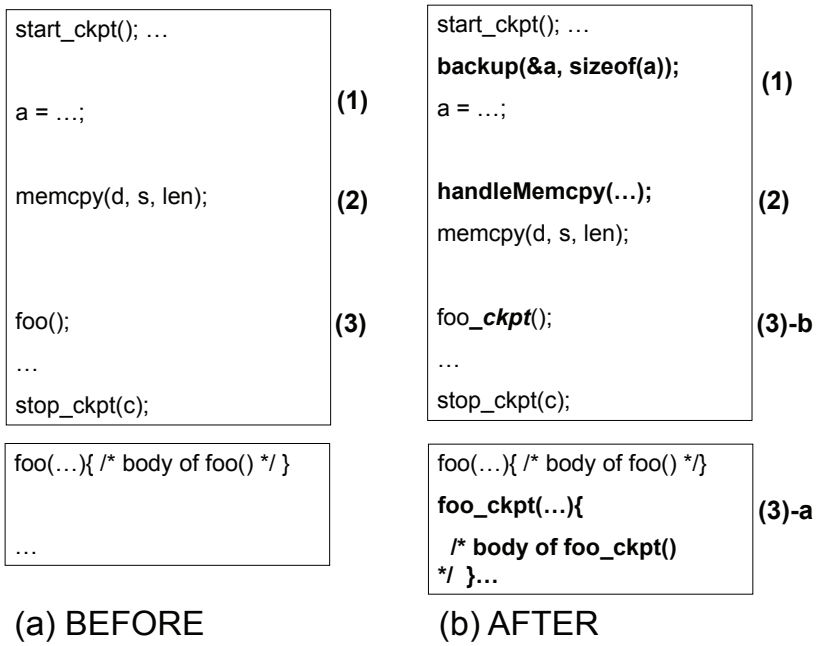


Figure 3.2: Three code-transformation actions to enable checkpointing

3.3 Checkpointing Analysis

Before our checkpointing system can conduct program transformations to enable checkponting, the compiler must first analyze the user code both inside and outside of the checkpointing region. This analysis collects program information, understands program structure, makes plans and decisions on code transformation, and guides the remaining transformation process. Thus on the highest level, we separate the process to enable checkpointing into roughly two steps: (i) checkpointing analysis and (ii) checkpointing transformation.

3.3.1 Identifying Checkpointing Regions

Users of the checkpointing system only need to identify the checkpointing region boundaries, and the rest of the checkpoint-enabling process is fully automated by our compiler. A programmer can insert a pair of special function callsites to identify the checkpointing region: `start_ckpt()` to mark the begin of the checkpoint region, and `stop_ckpt(bool cond_code)` to mark the end of the checkpoint region, respectively. The checkpointing region can be arbitrarily large and may contain complex program constructs, including pointers, function callsites, loops, recursions, etc.

We establish the following two requirements for the placement of checkpoint region markers:

1. `start_ckpt` must **dominate** `stop_ckpt`;

2. `start_ckpt` and `stop_ckpt` must be on the same lexical scope.

These two dominating requirements guarantee that a program's control flow will reach the end of the checkpoint region after visiting the begin of the checkpointing region if the control flow doesn't prematurely terminate within the checkpointing region. These two dominating requirements also imply that both the `start_ckpt` and the `stop_ckpt` markers are within the same function.[1]

Note that `stop_ckpt` takes a boolean argument (`cond_code`), which instructs the checkpointing system to further conduct a checkpoint *abort* or *commit* operation. If a boolean value *true* is provided, the checkpointing system will commit the current checkpoint. Otherwise, if a boolean value *false* is provided, the checkpointing system will abort the current checkpoint and rewind execution back to the checkpoint-begin location. We use this method to allow a programmer to communicate with the checkpointing system and to control the post-checkpointing action through an argument over the published checkpointing API once execution reaches the end of checkpoint region. We give the complete checkpointing API in appendix A.

A *run-away checkpoint* is a thread that has started a checkpoint region without ending it. Although we can dynamically grow the checkpoint buffer in main memory, a run-away checkpoint will likely completely exhaust available memory and fail due to lack of memory. We thus establish requirements on (i) pairing the checkpoint-region markers over a dominating relationship and (ii) forcing them to appear within the same lexical level. These requirements guarantee that the checkpoint-enabled application is free of run-away checkpoints.

The compiler can potentially provide feedback if a user places the `stop_ckpt` marker at an inappropriate position (e.g., not on the same lexical level as the `start_ckpt` marker). The compiler can suggest nearby alternative positions that satisfy the marker-replacement requirements, or force to insert a `stop_ckpt` marker at the correct location if the compiler has high confidence of its action. Alternatively, if a user's real intention is to enable run-away checkpointing, the user can pass this requirement through a compile-time flag so that the compiler will skip the analysis on the placement of the `stop_ckpt` marker.

We can further instruct the compiler to allow multiple `stop_ckpt` markers. A user can selectively place more than one `stop_ckpt` marker in multiple program locations (e.g., both paths from a control

[1] Placing both checkpoint region boundaries inside the same function is a necessary requirement. Otherwise, the compiler can not guarantee a correct execution order because the `stop_ckpt` can potentially be invoked before `start_ckpt`.

branch). Our compiler can collect all `stop_ckpt` marker locations and analyze their combined intent: if it is equivalent to placing a single `stop_ckpt` marker (e.g., placing two `stop_ckpt` markers on both paths from a control branch is the same as placing a single `stop_ckpt` marker on the dominating path of the control branch) on the same lexical level as the `start_ckpt` marker, the enable-checkpoint process will proceed as normal.

There are alternative methods to allow a programmer to identify a specific program region. This usually includes using `pragma` or language-extension primitives. For simplicity we utilize a pair of special user-space function callsites as region delimiters to avoid modifying the LLVM front-end.

3.3.2 Single-File Applications

A modern software project is normally composed of a large number of files residing in various (often nested) directories. This is a preferred design for good software engineering practice. However, this also creates difficulties for our compiler-based checkpointing work. Since the checkpointing framework needs to process all user-defined callsites within the checkpoint region, it must identify the location (file name and line number) of potentially each user-defined function. This unnecessarily complicates the checkpointing analysis with virtually no benefit. We need a simple and effective answer for this problem.

We propose a solution called Single-File Application (*SFA*). For each project, a *SFA* is basically one giant file that contains full details of all participating files regardless of directory nesting. Since it is the only file of the project, all user-defined functions will appear inside this file. SFA is a way to ease the complexity to perform whole-program analysis. SFA saves a significant amount of work that needs to locate the proper file under nested directory structures, the work that needs to parse the file to identify a function in need, as well as linking all relevant LLVM IR (in separated files) to the caller (the file that defines the checkpointing region). For the vast majority of our test applications, building a SFA manually for an existing application is a straight-forward but labour-intensive process. Some difficulties arise in dealing with `static` data and `static` functions. Fortunately, we manage to work around them with some careful manual efforts.

3.3.3 Callsite Analysis

Our compiler needs precise knowledge on all user-defined functions that may be called directly or indirectly from the checkpointing region. We call the process of discovering all such user-defined functions *callsite analysis*. Callsite analysis is an LLVM analysis pass that visits an application's

```
start_ckpt();
…
  foo();
  …
  if (C) bar();
  …
…
stop_ckpt(c);

foo(void){ printf("foo\n"); }

bar(void){ func1();
  … if(C1) func2(); …
}
```

Code to enable checkpointing

```
1. foo();

2. bar();

3. func1();

4. func2();
```

List of all functions that need checkpointing

Figure 3.3: Code sample for callsite analysis

```
INPUT: CFG of the input program
OUTPUT: listed of functions that need to have CKPT-enabled version
INTERMEDIATE: List of Callsites ListCS (Initialized to NULL)
              List of Functions ListF (Initialized to NULL)

BEGIN
  // 1. collect All user-defined callsites in CKPT region
  foreach CallSite CS within CKPT Region do
    if ( ! isDeclaration (CS)) insert (CS, ListCS)

  // 2. recursively visit each involved user-defined function:
  while (! empty (ListCS) ) do
     CS1 = pop (ListCS);  F = getFunction (CS1); insert (F, ListF);
     foreach CallSite CS in F do
       if ( !isDeclaration (CS) AND !visited (CS, ListF, ListCS))
         insert (CS, ListCS)

  // 3. result appears in ListF:
END.
```

Figure 3.4: Algorithm of callsite analysis

SFA LLVM IR, performing program analysis but doesn't commit any program transformation. LLVM maintains the analysis result and will provide it when another analysis or transformation pass requests it later. The callsite analysis pass visits each node in the application's partial call graph that originates from the annotated checkpointing region. It proceeds by recursively identifying all user-defined functions in this partial call graph and mark them as functions that require the creation of a checkpoint-enabled version. We give an example of callsite analysis in Figure 3.3 and present its algorithm in Figure 3.4.

Figure 3.3 shows a code sample with user annotations that mark the checkpointing region. The checkpointing region contains a user-defined callsite (`foo`) on all paths and a user-defined callsite (`bar`) on a conditional path. Function implementations of both the callsites are available. The callsite analysis algorithm starts by pushing all user-defined function callsites within the checkpointing region (`foo and bar`) onto an empty list (`ListCS`, Figure 3.4, step 1). While `ListCS` is not empty, the algorithm proceeds by isolating the first callsite from top of the list (`CS1` which holds `foo`), and pushing all user-defined function callsites found inside its implementation back onto the list (`ListCS`). Each time the algorithm visits a new callsite, the same callsite is also inserted into a result container (`ListF`). Before processing a new callsite, we check the result container to ascertain that it has not yet been processed. This extra step helps to avoid indirect recursions. Note that in `foo`, the callsite `printf` is not a user-defined function because its function implementation is not available in the current SFA. Thus, `printf` will not be pushed into either list. The algorithm continues its breadth-first callgraph traversal until there is no callsite candidate remains (`ListCS` becomes empty, in Figure 3.4, step 2). When the algorithm finally converges, `ListF` contains the result of the callsite analysis—all user-defined functions that will be called directly or indirectly from the checkpoint region in the current *SFA*. Each function in this result list (`ListF`) needs to have a checkpoint-enabled version that will coexist with its original version.

3.4 Checkpointing Transformations

Once callsite analysis completes its work and generates the list of functions it identified through its analysis algorithm, the compiler can proceed to enable checkpointing through program transformations.

3.4.1 Intra-procedural Transformations

The compiler converts code inside the user-annotated region to its checkpoint-enabled equivalent version in three steps. Step-1 is to precede each write with code to backup the write location into a checkpoint buffer. Figure 3.2 step-1 shows that variable `a` is modified and thus preceded with a `backup` operation

which copies its contents into the checkpoint buffer immediately before the corresponding write. The backup activities occur inside the `backup(char * addr, int len)` function call. The `backup` interface takes two arguments: a `char * addr` that indicates the memory address of the memory contents to be copied, and an `int len` that shows the length of the to-copy memory contents. The transformation in step-1 generates a `backup` call with the proper starting address and precise length to cover the memory contents that need to be protected through the backup operation. All activities inside the backup operation are handled within the `backup` call. We encapsulate its details inside a run-time checkpointing support library that we will introduce in chapter 5.

Step-2 is to handle certain system functions that have implicit memory writes. Figure 3.2-(2) illustrates the handling for one of such routines (`memcpy`) by placing a special handling function (`handleMemcpy`) immediately before it. The `handleMemcpy` understands the expected behaviors of `memcpy` and will properly backup memory contents that may be overwritten inside `memcpy`. There are a limited number of system functions that need special handling for checkpointing. We provide an exhaustive list of all supported special system functions and their respective handling routines in Table 3.1. We further provide details on relevant implementations of all available handling routines in appendix B.

Step-3 is to rename any user-defined function callsite to its checkpoint-enabled version inside the checkpointing region. Figure 3.2-($3-b$) shows that a user callsite (`foo`) is renamed to its checkpoint-enabled equivalent (`foo_ckpt`) by appending `_ckpt` on its name. All user-defined functions residing inside the checkpoint region need to rename to their checkpoint-enabled equivalent to comply with correct checkpointing semantic. However, the same user-defined function may also be called outside of the checkpointing region. In this case, the original version will remain unchanged. Thus for each user-defined function that is identified through the callsite-analysis process, we generate a checkpoint-enabled version that co-exists with its original function. E.g., Figure 3.2-($3-a$) shows an automatically generated function (`foo_ckpt`) that is the checkpoint-enabled version of a user-defined function (`foo`). The respective callsite is renamed to its checkpoint-enabled version only inside the checkpointing region or any other checkpoint-enabled functions.

Note that the actions conducted inside intra-procedural transformations deal with only the code within the user-annotated checkpoinitng region. We introduce a separate transformation process—inter-procedural transformations, to generate the checkpoint-enabled version for any user-defined functions that may be called directly or indirectly from the checkpointing region (the user-defined function list identified by the callsite-analysis phase).

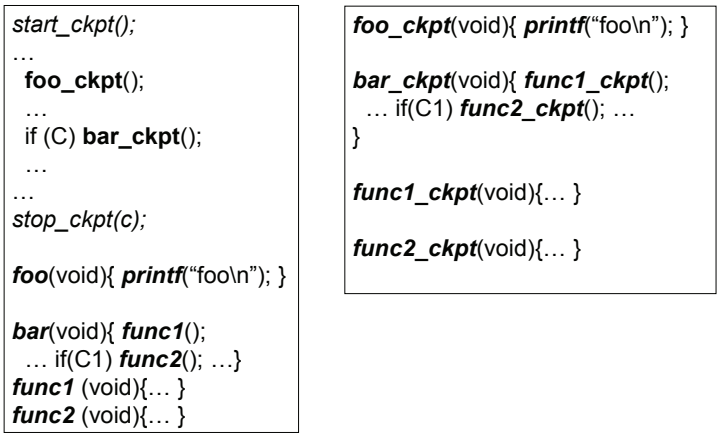


Figure 3.5: Sample code with checkpointing enabled

3.4.2 Inter-procedural Transformations

The final step is to enable checkpointing on all user-defined routines that may be called directly or indirectly from the checkpointing region. Recall these routines are identified through the *callsite-analysis* phase. Each callsite to this list of functions inside the checkpoint region has been renamed to its checkpoint-enabled version. For each function in this identified function list, we clone its function body and rename it by appending `_ckpt` to its original name, as shown in Figure 3.2-(3). Inside the body of the cloned function, we recursively and repetitively apply the same three principles introduced in section 3.4.1: (i) precede each store with a `backup` operation, (ii) handle special system functions that have implicit memory writes by inserting a special handling routine, and (iii) handle user-defined function callsites through renaming. With some careful software engineering practice, we can reuse most of the code developed for section 3.4.1. When the process completes, we produce a checkpoint-enabled version for every user-defined function that can potentially be called from the checkpointing region.

Figure 3.5 presents the checkpointing-enabled sample code whose original version is given in Figure 3.3. As shown in Figure 3.5-(a), the transformation has renamed all user-defined function callsites within the checkpointing region: `foo` becomes `foo_ckpt`, and `bar` becomes `bar_ckpt`. The checkpointing code region is relatively simple and the other two types of enabling transformations (handling special system functions and generate a `backup` call per store) are not applicable. Callsite-analysis process identifies a total of four functions that need to generate their respective checkpoint-enabled

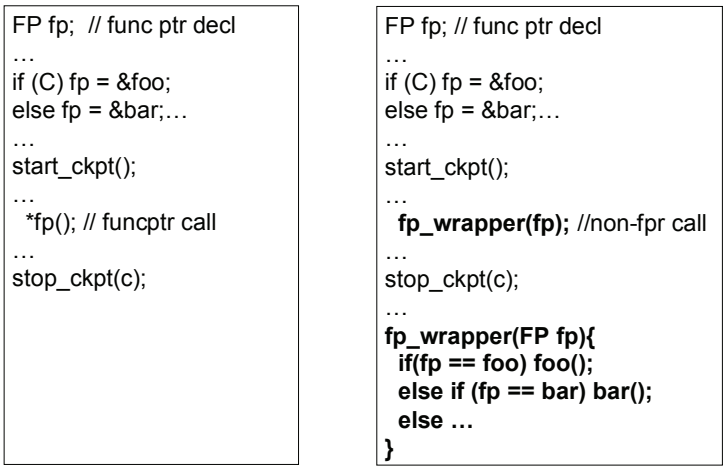


Figure 3.6: Function pointer callsite

versions. These functions are: `foo, bar, func1 and func2`. As a result of the inter-procedural transformation, in Figure 3.5-(b) we present their corresponding checkponit-enabled versions. These newly generated functions are `foo_ckpt, bar_ckpt, func1_ckpt, and func2_ckpt`. The list of functions is generated through the callsite-analysis process given in Figure 3.3. These checkpoint-enabled functions are thus generated according to the established rules presented in section 3.4.2. Notice that in the checkpointing-enabled code, both the original functions and their checkpoint-enabled versions coexist. This will unavoidably increase the code size. However, since we limit these functions to those that are identified through the callsite-analysis process, only a small fraction of all user-defined functions are applicable for the transformation.

3.4.3 Handling Special Cases

In addition to the analysis and transformations presented earlier in this chapter, there are special conditions that may prevent our compiler-based checkpointing from completing its tasks. These special conditions need special treatments to work around them. We have thus identified two special cases: function pointer callsite and premature return.

3.4.3.1 Function Pointer Callsite

Special cases exist during the checkpoint-enable process. Since our checkpointing scheme proceeds with cloning user-defined functions, the compiler needs to identify the precise callee function for each

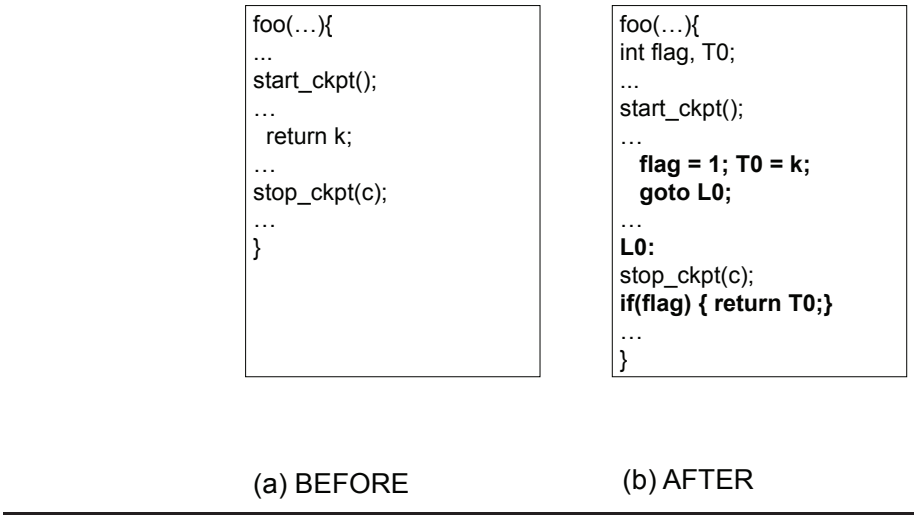


Figure 3.7: Premature return from checkpointing region

Index	System Function	Checkpont Support Function
1	memcpy(i8*, i8*, i32)	handleMemcpy(i8*, i8*, i32)
2	memmove(i8*, i8*, i32)	handleMemmove(i8*, i8*, i32)
3	memset(i8*, i32, i32)	handleMemset(i8*, i32, i32)
4	strcpy(i8*, i8*)	handleStrcpy(i8*, i8*)
5	strcat(i8*, i8*)	handleStrcat(i8*, i8*)
6	strncpy(i8*, i8*, i32)	handleStrncpy(i8*, i8*, i32)
7	strncat(i8*, i8*, i32)	handleStrncat(i8*, i8*, i32)
8	sprintf(i8*, i8*, va_arg)	handleSprintf(i8*, i8*, va_arg)
9	vsprintf(i8*, i8*, va_arg)	handleVsprintf(i8*, i8*, va_arg)
10	snprintf(i8*, i8*, va_arg)	handleSnprintf(i8*, i8*,va_arg)

Table 3.1: Supported System Functions

involved user callsite at compile time. Calls through function pointers won't satisfy this requirement because the precise callee function is only resolved at runtime. We give an example of call-through-function-pointer callsite in Figure 3.6-(a).

We handle this function pointer callsite ambiguity by changing from a function pointer callsite to a normal function callsite with the function pointer wrapped as an argument. If there are arguments from the original function pointer callsite, they will be passed as additional arguments on the wrapper function. Within the wrapper function, each possible callee (any function that has its address taken within the entire SFA) is explicitly examined through a list of parameter-match candidates. As shown in Figure 3.6-(b), we change from the original function pointer callsite (`*fp`) into a normal function wrapper callsite (`fp_wrapper(fp)`), with the function pointer wrapped as an argument on the wrapper function. Inside the wrapper function, we exhaustively examine all functions that have their addresses taken across the entire program. E.g., both function `foo` and function `bar` have their addresses taken within the current `SFA`. We thus explicitly examine both of their addresses inside the wrapper function, trying to match the precise function callsite. Once we identify a match, we will make a normal function call to this matched function (implementation of wrapper function `fp_wrapper()`). This may seem to be overwhelming in the beginning. But in practice, we find it reasonably easy and straight-forward to implement. In addition, the number of all functions that have their addresses taken is relatively small in our testing applications.

3.4.3.2 Premature Optional Return

An other special case deals with early exits from the checkpointing region, as shown in Figure 3.7-(a). A `return` statement within the checkpoint region may prematurely terminate the program's execution without visiting the `stop_ckpt` marker. Visiting the `stop_ckpt` marker is essential to complete the current checkpointing processing, thus an optional premature return violates the rule that the checkpoint region markers must be visited in pairs and will put the current checkpointing process in an unknown or inconsistent state.

Figure 3.7-(b) suggests a possible solution. It reserves the appropriate return value (in variable `T0`) and transforms the code with a mandatory `goto` statement and thus forces execution to branch to the `stop_ckpt` marker. We further introduce a boolean variable (`flag`) that controls the return value, in preparation for the original `return` statement that may optionally returns a value. Note that both special-handling cases are rare in our test applications. We thus only conduct the necessary changes through manual steps rather than building compiler passes to automate the special-case handling.

3.5 Summary

In this chapter, we introduce the detailed steps to enable checkpointing for any user-annotated program region. A user only needs to mark the region's boundaries and our compiler automatically completes the rest of the checkpointing-enablement process. Compiler analysis examines the marked checkpointing region to discover all user-defined functions that may be called directly or indirectly from the checkpointing region. Compiler transformations proceed to generate a `backup` call immediately preceding each individual memory store, as well as renaming user-defined function callsites and handling special library functions. The same principles apply to both the checkpointing region and all participating functions that need to have their checkpoint-enabled versions. We will provide results of program partition and checkpointing region formation in chapter 7.

This process enables fine-grain software-only checkpointing on a per-store granularity. It supports arbitrarily large checkpointing regions because we buffer program states in main memory and can dynamically increase this buffer when necessary. It is a compiler-driven automatic process that frees users from the tedious and error-prone details of conducting manual checkpointing. However, it is also a user-level process that can only checkpoint code that is available to the compiler. Default checkpointing overhead is relative to the intensity of memory stores in the checkpoint-enabled program region. Thus a checkpoint-enabled program will often have ample opportunities for compiler optimizations, and we will discus these optimization details in chapter 4.

Chapter 4

Optimizing Checkpointing

The basic transformations described in the previous chapter enable checkpointing on any user-annotated program region by backing up memory contents before each explicit or implicit write, handling certain system functions, and dealing with all user-defined function callsites that are called directly or indirectly within the checkpointing region. This process creates a large number of `backup` calls that are potentially redundant or unnecessary, and leaves ample opportunities for program optimizations to reduce checkpointing overhead. In this chapter, we will describe checkpointing-specific compiler optimizations that are organized as a checkpointing optimization framework.

4.1 Checkpointing Optimization Framework

We present a detailed overview of the compiler checkpointing optimization framework in Figure 4.1. The framework takes as input checkpointing-enabled `LLVM IR`, performs checkpointing-specific analyses and optimizations, and produces checkpointing-optimized `LLVM IR` that can further target multiple backend platforms. Each individual optimization is a standalone LLVM pass. All available optimizations operate in a pipeline fashion where the output of an immediately previous optimization becomes input for the current optimization. The framework includes numerous analysis passes and a total of 12 different optimizations organized as ordered LLVM passes respecting their explicit or implicit dependency. In the rest of this chapter, we introduce them in the order of importance. Note that LLVM supports a large number of native backend platforms including x86, x64, ARM, SPARC, PowerPC, and more. LLVM's code generator is able to emit production-quality binary code on x86, x64 and ARM platforms. In addition, LLVM provides `C` and `C++` backends that allows the conversion of optimized `LLVM IR` back to low-level `C` or `C++` source code, while respecting the language

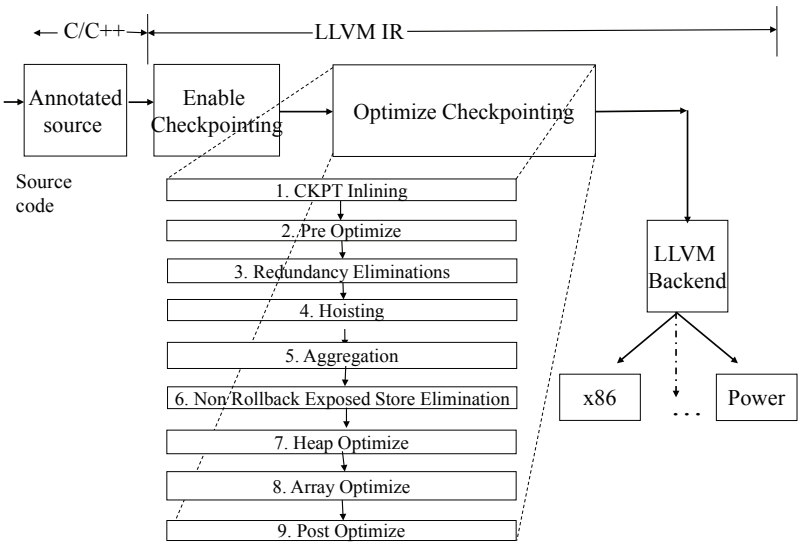


Figure 4.1: Overview of checkpointing optimization framework

syntax and semantics—improving code portability, providing an alternative path for verification, and helping to capitalize on any available native backend compilers.

4.2 Redundancy Eliminations

As we demonstrate later in Chapter 7, the most important optimizations are three cases of redundancy elimination that try to discover, isolate, and eliminate different types of redundancies among `backup` operations within the checkpoint-enabled region or checkpoint-enabled user functions. In this section, we introduce three forms of redundancy elimination: regional redundancy elimination (*RRE*), function-private redundancy elimination (*FPRE*), and hoistable redundancy elimination (*HRE*).

4.2.1 Regional Redundancy Elimination (RRE)

Figure 4.2 shows the detailed steps for performing regional redundancy elimination (*RRE*) analysis and transformations, and we give its compiler algorithm in Figure 4.3.

RRE uses **dominating** relationships among `backup` calls and the transformations are organized into four consecutive steps. Step one (Figure 4.2(a)) is to recognize all `backup` operations within the checkpoint-enabled region or function that are suitable for *RRE*-type of redundancy elimination. In the given code sample, *RRE* identifies three suitable `backup` calls because they all operate on the same address (*&a: address of variable* `a`) with the same length (*sizeof(a): size of variable* `a` *in memory*). Since they all are `backup` functions operating on the same address and length, we give each a numerical

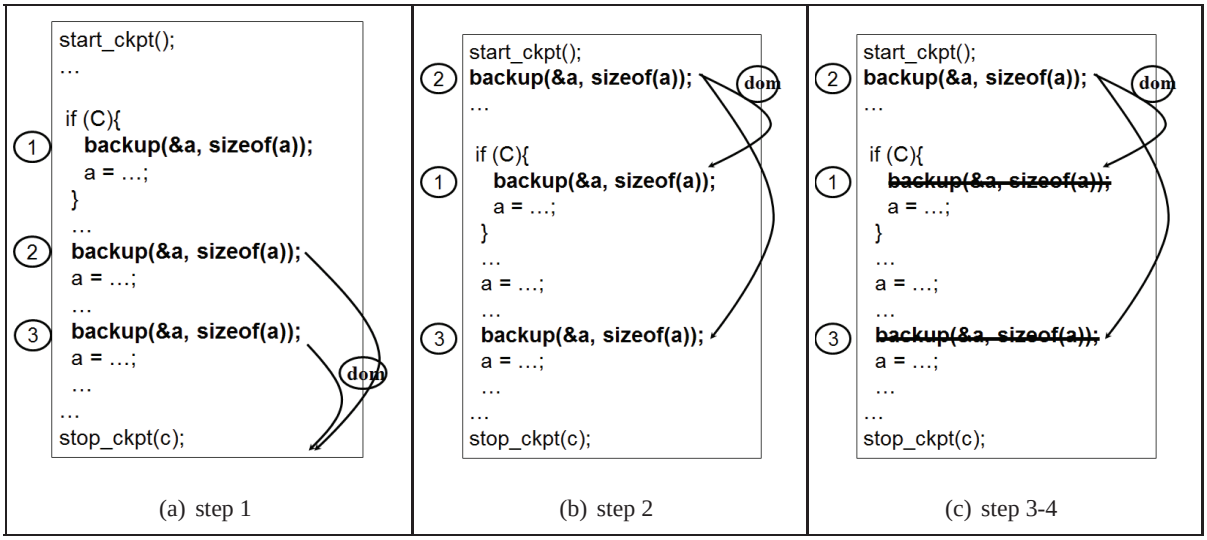


Figure 4.2: Regional Redundancy Elimination (RRE) optimization

ID (from 1 to 3) to better identify and differentiate among them, and will use their numerical ID to precisely identify individual `backup` calls throughout this section. *RRE* then establishes *dominating* relationships between each individual `backup` operation and the `stop_ckpt` region marker. As shown in Figure 4.2(a), both `backup` (2) and `backup` (3) dominate the `stop_ckpt` region marker, but `backup` (1) doesn't.

Step two is to recognize a *leading* `backup` call and attempt to hoist it to a position as early as possible within the region. This hoisting step needs to respect all checkpointing semantics and language syntax. In step one, we have identified that both (2) and (3) dominate the region-end marker, thus they form a chain of backup operations. Since (2) is the first call in this chain, we name it the *leading* backup call (a.k.a, the *leader*). If the leader has no further dependency or limitation, we hoist it to a location immediately after the start-region marker—the earliest position within a checkpoint-enabled region that a compiler optimization can potentially promote an individual `backup` operation to. As shown in Figure 4.2(b), (2) is hoisted to be positioned right-after the `start_ckpt` marker. Due to various constraints, an attempt to hoist a *leader* to the earliest position within the checkpointing region may not always be successful—e.g., if a variable is defined within a checkpointing region, its earliest hoistable target location will not be able to cross its definition position.

Step three is to re-establish *dominating* relationships among relevant `backup` calls after hoisting the *leader*. For all `backup` calls that operate on the same address and length (on the `backup` call chain), we re-establish a pair-wise *dominating* relationship between the *leader* and the rest of the relevant `backup` call(s). Figure 4.2(b) shows that after *leader* promotion, (2) (the *leader*) dominates both (1) and (3).

```
INPUT: CFG of the CKPT Region or Function
OUTPUT: RRE optimized CFG
Intermediate: Leading Backup CallSet: LS = ∅; Promoted CallSet: PS = ∅;

BEGIN
  // 1. identify leading backup calls on the same address:
  group backup calls according to their backup address and length
  foreach backup call bkp in group g within region do
    if (bkp dominate stop_ckpt) insert (bkp, LS)
  // 2. promote leading backup calls:
  foreach backup call bkp ∈ LS do
    if (def(bkp) ∉ region ) AND (bkp ∉ PS) insert (bkp, PS)
  // 3. re-build dominators, and eliminate non-leading backup calls:
  foreach unique backup call bkp1 ∈ (PS or LS) do
    foreach backup call bkp2 ∈ region do
      if ( (bkp1 dominate bkp2) AND (bkp1 ≠ bkp2) )
        remove (bkp2)
END.
```

Figure 4.3: Regional Redundancy Elimination (RRE) algorithm

Step four (the final step) is to keep the *leader* and eliminate all non-leading `backup` call(s) on the call chain that the *leader* dominates within the checkpointing region. Figure 4.2(c) shows that both (1) and (3) are non-leading `backup` calls and dominated by (2) (the *leader*) on the same call chain, thus *RRE* eliminates both (1) and (3).

We present *RRE*'s compiler algorithm in Figure 4.3, which closely resembles the four transformation steps we have introduced. We partition the algorithm into three sections: (i) analyze code and establish dominating relationships with respect to the `stop_ckpt` marker, (ii) identify and promote a *leader* within each `backup` call chain, and (iii) re-establish dominating relationship among `backup` calls inside the same call chain, keep the *leader* and eliminate all non-leading `backup` calls. Notice the seemingly optional leader-promotion step: the transformations will still be correct without *leader* promotion, however this missing step will cause the algorithm to cover much fewer applicable cases. Without the critical leader-promotion step, the `backup` call in (1) will not be eliminated in Figure 4.2 because the leader-dominated `backup` call chain will instead only include (2) and (3). Promoting the identified *leader* enables the algorithm to eliminate *all* relevant `backup` call(s) available in the same call chain, regardless of its relative position.

4.2.2 Function-Private Redundancy Elimination (FPRE)

Function-Private Redundancy Elimination ($FPRE$) identifies all `backup` calls operating on functions' non-pointer type local variables (user data that is allocated on a function's stack storage) and eliminates

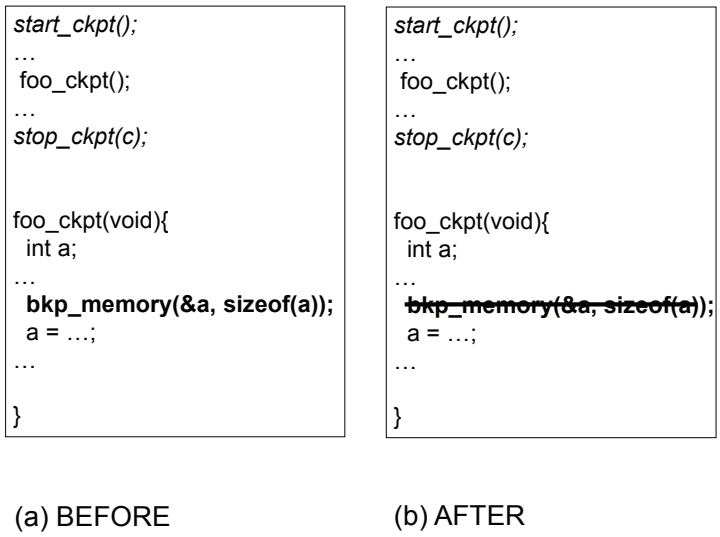


Figure 4.4: Function-Private Redundancy Elimination (FPRE) transformation

INPUT: CFG of a CKPT Function
OUTPUT: FPRE optimized CFG
INTERMEDIATE:

BEGIN

// identify and remove backup on local variables
foreach backup call *bkp* within a ckpt-enabled function **do**
v = ***bkp_addr*** (*bkp*);
if (***isLocalAddr*** (*v)*) ***remove*** *(bkp)*

END.

Figure 4.5: Function-Private Redundancy Elimination (FPRE) algorithm

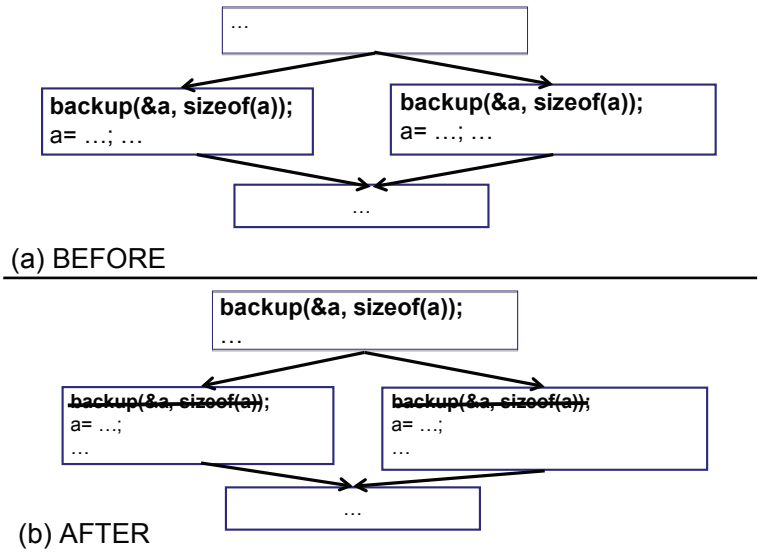


Figure 4.6: Hoistable Redundancy Elimination (HRE) transformation

such `backup` calls. Since any stack-allocated local variables have no memory footprint outside of the function's calling context (when the function is not being called), it is safe to remove `backup` calls operating on local variables in a checkpoint-enabled function without impacting the correctness of checkpointing. Figure 4.4(b) shows that the `backup` call operating on local variable `a` is eliminated. *FPRE*'s compiler algorithm is relatively straightforward and we present it in Figure 4.5. Different from *RRE* that covers both checkpointing-enabled functions and regions, the applicable domain of *FPRE* is limited to checkpointing-enabled functions only.

4.2.3 Hoistable Redundancy Elimination (HRE)

Hoistable Redundancy Elimination (*HRE*) conducts transformations similar to a normal compiler optimization on common sub-expression elimination (*CSE*) by searching for duplicated `backup` operations in both control paths diverged from a branch (shown in Figure 4.6). Once it finds a pair of matching (duplicated) `backup` calls that operate on exactly the same address and length, it will hoist one of the `backup` calls into the immediate common dominator block for both paths, and remove the other one from its original location.

Figure 4.6 shows that *HRE* identifies a `backup` call on both control-flow paths from a direct dominator block. This is a suitable case for *HRE*, and thus the optimization consolidates them into a single `backup` operation and place it into the common dominating block that covers both paths.

We present HRE's algorithm in Figure 4.7. For ease of implementation, we group all `backup` calls

```
INPUT: CFG of the CKPT Region or Function
OUTPUT: HRE optimized CFG

BEGIN
 // 0. identify identical backup calls:
  group backup calls according to their backup address and length
 // 1. identify to-promote backup call pairs inside each group:
  foreach backup call bkp1 in group g within region r do
    foreach backup call bkp2 in group g within region r do
      if ( (bkp1 ≠ bkp2) and (bkp1 !dom bkp2) and (bkp2 !dom bkp1)
          and (i_dom (bkp1) == i_dom (bkp2))
          and (num_dom (i_dom (bkp1)) == 2) )
          // 2. promote backup calls:
          rep_bkp = replicate (bkp1) ; i_dom_bb = i_dom (bkp1)
          insert (rep_bkp, i_dom_bb)
          // 3. remove both of original backup calls:
          remove (bkp1)
          remove (bkp2)
END.
```

Figure 4.7: Hoistable Redundancy Elimination (HRE) algorithm

within the checkpointing region or function according to their target backup address and length (step-0). We then proceed with a doubly nested loop that visits each possible combinations of `backup` call pairs that every group may have (step-1). For this potential `backup-backup` call combination, the algorithm checks for the following conditions. First, they cannot dominate each other. This guarantees that they reside in different basic blocks on the divergence paths originated from the same branch. Second, they must share the same immediate (common) dominator. This eliminates all possible `backup` pairs that are far apart in distance (further down in the control-flow graph, not sharing the same immediate dominator). And third, the common immediate dominator can only have two immediate children. This limits the search paths to only two immediate basic block successors from the branch [1]. When all the conditions hold, we create a replica of the identified `backup` call and insert it into the common dominator block (step-2) and remove both participating `backup` calls from their original locations (step-3).

4.3 Hoisting

Checkpointing hoisting optimization (*hoisting*) aims to harness optimization opportunities inside loops, with a focus on those `backup` calls that can potentially be moved into a loop's preheader. We give a hoisting example in Figure 4.8. A *hoisting* optimization attempts to promote `backup` operations written unconditionally within a loop (variable `z` in Figure 4.8) to the outside of that loop (by default the loop's

[1] The algorithm will ignore code that contains multi-way branches generated from the lowering of a `switch-case` statement—this is the root cause of some performance degradation that we encountered in development.

```
foo(){
int x, y, z;
start_ckpt();
…
for(…){
  …
  backup(&z, sizeof(z));
  z = …; …
  if(…) {
    backup(&y, sizeof(y));
    y = …;
  }
  …
} …
stop_ckpt(c);
…
}// end of foo()
```

(a) BEFORE

```
foo(){
int x, y, z;
start_ckpt();
…
backup(&z, sizeof(z));
for(…){
  …
  z = …; …
  if(…) {
    backup(&y, sizeof(y));
    y = …;
  }
  …
} …
stop_ckpt(c);
…
}//end of foo()
```

(b) AFTER

Figure 4.8: Hoisting optimization transformation

INPUT: CFG of the CKPT Region or Function
OUTPUT: CKPT Hoist Optimized CFG
IMMEDIATE: Set of Loop Exit Basic Blocks: *ES* = Φ; Preheader Block of a Loop: *P* =Φ;

```
BEGIN
  // 0. initialize temp data:
  ES = Φ, P = Φ;
  // 1. identify potential hoist-able backup calls
  foreach backup call bkp within a loop L (use depth-1st traversal) do
    ES = getExitBlocks (L);
    P  = getPreHeader (L);
    if ( isLoopInvAddr (bkp) and (bkp dominate ES) )
      // 2. insert a copy of the backup call into the loop's preheader:
      rep_bkp = replicate (bkp);
      insert (rep_bkp, P);

      // 3. remove the original backup call: bkp
      remove (bkp);
END.
```

Figure 4.9: Hoisting optimization algorithm

preheader). Thus for all unconditionally-hoisted backup operations, this optimization ideally brings the benefit of *N*-to-1 reduction on checkpointing overhead where *N* is the loop's trip count. Such hoisting would not be performed by a normal compiler hoisting pass (e.g., loop-invariant code motion—*LICM*) since the write to the variable is not necessarily loop invariant.

The decision on hoisting conditionally modified `backup` calls remains interesting because we encounter cases that produce results on both sides of the performance impact. Due to the difficulty in precisely predicting branch behaviors statically at compile time, it is hard to generalize such cases and abstract them into an algorithm that leads to solid performance benefits. After conducting intensive empirical experiments, we decide not to hoist variable `y` in the given example because hoisting such cases often results in more overhead than benefit. [2] We qualify this condition as hoisting *only* `backup` calls that dominate *all* of the loop's exits.

We present the *hoisting* algorithm in Figure 4.9 that focuses only on handling unconditional `backup` operations inside a loop. The algorithm starts by examining each `backup` calls inside a loop following a depth-first traversal order (step-1). For any `backup` call inside the loop, the algorithm identifies the loop's preheader and collects all of the loop's exits. It then proceeds by examining the address that the `backup` call performs checkpointing on. If the backup address is loop invariant and the `backup` call dominates *all* of the loop's exits, it hoists this `backup` call into the loop's preheader by making a replication of the `backup` call and inserting it into the loop's preheader (step-2), and then removing the original `backup` callsite (step-3) from inside the loop.

The *hoisting* optimization focuses on promoting `backup` calls that operate on *loop-invariant* (*loop-inv*) addresses into the loop's preheader. A *loop-invariant* address denotes an object whose address doesn't change within different iterations of a loop. This implies that all items that participate in the address-generation calculation are either a constant or defined outside of the loop and not being redefined within the loop.[3] For each `backup` operation inside a loop, the algorithm in Figure 4.9 checks whether (i) the `backup` address is loop invariant, and (ii) the `backup` calls dominates *all* exits of the said loop. If both conditions hold, the hoisting algorithm proceeds to move the `backup` call into the loop's preheader.

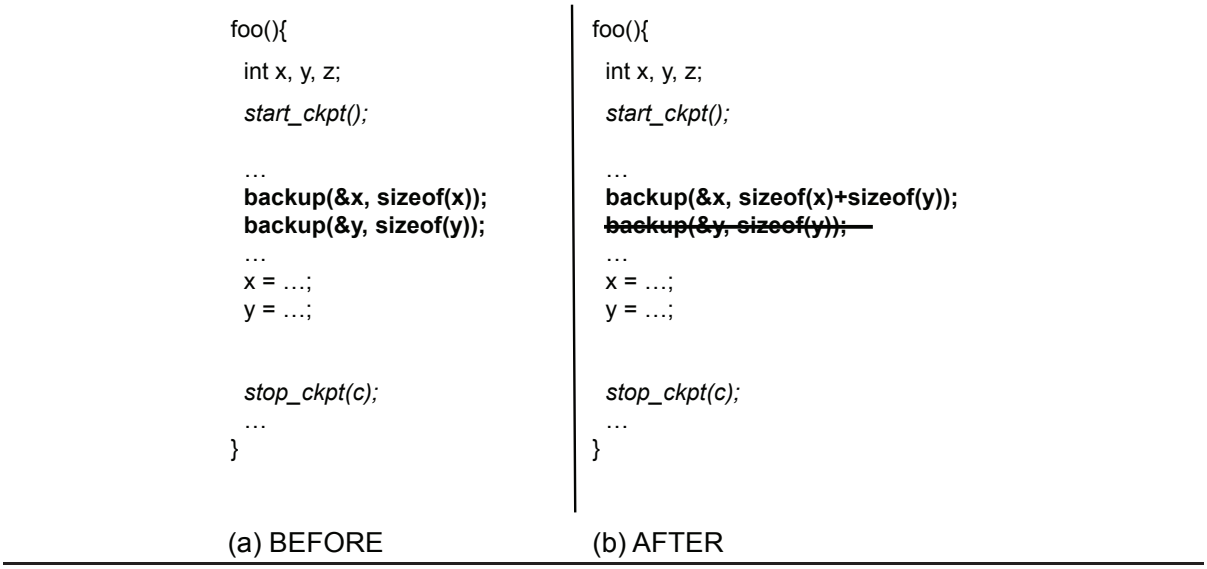


Figure 4.10: Simple aggregation transformation

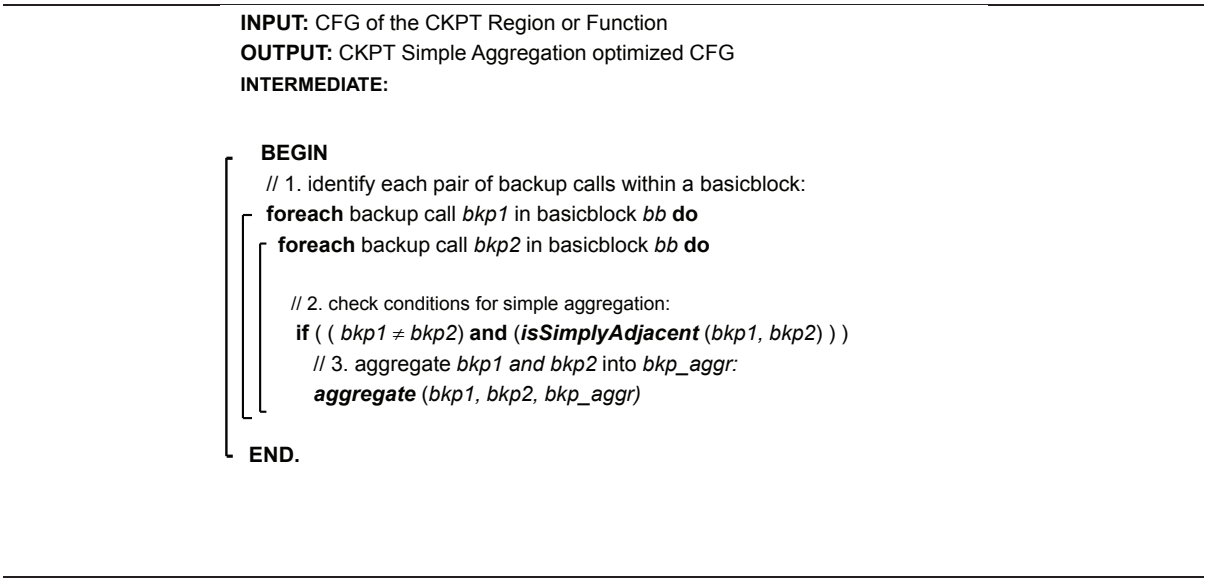


Figure 4.11: Simple aggregation algorithm

4.4 Aggregation

The *aggregation* optimization examines `backup` operations for variables whose addresses are adjacent in memory, and consolidates multiple `backup` operations over adjacent memory addresses into a single one that covers the entire memory range. An advanced version of aggregation can even potentially rearrange the layout of the variables to ensure that they are adjacent.

4.4.1 Basic Aggregation

Figure 4.10 shows that two individual `backup` operations (on variable `x` and variable `y`) have adjacent memory addresses through variable declaration and thus can be merged into a single `backup` operation, covering the entire memory range for both variables.[4] After the basic-aggregation transformation (in Figure 4.10-(b)), we notice that only one `backup` call remains, and this `backup` operation covers the memory range of both original participating variable `x` and variable `y`. Different architectures may allow the stack to grow in opposite directions. The simple-aggregation optimization is capable of performing some simple tests that identify the participating variable with *low* memory address. In Figure 4.10, the compiler recognizes that variable `x` is the one that has lower starting address on stack, and uses it as the starting address for the fused `backup` operation.

Simple aggregation optimization relies on the fact that the memory layout for both participating variables be adjacent. We call this *basic aggregation* optimization. Figure 4.11 presents its algorithm that attempts to aggregate two `backup` calls from each basic block that resides in a checkpoint-enabled region or a checkpoint-enabled function. Within the basic block, the algorithm compares each possible pair of addresses that `backup` calls operate on. Once it finds that there are two addresses that are adjacent in their memory layout, the basic aggregation proceeds by fusing the two participating `backup` calls into a single one that covers the entire memory range.

Basic Aggregation limits its search to only stack-allocated `backup` addresses whose layouts are already adjacent in memory (by default, through the variables' declaration order). It will not perform declaration reordering to harness more opportunities at this stage. We will soon introduce a more powerful and capable revision of the aggregation optimization that can do declaration reordering to reach more potential aggregation opportunities. Note that global variables may have different available

[2]Consider a branch that resides inside a loop, if this branch is never taken, hoisting a `backup` call inside this branch effectively introduces only checkpointing overhead. Multiple levels of loop nesting often exacerbates the situation.

[3]All loop-invariant code by default operates on loop-invariant addresses.

[4]Note that for a source-to-source transformation this is not necessarily a safe optimization as the back-end compiler may further rearrange the variable layout—an implementation in a single unified compiler would not have this problem.

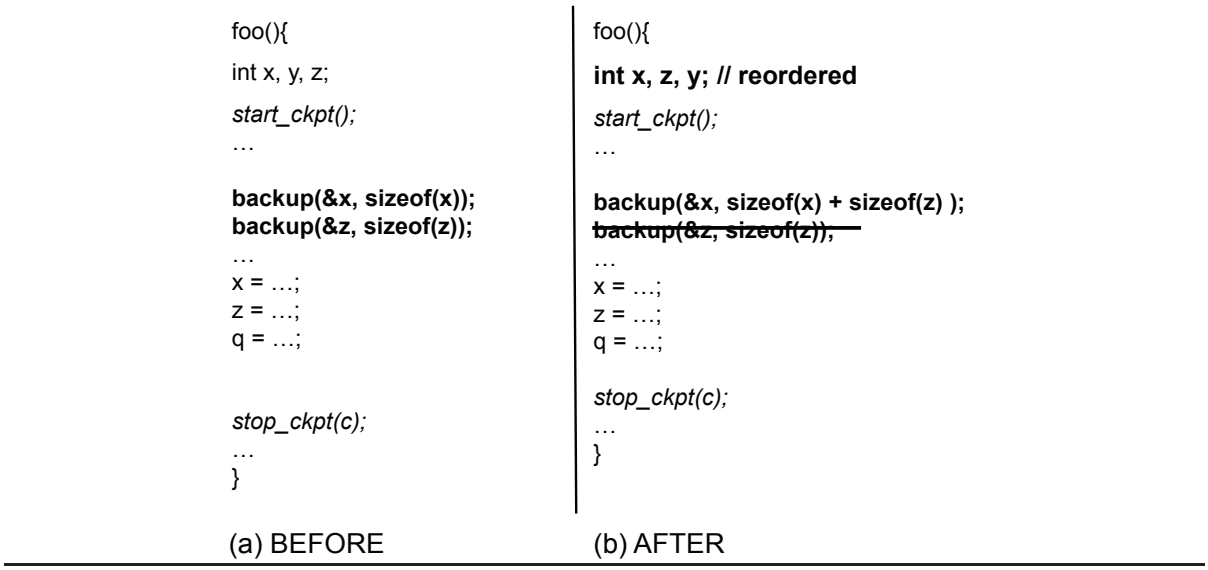


Figure 4.12: Advanced aggregation transformation

life spans and seemingly adjacent global variables may not co-exist at runtime. We thus exclude global variables from the candidates for aggregation optimizations and instead focus entirely on stack-allocated local variables.

4.4.2 Advanced Aggregation

Comparing with *basic aggregation*, a more complex form of aggregation optimization changes the order of variable declaration orders to make otherwise non-adjacent variables adjacent in the memory address layout. We present its transformation in Figure 4.12 and call this complex form of aggregation *advanced aggregation*. Figure 4.12 shows that two individual `backup` addresses (on variable `x` and variable `z`) can be merged into one adjacent address range, covering both participating variables (`x` and `z`). Note that in the original variables' declaration order, `x` and `z` are not adjacent (there is a variable `y` in between). Advanced aggregation optimization understands the memory layout order on stack, and rearranges the order (from: `x, y, z to: x, z, y`), such that the participating variables (`x and z`) occupy a range of adjacent memory address space after the declaration reordering.

In Figure 4.13, we provide an algorithm to conduct *advanced aggregation* analysis and transformation. It begins by collecting all stack-based addresses of participating `backup` operations on basic-block granularity (step 1). It then builds an internal stack model to examine the possibilities of making two stack addresses adjacent by changing the order of variable declarations (step 2 and step 3). If there are more than two participating addresses, the algorithm repeats itself based on its previous discovery and

```
INPUT: CFG of the CKPT Region or Function
OUTPUT: CKPT Advanced Aggregation optimized CFG
  BEGIN
   // 1. identify ALL backup addresses within a basicblock:
   foreach backup call bkp within basicblock bb do
     addr = getAddr(bkp); collect (addr);
   // 2. evaluate all possible variable ordering on stack:
   foreach bkp address order o1 do
     benefit b1 = evaluate (o1);
     save (pair<b1, o1>)
   // 3. sort according to benefits and pick the highest benefit:
   sort (collection<benefit, order>, DECR);
   pair<b, o> = select (collection<benefit, order>, MAXIMUM);
   // 4. perform aggregation:
   if ( b ≠ 0)
     select (bkp1, o); select (bkp2, o);
     aggregate (bkp1, bkp2, bkp_aggr);
  END.
```

Figure 4.13: Advanced aggregation algorithm

attempts to generate an adjacent order that covers as many participating addresses as possible. Note that in this case, the algorithm may produce multiple optimal solutions. (E.g., in Figure 4.12, the declaration order of `x, z, y; z, x, y; y, z, x; and y, x, z;` are all equally good, provided variables `x and z` are adjacent.) Finally, the advanced aggregation algorithm exhaustively evaluates each possible combination of variable declaration order, saves a cost-benefit pair for each order it evaluates, and selects the order that yields the highest benefit to conduct aggregation transformations. In case of ties, it selects the first saved order within all candidates that have the tie (step 4).

Aggregation reduces checkpoint buffer-management overhead by combining two or more adjacent `backup` operations into a single one, potentially reorder declarations to make them adjacent. From the perspective of checkpointing's buffer management, it consolidates multiple meta-data management records into a single record—a scheme that reduces both meta-data size and the number of meta-data records. However, there are no savings in the checkpointing's data buffer. Memory contents from multiple participating `backup` operations will be copied into the data buffer regardless of *aggregation*. As a result, this optimization can be treated as optimizing for meta-data efficiency. Due to possibility of different life-time span on global variables, we limit the candidates of aggregation optimization only to data allocated on stack and thus exclude all global variables from aggregation optimization.

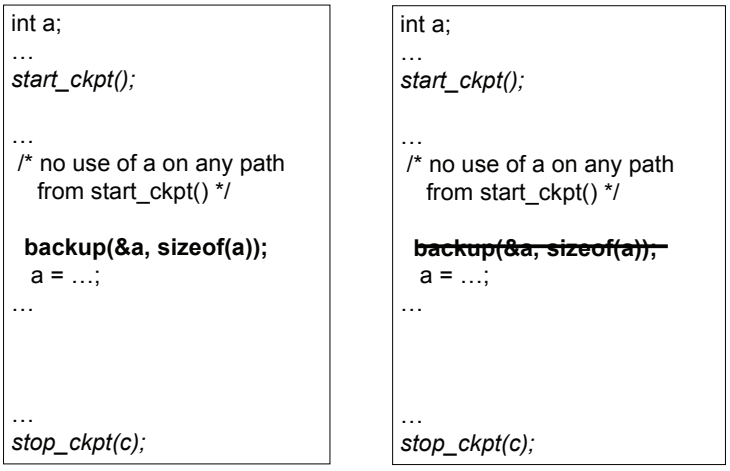


Figure 4.14: NRESE transformation

INPUT: CFG of the CKPT Region or Function
OUTPUT: NRESE optimized CFG
Intermediate: AliasSet *AS* = Φ*; cond1= false, cond2 = true*

BEGIN
// 1. analyze each possible backup call:
foreach backup call *bkp* within CKPT region or CKPT Function **do**
cond1= false, cond2 = true;
// 2. analyze backup address alias:
addr = ***getAddr*** (*bkp*);
AS = ***getAliasSet*** (*addr*);
if (*AS* == Φ) *cond1* = true;
// 3. check for read access on any path through linear scan:
foreach *instruction ins* between *start_ckpt* and *bkp* **do**
foreach *operand op* in instruction ins **do**
if (***use***(op, addr)) cond2 = false;
// 4. operate on NRESE:
if (*cond1 && cond2*) ***remove*** (*bkp*);
END.

Figure 4.15: NRESE algorithm

4.5 Non-Rollback-Exposed Store Elimination (NRESE)

Given any variable that is being written inside a checkpointing region, if along *any* path from the beginning of the region, there is **no** read from the variable and its address doesn't alias to anything (a.k.a. empty points-to set), an optimization can remove the respective `backup` operation on this variable without impacting checkpointing correctness. We call this optimization *non-rollback-exposed store elimination* (*NRESE*). To the best of our knowledge, this optimization has never been presented or explored previously in related contexts. Figure 4.14 presents an example of *NRESE* program transformation and we give its corresponding compiler algorithm in Figure 4.15.

Notice that the `backup` operation on variable `a` can be safely removed through *NRESE* , since there is no direct or aliased read from variable `a` along any path from the beginning of the checkpoint region to the respective store. For any checkpoint that commits successfully, whether we remove the `backup` operation on variable `a` is irrelevant. For any checkpoint that is aborted, the value of variable `a` is recomputed each time after abort and this re-computation is essentially independent of the current value of variable `a` (since there is no use of variable `a` on any path from the start of the region, the current value of variable `a` has no impact on generating its new value). As a result, the `backup` operation on variable `a` is not necessary and eliminating this `backup` operation has no impact on the correctness of the program's checkpointing behavior.

As shown in Figure 4.15, *NRESE*'s algorithm starts by examining each $\widehat{\text{backup}}$ operation in the given checkpointing region or checkpointing-enabled function (step 1). It then leverages existing pointer alias analysis framework that the LLVM compiler infrastructure provides and verifies that the address of the current `backup` operation is not aliased to anything (points-to set is empty) (step 2). It further examines each instruction between the start of the checkpoint region (`start_ckpt`) and the respective store instruction (on variable `a`) to ensure that there is no *read access* on the address of variable `a` [5] (step 3). If both conditions hold, the algorithm proceeds to remove the corresponding `backup` operation (step 4).

4.6 Dynamic Memory (DynMem) Optimization

Opportunities exist for any `backup` call that operates on dynamically allocated memory (heap). If the heap allocation is within the checkpointing region, the `backup` call operates on this allocated heap, and

[5] Since the generated `backup` calls are not part of the original code, the examining process refrains from checking the `backup` operation's function callsite, or any part of its internal implementations.

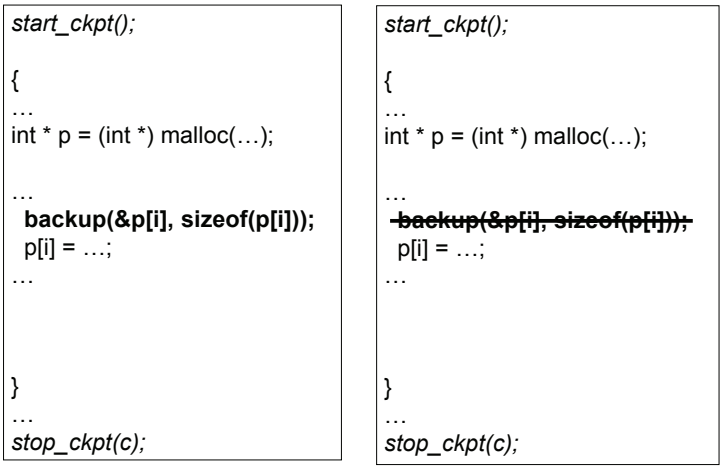


Figure 4.16: DynMem optimization transformation

INPUT: CFG of the CKPT Region or Function
OUTPUT: DynMem (Heap) optimized CFG

```
BEGIN
// 1. perform reaching definition analysis for each dynamic-memory
//    allocation site in the CKPT region or function:
foreach CKPT region r or CKPT-enabled Function f do
  dyn_allocs = collect (malloc, calloc, realloc, free, r/f);
  foreach dyn_alloc in dyn_allocs do
    DataFlowAnalysis (Reaching_Definition, dyn_alloc, r/f);

// 2. analyze backup address alias:
foreach backup call bkp within a CKPT region r or a CKPT Function f do
  foreach each dynamic allocation site dyn_alloc do
    addr = getAddr (bkp)
    if ( isDynMem (addr) and reach (dyn_alloc, bkp, r/f) and
         dominate(dyn_alloc, bkp)) then
           remove (bkp)
END.
```

Figure 4.17: DynMem optimization algorithm

the heap allocation reaches and dominates the `backup` operation, then the `backup` call operating on the heap-allocated memory can be safely eliminated.

Figure 4.16 demonstrates the process of removing a `backup` operation on heap-allocated array variable `p`. Because the heap allocation happens within the checkpoint region, the heap-allocated contents have no memory footprint before the checkpoint starts. Since there is no need to checkpoint memory contents that are irrelevant (or invisible) with respect to the checkpointing process, it is always safe to remove this `backup` call (`backup(p[i], sizeof(p[i]))` that operates on heap-allocated memory `p`.

We present the dynamic-memory (*DynMem*) optimization algorithm in Figure 4.17 that contains two major steps. Step one performs reaching-definition (*RD*) analysis for each dynamic-memory allocation site residing in a checkpoint region or checkpoint-enable function. A dynamic memory allocation site (via `C` interface) includes three function calls: `malloc, calloc` and `realloc`; and a dynamic memory reclamation site (via `C` interface) includes just one function call: `free`. Each dynamic memory allocation site starts a new dynamic-memory range on its starting address and length; while a dynamic memory reclamation site ends its respective dynamic-memory range. Notice that the special call `realloc` is both an allocation and reclamation site. It ends the current active dynamic memory range and immediately starts a new one with the same starting address but a potentially different length. A `realloc` call with zero length effectively terminates a dynamic-memory range as well. Step two analyzes each applicable `backup` call that operates on heap-allocated memory. If the `backup` call operates on a dynamically-allocated memory within the checkpointing region or checkpoint-enabled function, the dynamic memory allocation site inside the respective region or function, and the `backup` operation is within the reach of an active dynamic-memory coverage region, the algorithm can proceed to remove the respective `backup` operation.

Our *DynMem* optimization focuses on `backup` calls operating over dynamically allocated memory whose allocation site is within the respective checkpointing region or checkpoint-enabled function. Under such conditions, the dynamically allocated memory is invisible before the checkpointing region begins. It is safe not to `backup` memory content that is not visible for the current application from the persective of checkpointing. However, the situation can be vastly different if the dynamic memory allocation is outside of the checkpointing region or checkpoint-enabled function, because the dynamically-allocated memory is visible even before the checkpoint region begins. Skipping backup operations on writes to this memory will produce inconsistent memory states after a checkpoint abort operation. Thus, the *DynMem* algorithm refrains from eliminating `backup` operations on dynamically

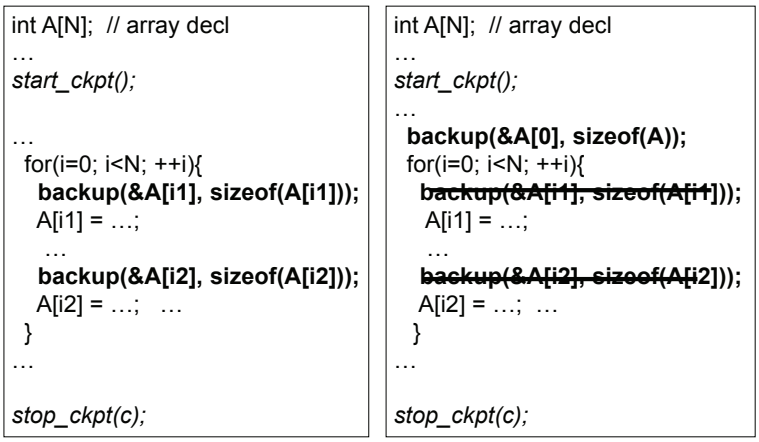


Figure 4.18: Array optimization transformation

allocated memory whose allocation site is out of the current checkpointing region or checkpoint-enabled function.

4.7 Array Optimization

More interesting opportunities exist among `backup` calls that operate on array-based addresses inside a loop, and we give an example in Figure 4.18. Both writes into array-based addresses (`A[i1]` and `A[i2]`) are correlated with the loop's index (variable `i`). *Hoisting* cannot remove any backup redundancy because the address is not loop-invariant in either case. However, keeping both `backup` operations inside the loop has potentially high overhead. With a large trip count and a relatively small array size, the amount of accumulated back-up contents can potentially be bigger than backing up the entire array. In such cases, it is normally beneficial to merge multiple backups on individual array elements into a single backup operation, potentially covering a continuous sub range of the array's memory space or even the entire array. Since the array's starting address is loop-invariant, moving this single `backup` operation out-of the loop (into the loop's preheader) can further eliminate any `backup` call that operates within the array's address range inside the loop. Thus in Figure 4.18(b), both of the original `backup` calls are consolidated into a single `backup` call that covers the entire memory range of the given array (`sizeof(a[])`). Because the array's starting address is loop invariant, this single `backup` call is further hoisted out of the loop and placed into the loop's preheader. We call this transformation *array optimization (ArrayOpti)*.

```
INPUT: CFG of the CKPT Region or Function
OUTPUT: Array optimized CFG
INTERMEDIATE: store_instances = 0, array_size = 0, trip_count = 0,
              tolerance_factor = 0.5, array_addr = 0, bArrayOpti[..] = false;
BEGIN
  // 1. obtain backup call and loop-related info for array analysis:
  foreach backup call bkp in loop L do
    trip_count = getLoopInfo (L, TRIP_COUNT); addr = getAddr (bkp);
    if (isArrayAddr (addr, unique) ) do
      store_instances += getLoopInfo (L, STORE_INSTANCE, addr);
      array_size = getArraySize (addr); array_addr = getArrayAddr (addr, 0);
  // 2. perform array analysis and optimization:
  if ( trip_count * store_instances >= tolerance_factor * array_size) do
     array_bkp = create_bkp ( array_addr, array_size);
     insert (array_bkp, L.getPreheader () ); bArrayOpti[array_addr] = true;
  // 3. remove any participating backup call on the reference array:
  foreach backup call bkp in loop L do
    if ( isArrayAddress (getAddr(bkp), array_addr)) and
       (bArrayOpti[array_addr] ) do remove (bkp)
END.
```

Figure 4.19: Array optimization algorithm

We present an algorithm for array optimization in Figure 4.19. It considers not only the *array size*, the loop's *trip count* and *store intensity*, but also a *tolerance factor* that a user can control through command-line options. Writes to non-adjacent regions inside the array may happen when the program executes inside a loop, thus the tolerance factor specifies the amount of checkpoint buffer storage tradeoff that a user may allow, in return for improved checkpointing efficiency.

We present detailed analysis of the *array optimization* algorithm below.

Let *trip_count* denote to the number of iterations for a given loop where we intend to perform array optimization. If the loop's trip count cannot be obtained through static analysis at compile time (missing either known lower bound or upper bound of the given loop), we use a numerical value 10 as a best-effort estimate for the loop's default trip count.

Let *store_instances* denote to the intensity of all static `backup` operations that a loop contains. Within each iteration, we give one point for the *store_instance* of any `backup` operation that dominates all of the loop's exits; and 0.5 otherwise (best-effort compile-time estimates for branch behavior). The loop's *store_instances* is the accumulation of each *store_instance* associated with individual `backup` operations within the loop's body. This represents a best-effort static estimate of the `backup` operation's execution frequency and their potential impact on the array within each loop iteration.

For example, in Figure 4.18(a), both `backup` operations dominate the loop's exits, thus they each

have a *store_instance* value of *one* within each iteration. Assuming these are the only `backup` operations within the given loop's body, thus the loop's *store_instances* is value *two*.

Let *array_size* denote to the number of elements in a given array,

and
Let *tolerance_factor* denote to a degree of tolerance that a user agrees to trade off memory for improved checkpointing efficiency.

The *tolerance_factor* is a floating-point value between *zero* and *one*. Value *one* of a *tolerance_factor* indicates no tolerance at all, value *0.5* for a *tolerance_factor* indicates that a user agrees to trade off 50% of the buffer size in return of improved checkpointing efficiency, whereas value *zero* of a *tolerance_factor* indicates complete tolerance without any constraint. In our *array optimization* `LLVM` compiler transformation, we preset a default *tolerance_factor* of *0.5*. This default value can easily be overwritten using a command-line option.

We abstract the key component of the array-optimization algorithm into the following equation:

$$f = trip\_count * store\_instances - tolerance\_factor * array\_size \tag{4.1}$$

The decision on whether to perform array optimization is based on the algorithmic evaluation result of function f, as:

$$f = \begin{cases} \geq 0 & \text{yes, perform array optimization} \\ < 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The value of *store_instances* statically estimates the number of occurrence that the program conducts `backup` operations on a per-iteration basis, so *trip_count * store_instances* predicts the total number (size) of `backup` operations across the entire loop. At the same time, *tolerance_factor * array_size* approximates the size of the array that the `backup` operation(s) can potentially have impact on. Thus equation 4.2 conducts comparative evaluation between the predicted total `backup` size and the estimated size of the array that also includes a user's willingness to trade off checkpointing buffer space for improved checkpointing efficiency. When the predicted total `backup` size is bigger than the estimated size of the affected region of the array, the algorithm decides to perform array optimization.

We present the complete array optimization algorithm in Figure 4.19, which is a loop-based `LLVM` transformation that contains three major steps. For each loop that contains at last one `backup` call that operates on an array-based address, step one identifies the `backup` operation and obtains its starting address. If the starting address is part of an array-based address space, the algorithm proceeds by accumulating and estimating its *store_instances*. Step one also extracts related information from the loop and the array involved, including the loop's trip count, array size and the array's starting address. Step two performs the key array-optimization analysis. It combines all *store_instances* for any `backup` call that operates on the same array's address range inside the loop's body. It then evaluates the algorithm according to our previous discussion in equation 4.1 and 4.2. If the result is positive, the array optimization algorithm creates a new `backup` call that operates on the starting address of the array, and covers the entire memory address range of the said array (backup the entire array). It then inserts this newly created `backup` operation into the preheader basic block of the loop, and marks *true* for the (loop, starting address, length) tuple. Step three is optional and is active only upon at least one successful *array optimization* transformation in step two. When this condition holds, it once again scans the original loop's body and identifies each `backup` operation whose starting address and length fall into the array that has just been promoted into the loop's preheader. Within the loop's body, it eliminates each identified `backup` call that satisfies the given condition.

Array optimization seeks the best-possible checkpointing performance by trading off memory for potentially reduced `backup` overhead. When array optimization proceeds, it generates a single `backup` operation that potentially covers the entire array. It then places this newly generated `backup` operation outside of the loop before the loop begins (into the loop's preheader). This renders any `backup` operation on any of this array's valid address range irrelevant, and thus be eliminated from the loop's body. The *tolerance_factor* is a user-controllable knob that represents the amount of tolerance that a user allows, in return to trade off memory for improved checkpointing efficiency. It is a floating-point value spanning between the range of 0.0 and 1.0, with a default set to 0.5. This indicates that a user agrees to have a 50% backup buffer memory overhead on the affected array, in return for eliminating all `backup` calls that operate on that array within the loop.

Due to the use of *tolerance factor* and its checkpointing efficiency trade-off, array optimization can sometimes be storage inefficient. Thus a programmer has the flexibility to fine tune the behavior of array optimization through a command-line interface on tolerance factor for important loops, aiming for maximum checkpointing overhead reduction. Since we can dynamically grow the checkpointing buffer in memory, we consider the memory trade off worth the effort when the main goal is to improve

checkpointing efficiency.

Array optimization limits its matching capacity within the `LLVM` array-type IR. Thus it is currently incapable to handle any pointer arithmetic operations that uses array syntax. The `LLVM` frontend will turn such operations into direct memory load/store IRs rather than array IRs. We will address this limitation in our future-work section.

4.8 Miscellaneous Optimizations

Miscellaneous optimizations are mainly used for setup and cleanup of the optimization framework. For example, `Inlining` inlines all special system handling routines. This helps to unbox the `backup` operations originally hidden inside system handling routines and allows the optimizer to instead focus exclusively on analyzing and eliminating only the `backup` operations. `Pre Optimize` and `Post Optimize` perform miscellaneous clean-up operations (e.g., remove zero-length `backup` calls). Any future or newly-discovered clean-up or maintenance operations will likely fit into the miscellaneous optimizations category.

4.9 Summary

In this chapter, we introduce a large and comprehensive checkpointing optimization framework that targets checkpointing overhead reduction through aggressive compile-time analysis and transformations. It examines checkpointing overhead from different program aspects and attempts to remove as much checkpointing overhead as possible while still maintaining the correct behaviors of checkpointing-enabled programs. For each individual optimization, we present a detailed transformation example with discussions on its respective algorithm. In particular, we introduce *NRESE* optimization that is brand new to our best knowledge. We will provide testing details of each individual optimization as well as combined efforts of all available optimizations in section 7.7.

In the next chapter, we will discuss details of the checkpointing buffering schemes with focus on both buffer efficiency and runtime performance.

Chapter 5

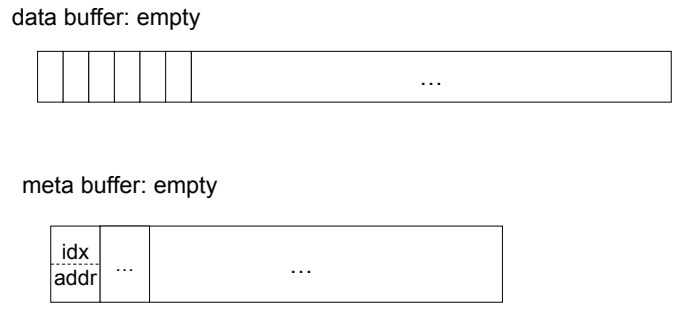
Checkpoint Buffer Implementation

Our fine-grain checkpointing scheme buffers program changes into a checkpointing buffer while the program proceeds. As we will demonstrate, the buffer design and implementation has a critical impact on checkpointing efficiency. In this chapter, we introduce several designs of checkpointing buffering, analyze the trade-offs among them, and discuss the important decisions made to balance execution efficiency and storage utilization.

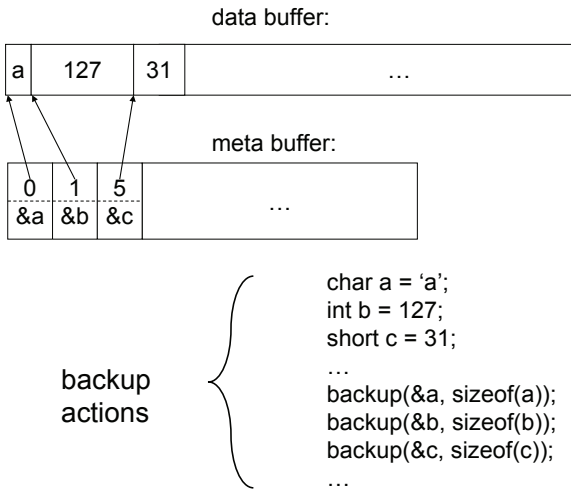
5.1 Undo-log vs. Write Buffer

The most important design decision in a checkpointing scheme is the approach to buffering: whether it will be based on *write-buffer* [28, 43] or alternatively an *undo-log* [32, 47]. A write-buffer approach buffers all writes from main memory, and therefore requires that the write-buffer be searched on every read. Should the checkpoint commit, the write-buffer must be committed to main memory; should the checkpoint fail, the write-buffer can simply be discarded. Hence for a write-buffer approach the checkpointed code proceeds more slowly, but with the benefit that parallel threads of execution can be effectively checkpointed and isolated (e.g., for some forms of optimistic transactional memory [28,45]). An undo-log approach maintains a buffer of previous values of modified memory locations, and allows the checkpointed code to otherwise read or write main memory directly. Should the checkpoint commit, the undo-log is simply discarded; should the checkpoint fail, the undo-log must be used to rewind main memory. Hence for an undo-log approach the checkpointed code can proceed much more quickly than a write-buffer approach.

We conducted a preliminary study to compare runtime performance of undo-log and write-buffer approaches, while checkpointing loops of integer-sorting applications. Compared with undo-log, the



(a) Empty checkpointing buffers



(b) Checkpointing buffers populated with backup data

Figure 5.1: One-Dimensional array buffer scheme.

write-buffer scheme suffered severe performance penalty, usually ranging between 10X and 100X. In this thesis, since we are considering only a single thread of execution with emphasis on performance, we thus focus our future discussions only on the undo-log buffer scheme.

5.2 One-Dimensional Array-Based Undo Log

As discussed in Section 3.4.1, the interface of conducting a checkpoint backup operation is through a `backup` function call, which takes as arguments a pointer to the start-to-copy address and its to-backup length in bytes. We give the full checkpointing APIs in appendix A. Figure 5.1(a) illustrates our initial design of an undo-log buffer based on one-dimensional data array (1-*D array buffer*). In this scheme, we divide the undo-log buffer into two structures: (i) a data buffer which is a concatenation of all backup data values of arbitrary sizes; and (ii) a meta-data buffer (meta buffer) which stores pairs of the starting address and length of each backup data record (*meta data*) that appears in the data buffer. As an example, Figure 5.1(b) shows the contents of the undo-log after executing three `backup` calls to variables `a, b`, and `c`, respectively. Notice that both the data buffer and the meta-data buffer are now populated, with an address-value pair in the meta buffer maintaining bookkeeping information for each data record appears in the data buffer. When a checkpoint commits, we simply move the data buffer and meta buffer pointers back to the start of each buffer—effectively discarding the contents in both buffers. When a checkpoint must be rewound, we use the address-value records in the meta buffer to walk backwards through the data buffer, replacing each backed-up memory location with its original value.

The one-D array buffer lays down its contents linearly in memory. The buffer-append action dominates all buffer activities. Buffer-append operations always occur at content-insert location and result in linear growth of both data buffer and meta-data buffer. Thus the undo-log scheme based on one-D array buffer benefits from both extremely simple design and cache friendliness. [1] However, its main drawback is that it can suffer from data redundancy, since multiple versions of the same backed-up memory location might reside in the buffer. A one-D array could be maintained in a redundancy-free manner by performing a linear search prior to each `insert` operation. Alternatively, the one-D array buffer could be maintained in a sorted order. Preliminary evaluation found that the performance penalty of linear scan for inserting is between 50X to 100X. This negative performance impact is far beyond our level of tolerance for checkpointing on performance-sensitive applications.

[1] Since one-D array buffer grows linearly, for buffer-insert operations, one cache miss will lead to cache hits for up to cache-line size buffer length. But for hash-table based buffers, due to the unpredictability of a hash-node's address, each insert operation is more likely to be a cache miss.

	one-dimensional array	hash table
insert	**O(1)**	**O(N)**
search	O(N)	O(N)
delete	O(N)	O(N)
reset	O(1)	O(N)

Table 5.1: Comparison of worst-case buffer-operation efficiency

For a fair comparison with one-D array implementation, we need an alternative implementation that natively and implicitly incorporates redundancy elimination when inserting data records. We select a hashtable-based buffering design for this purpose.

5.3 HashTable-based Undo Log

Rather than searching or sorting a linear list, demonstrated to be infeasible in the previous section, another method worthy of consideration is to maintain a hash table of backup entries. A hash table (*HT*) is a data structure that maps keys to values through a hash function. In the context of the undo-log buffer design, it maps an address (key) to its associated memory content (value). Popular hash-table designs implement hash bucket list with chaining. When multiple different addresses map to the same hash-table target location (hash collision), the hash nodes are chained together into a linked list. For any address that is mapped into a non-empty linked list, a search-based traversal is necessary to identify the correct data and resolve any conflict. We implement an undo-log scheme using both 1-D array and hash table. For an easy comparison, we present the worst-case buffer operation efficiency in Table 5.1.

Under the one-D array implementation, both `insert` and `reset` operations are $O(1)$. `Insert` will always append at the end of array and `reset` only adjust buffer counters, so they both take constant time to complete. Since the one-D array buffer is not sorted, `search` and `delete` will need to iterate linearly across the entire meta buffer to identify the matching index in the data buffer, thus they are both $O(N)$. Under the hashtable-based implementation, each operation needs to map its key (address) to a value (hash node) that may reside on a non-empty linked list. If the desired hash node doesn't exist, we need to create the hash node and insert it to the right location. If a non-empty linked list exists after the hash mapping, the scheme needs to iterate over each available node on the linked list. Thus under a hashtable-based undo-log buffer scheme, all worst-case operations have cost of $O(N)$.

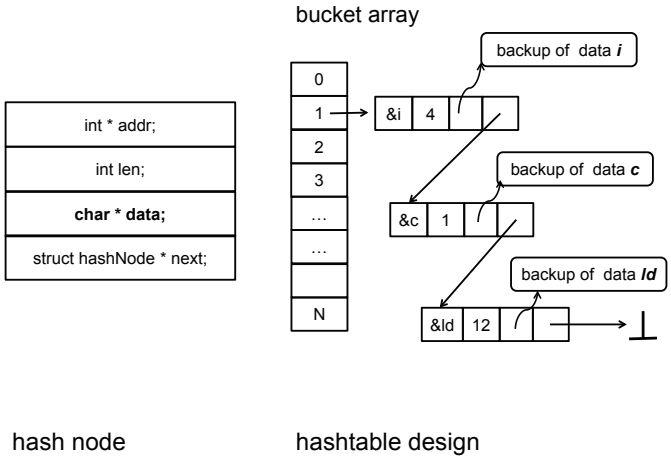
```
int i;
char c;
long double ld;
…
start_ckpt();
…

  backup(&i, sizeof(i));
  i = …;
   …
  backup(&c, sizeof(c));
  c = …;
  …
  backup(&ld, sizeof(ld));
  ld = …;
  …
…
stop_ckpt(c);
```

Figure 5.2: Sample of checkpoint-enabled code

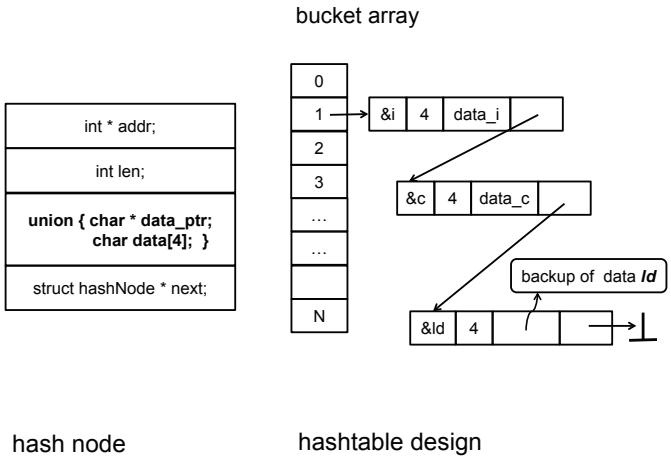
Notice that *insert* is the most frequently used operation in checkpointing. Given data available in Table 5.1, we can safely predict that an array-based buffer would normally outperform (higher runtime efficiency) its hashtable-based counterpart, at the expense of buffer storage size (potential redundancy). We will present more details of further analysis on buffer efficiency and storage trade-offs later in this chapter.

A hashtable-based undo-log scheme always conducts an implicit search either before inserting a new node (if an address does not already exist) or attempting to identify the right value (if an address exists in the linked list already), thus it has perfect storage behavior (0% redundancy). When it needs to insert a new node, it will perform a series of actions—allocating memory for the new hash node, populating the new node with data, and linking the node at the proper location of the linked list. All of these actions involve runtime overhead and will negatively impact checkpointing performance. One way to alleviate the overhead is to reduce the total number of needed dynamic memory allocations when creating a new hash node. This motivates various hash node designs with different performance trade-offs.

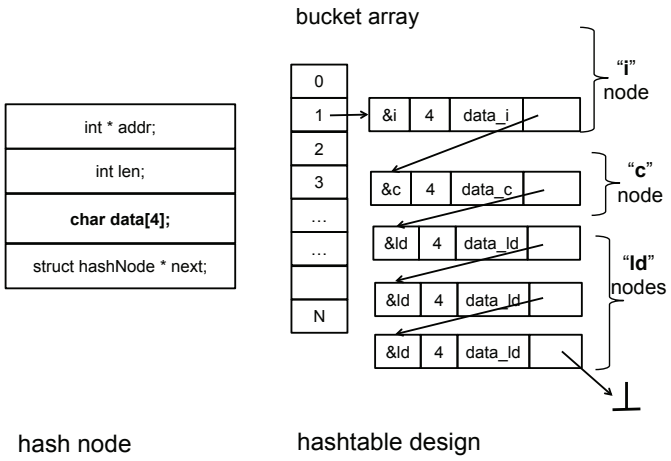
As illustrated in Figure 5.3, we consider three hash-table designs that are based on the options for the different designs of hash nodes, including: *pointer-to-data* (*PTD*), that stores a pointer to dynamically-allocated data storage; *inline/union* (*union*), that stores a union field that can be used either to directly store a 32-bit value inline, or instead as a pointer to dynamically-allocated data storage larger than 32 bits; and *fixed-size* (*fixed*), that always stores 32 bits of data per node and requires a list of nodes to store data values longer than 32 bits.



(a) Hashtable with pointer-to-data node



(b) Hashtable with inline/union node



(c) Hashtable with fixed-size node

Figure 5.3: Design options for an undo-log implementation.

5.3.1 Pointer-To-Data (PTD) Node

Figure 5.3(a) presents our basic hash node design: *pointer-to-data* (*PTD*) that stores a pointer to dynamically-allocated data storage. *PTD* is a structure that contains all necessary fields to facilitate a backup operation: a `char*` field to store the backup address, an `int` field to store the backup length, a `char*` field to store the pointer of explicitly allocated memory that holds the backup data, and a `struct hashNode * next` pointer to maintain a singly linked list with other nodes. Note that the backup data is allocated and managed through the explicit `char* data` field. This indicates that for **each** new *PTD* hash node, there will be **two** explicit dynamic memory allocations: one for the hash node itself, another for the explicit data field.

Figure 5.2 gives a checkpointing-enabled code sample to illustrate the impact of different hash-node designs on checkpointing buffer efficiency. This sample code needs to perform `backup` operations on three addresses with various backup lengths: four bytes for variable `i` (`int` type), one byte for variable `c` (`char` type), and 12 bytes for variable `ld` (`long-double` type). To simplify the scenario, we perform a `backup` operation on each unique address once.

Figure 5.3(a) shows one possible (worst-case) memory layout after performing all three `backup` operations. Since each hash node needs two dynamic memory allocations (*mallocs*), there is a total of *six* mallocs that are necessary to satisfy the requirements of accommodating all three hash nodes.

5.3.2 Inline/Union Node

Figure 5.3(b) presents an improved hash node design using inline/union node (*union*), that stores a union field that can be used either to directly store a (up-to) 32-bit value inline, or instead as a pointer to dynamically-allocated data storage larger than 32 bits. Comparing with *PTD*, a *union* type hash node overlaps a pointer to explicitly allocated static data with a fixed-size array of four bytes. When the backup data is less than or equal to four bytes, the data array will be used; otherwise when the backup data is larger than four bytes, *union* node will fall back to explicitly allocating and managing heap storage for the data (same as *PTD*).

We consider *union* node an optimization to reduce dynamic memory management overhead through using statically allocated buffer as appropriate. When the majority of `backup` operations are less than four bytes long (true for our test applications that are dominated by integer arithmetic operations), *union* node design improves checkpointing efficiency by reducing the number of needed dynamic memory allocations.

5.3.3 Fixed-Size Node

The third hash-node design aims to completely eliminate the need for a separate data pointer, as well as explicitly allocating memory and managing backup data. Figure 5.3(c) presents this *fixed-size* (*fixed*) hash node design that always stores (up-to) 32 bits of static data per node and may require a list of nodes to store data that is longer than 32 bits. *Fixed* hash node is a simplified version of *union* – it removes the overlapped data pointer and allows only data storage up to four bytes per node. If the backup data is larger than four bytes, the backup scheme will need to have a linked list of *fixed* nodes to fully accommodate the backup data.

5.3.4 Buffer Efficiency Analysis

In this section, we present an analytical comparison among the 3 different hash-node designs.

Let M be the number of bytes that a `backup` operation needs to copy;

Let C be the maximum number of bytes that a single *fixed* node can accommodate;

Let N be the number of *fixed* node(s) that need to successfully accommodate the `backup` operation, we have

$$N = \lfloor M/C \rfloor + \lceil (M\%C)/C \rceil \tag{5.1}$$

It is clear that when $M > C$, $N \geq 2$.

Thus for any `backup` operation whose data length is bigger than the maximum amount of storage that a single *fixed* node can accommodate, we need at least 2 *fixed* hash nodes to complete the `backup` operation. We conduct an empirical comparison among the three different hash node designs when conducting three `backup` operations on the code sample provided in Figure 5.2. We present the result on the number of needed hash nodes and the number of required dynamic memory allocations in Table 5.2. The results suggest that the hash table designs based on *union*-type hash nodes have the lowest number of malloc requests. Next we present a more generalized analysis.

Let A_1 be the total number of `backup` operations on i (integer type);

	PTD	Union	Fixed
# hash nodes	3	**3**	5
# data pointers	3	**1**	0
total # mallocs	6	**4**	5

Table 5.2: Dynamic memory allocation overhead comparison among three different hash designs.

	PTD	**Union**	Fixed
# hash nodes	$A_1+A_2+A_3$	$A_1+A_2+A_3$	$A_1+A_2+3A_3$
# data pointers	$A_1+A_2+A_3$	$0+0+A_3$	$0+0+0$
total # mallocs	$2A_1+2A_2+2A_3$	$A_1+A_2+2A_3$	$A_1+A_2+3A_3$

Table 5.3: Generic analysis on dynamic memory allocation overhead comparison

Let A_2 be the total number of `backup` operations on *c* (char type);

Let A_3 be the total number of `backup` operations on *ld* (long double type);

Table 5.3 demonstrates the analysis results on the total number of needed dynamic memory allocations (*mallocs*) among three different hash node designs under the conditions of A_1, A_2 and A_3. It is apparent that *union* hash-node design has the lowest total number of mallocs, verifying our claim. The reduction on the total number of malloc calls is mainly the result of overlapping the data pointer with statically allocated memory inside the *union* design given the appropriate conditions where the `backup` lengths fit.

The *PTD* node is the fundamental design that maintains a dedicated pointer to dynamically allocated data per node, thus it has the highest overhead on the number of required dynamic memory allocations. *Union* node improves dynamic memory usage by overlapping the data pointer with static storage. This can remove a large number of dynamic memory allocations for data storage without increasing the number of needed hash nodes if a significant amount of backup data is less than or equal to four-bytes long. We expect it to deliver the best runtime performance among all three hash node designs. The *fixed* node completely removes the need to maintain dedicated data pointers. However, it also increases the

number of needed hash nodes, as well as the necessary expense to manage and maintain data consistency. Thus we expect that a hashtable composed of *fixed* nodes may generally improve over the baseline with *PTD* nodes when the data length is short, but may also have poor performance for `backup` calls when average backup data lengths is longer than the pre-determined size (4B in the case of a *fixed* node). In general, the *fixed* type shall never perform better than the *union* type.

5.4 Redundancy Rate

To compare the potential undo-log buffer designs across all possible implementations, we introduce a measurement called redundancy rate (*RR*).

Let $Access(R)$ denote the total number of backups of a particular variable R that is written at least once within the checkpointing region, the redundancy rate (*RR*) for the region that contains R can be defined as

$$RR = \frac{\sum_1^n (Access(R_i) - 1)}{\sum_1^n Access(R_i)} \tag{5.2}$$

where n is the total number of unique addresses that are checkpointed within the region. *RR* quantifies the amount of checkpointing redundancy as a floating point value between zero and one. In an ideal region where each unique variable address is checkpointed exactly once, its *RR* rate will be zero. The higher the *RR* rate, the more redundancy remains in the given checkpointing-enabled region.

The redundancy rate is a metric to measure the amount of redundancy available in a checkpointing region. It is especially helpful to evaluate various optimization's effectiveness after performing certain type of redundancy eliminations. It is also a good indicator to estimate the remaining amount of redundancy and can serve as a guideline for future improvement.

5.5 Evaluation

Recall from Section 5.2 and Section 5.3 that we have a total of four different buffer implementations for conducting software checkpointing based on undo-log scheme: one-D array, *PTD* hash table, *union* hash table, and *fixed-size* hash table. Former analysis suggests that the lookup-free one-D array buffer has the highest runtime performance despite existing data redundancy. By always appending to the end of the buffer, one-D array has extremely simple and efficient buffer-insert operations, excellent cache

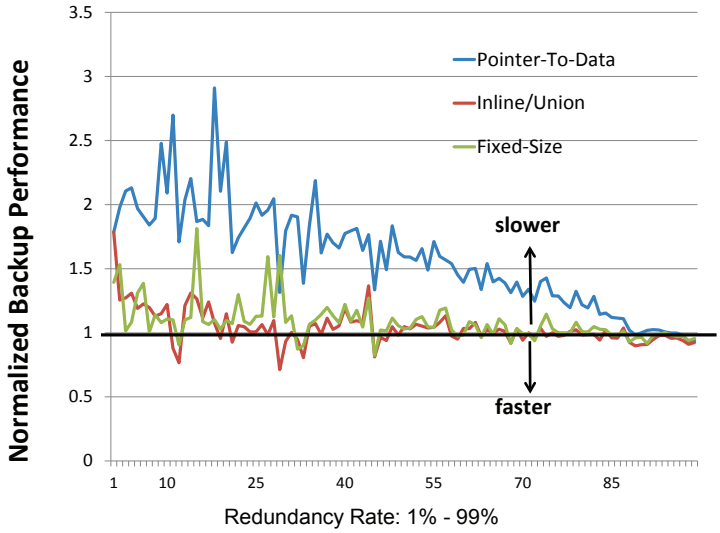


Figure 5.4: Performance comparison of buffer implementations

behavior, and thus avoids all cache-related problems associated with hashtable-based implementations. This is true when the redundancy is low and overhead from hash-table lookup and increased cache miss dominates. However, under conditions of high or extremely high redundancy, things may well change. Thus we are interested in discovering and understanding the relative performance of all buffering schemes under a wide range of all possible redundancy rates.

To conduct quantitative comparisons among different undo-log buffer implementations, we develop a simple micro benchmark application that performs intensive `backup` operations at random locations of a large integer array. By fixing the number of unique `backup` addresses (array elements) and varying the total number of `backup` operations, we can easily obtain any desired redundancy rate.

In Figure 5.4 we present relative performance of all available buffer implementations using the micro benchmark. We vary the micro-benchmark's access patterns to produce a wide range of redundancy rates and report checkpointing performance comparison result. The x-axis represents redundancy rate from 1% to 99%; the y-axis is the relative checkpointing performance of the three hashtable-based buffer implementations. Performance data is normalized to that of using an one-D array buffer, thus all curves that reside beyond row-one are considered *slower*. The figure represents checkpointing operations with 1024 unique backup addresses, with only four-byte backup length (due to the limitation using a static array).

Overall, the implementation based on one-D array buffer almost always outperforms any hashtable-based solution (higher than one on y axis). All three curves (hashtable-based implementations) converge

at a very high *RR* rate (close to 95%). When the redundancy rate increases, the performance difference among different backup schemes decreases. The three different hashtable-based implementations have perfect storage behavior; however this comes at a performance cost, mainly due to poor cache locality of link-list accesses within the mandatory hashtable lookup operations. *Union* and *fixed* are both heavily optimized for dynamic memory management, thus their performance is considerably and consistently better than *PTD*. When the redundancy rate is extremely high ($\geq$ 95%), performance differences of various buffer schemes diminish. Although undesirable, the huge amount of redundancy in the one-D array buffer amortizes all cache-related overhead under such conditions. In practice, we expect no real-world applications to exhibit such extremely-high redundancy rates. Because of the superior performance of one-D array buffer implementation over a wide range of possible redundancy rates, we select it as the default implementation for all undo-log buffering for the remainder of this thesis.

5.6 Summary

We describe undo-log and write buffer schemes for implementing checkpoint buffering, but focus on undo-log approaches because they are better-suited to single-thread roll-back applications like checkpointing. We discuss a total of four different undo-log buffer implementations: one-D array, *PTD* hash table, *union* hashtable, and *fixed* hashtable. We define *redundancy rate* as an evaluation metric and evaluate the behaviors of all four available buffer schemes on a `backup`-intensive micro benchmark. We conclude that despite data redundancy, the one-D array buffer implementation is the most efficient due to its low runtime overhead. For all three different hashtable-based designs, *union* type hash node is a clear winner because it successfully minimizes dynamic memory management related overhead.

For the next two chapters, we will begin to introduce three interesting applications that leverage the efficient software-only checkpointing support to gain distinct features. In particular, in chapter 6, we will present the *1st* key application—tolerating delinquent loads via checkpointing.

Chapter 6

Tolerating Delinquent Loads via Checkpointing

A delinquent load (*DL*) [15, 49] is a particular type of memory load in a program that frequently misses in a cache—typically the last-level cache on-chip. For many applications, we observe that DLs from a small number of source-program locations contribute a large fraction of all last-level cache load misses. Hence DLs, should they be reasonably persistent across target architectures, may pose an interesting checkpointing application. In this chapter, we introduce our first checkpointing-enabled application—overlapping execution with delinquent loads.

6.1 Overview

Figure 6.1(a) illustrates the challenge presented by a DL: the L2 miss latency for a DL can be lengthy, and the computation that follows the DL (`work()`) likely depends on the DL's result value (`x`). Thus the execution time that involves a `DL` simply becomes the accumulation of both the DL's latency, and the cycles of the `work` that needs the precise value from the DL. Rather than allowing a system to staying idle and waiting for the DL's value to return from the long-latency main memory system, Figure 6.1(b) provides an overview of the techniques to tolerate a DL by overlapping the DL's miss latency with speculative execution of the subsequent code using a predicted value (`v`). The DL is scheduled to issue as early as possible, followed by the value prediction (`v`).

The computation proceeds using the predicted value (`work(v)`), with that computation being checkpointed along its execution path to support program rewind. When the computation is complete, we compare the predicted value with the actual value. If they are equal then we can commit the

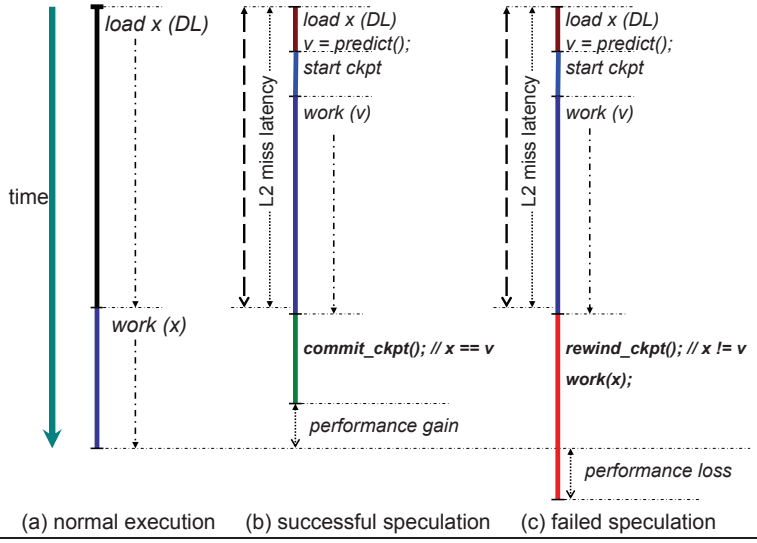


Figure 6.1: Overview: tolerating DL with speculative execution

checkpoint (as shown in Figure 6.1(b)). Ideally such a successful effort of prediction and speculation will result in a performance gain relative to the non-speculative original code. Should the value be mispredicted, as illustrated in Figure 6.1(c), then we must rewind the checkpoint and re-perform the computation with the correct result value of the DL (`work(x)`). The combined overheads of checkpointing as well as rewinding and retrying the computation can result in a performance loss relative to the original code.

In this chapter, we introduce two key compiler transformations that leverage compiler-based fine-grain checkpointing to tolerate DLs, namely data speculation and control speculation. For single-threaded speculation, we must make a prediction on the potential value of a DL and execute code that uses that prediction to make forward progress rather than pause the execution and wait for the DL's value to return from off-chip. This approach exploits the parallelism provided by a wide-issue superscalar processor that can execute instructions with memory access in parallel without the need of an explicit or implicit parallel thread or process. Ideally the latency of the DL is hidden when the prediction is correct, but execution can rewind and re-execute using the correct DL value should the prediction be incorrect.

6.2 Overlapping Execution with Delinquent Loads

A complete software-only speculative system is composed of the following three components: (i) a checkpointing system that backs up program changes along its execution path and helps to recover

from failed speculation, (ii) speculative compiler transformations that aggressively rearrange program layout that will normally be considered unsafe in order to enable speculative execution, and (iii) a value prediction system that generates a predicted value and allows the speculative system to optimistically make forward progress based on this prediction. We will have detailed discussions for each individual component and its particular nature in the context of DL speculation later in this chapter. In order to enable speculative execution overlapping with DLs, we need to identify the precise location of each individual DL that contributes for the speculative system.

6.2.1 DL Identification

We identify DLs by profiling second-level (L2) cache misses using a PIN [42]-based cache simulator that we developed for this work. The PIN framework provides an infrastructure to allow injecting 3rd-party user code at arbitrary program locations to conduct custom program analysis or transformations at runtime. The custom program analysis codes are thus PIN's plugins (also called pintools) that can be loaded dynamically into memory on demand. Our pintool-based cache simulation is a detailed functional cache simulator that properly mimics two levels of configurable cache hierarchies with separated L1 data cache and L1 instruction cache, as well as a shared L2 cache. The PIN framework recognizes each memory-access instruction from the application and redirects them to a software cache model established in the cache simulator pintool. Within each memory access, the software cache model captures necessary access signatures (read vs. write, effective memory address, length of data, etc.) and performs functional cache simulation. All cache transactions are recorded: both cache-miss and cache-hit events will update statistical counters within the cache model. A cache-miss event triggers cache behavior that bring in the missing data from higher levels. The cache simulator enforces inclusiveness—data in a lower cache level is guaranteed to be included in a higher cache level. The simulator also brings in the missing data by evicting a cache line based on a configurable cache-replacement policy when no cacheline within the corresponding cache block is available. Note that due to the nature of the PIN framework, our cache simulator currently can simulate the cache behaviors of only one application at a time. The software cache model is easily configurable when dealing with various cache architectures, including total levels of cache, cache size, cache-line size, degree of associativity, replacement policy, etc.

One compelling feature of the PIN infrastructure is that, when a benchmark is compiled with debug information enabled, it allows us to directly associate load and store instructions with their corresponding source code location. Hence the simulator can reliably map each load instruction that

Index	Line-size	Assoc.
0	32B	2
1	32B	4
2	32B	8
3	32B	16
4	64B	2
5	64B	4
6	64B	8
7	64B	16
8	128B	2
9	128B	4
10	128B	8
11	128B	16

Table 6.1: L2 cache configuration space explored (across a range of L2 cache sizes)

is responsible for a large fraction of L2 cache misses back to the offending source code location. This is a critical feature that can help us identify the DL locations on the source-program level.

For the rest of this chapter, we will consider a particular load instruction to be a delinquent load if it is responsible for greater than 5% of all L2 cache misses in a program. We will also refer to the actual percentage of L2 cache misses as the *significance* of that delinquent load (i.e., a load that is responsible for all of a program's L2 cache misses would have a significance of 100%).

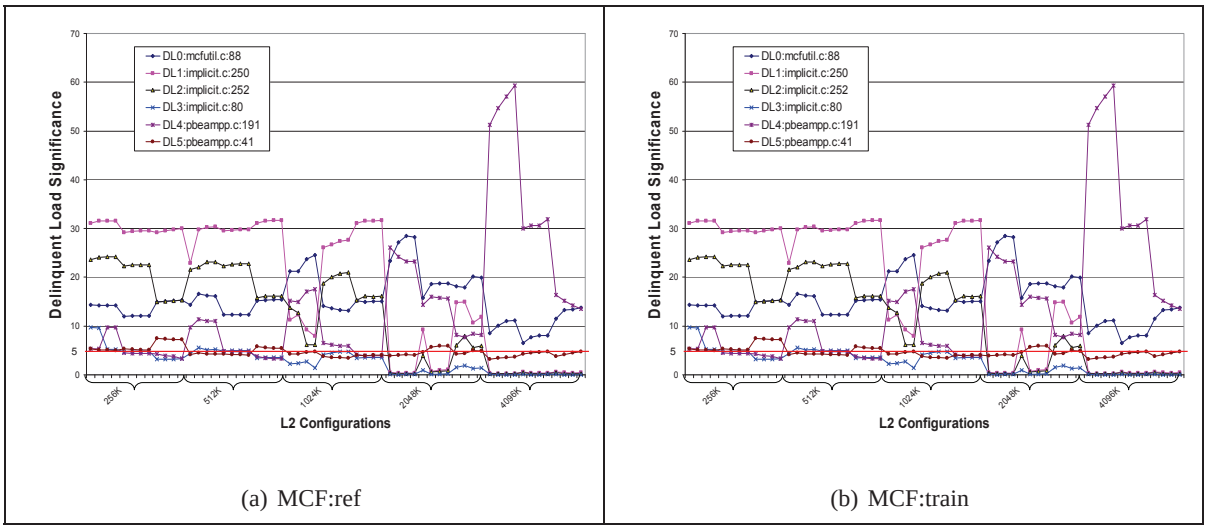
6.2.2 DL Persistence

We measure a wide range of L2 cache architectures, with sizes varying from 256KB to 4MB, cache-line size varying from 32B to 128B, and associativity varying from 2 ways to 16 ways. Table 6.1 summarizes the cache configurations that we have studied for each available cache size. This L2-cache exploration space covers a large number of cache configurations for existing CPUs that are commercially available. The index on the first column indicates the relative order among the possible cache configurations, and is the implicit order on x-axis data for the figures to appear in this section.

We use SPEC2000INT [17] benchmarks, compiled with compilers of various vendors (*gcc* and *icc*), versions (*gcc* 3.4.4, 4.0.4, 4.1.2, 4.2.4, 4.3.2, and *icc* 9.1), and optimization levels (*O*0, *O*2 and *O*3)

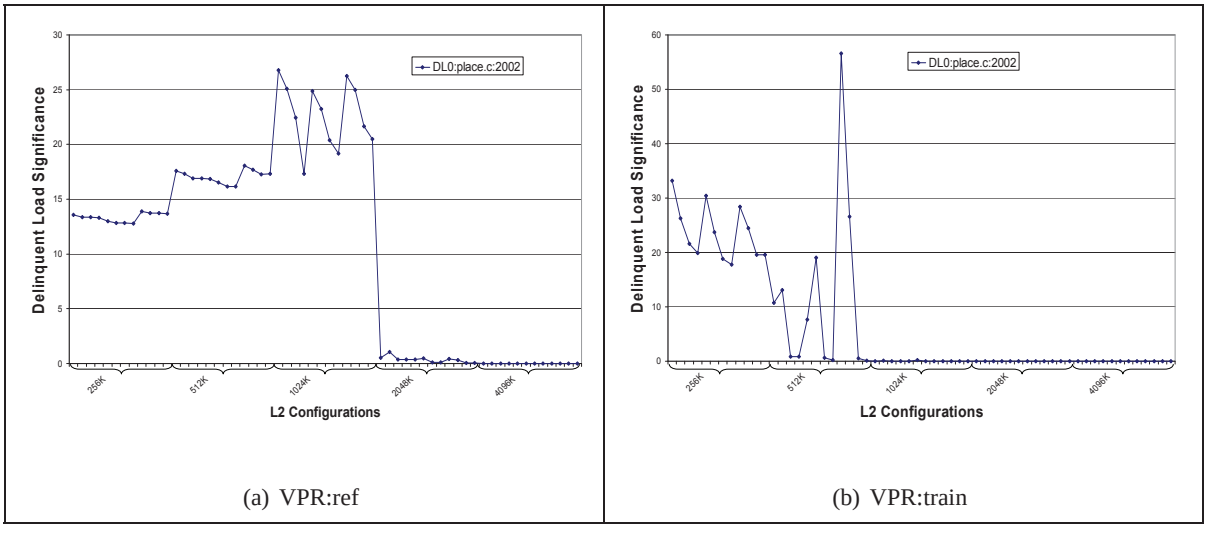
App Name	Input Data	% L2 misses	DL's Source Position
mcf	inp.in	DL0: 14.4%	mcfutil.c:88
	(ref input)	DL1: 31.1%	implicit.c:250
		DL2: 23.7%	implicit.c:252
		DL3: 9.7%	implicit.c:80
		DL4: 5.4%	pbeampp.c:191
		DL5: 5.3%	pbeampp.c:41
		total: 89.6%	
bzip2	input.program	DL0: 16.8%	bzip2.c:1260
	(ref input)	DL1: 12.2%	bzip2.c:2688
		DL2: 18.3%	bzip2.c:2688
		DL3: 14.9%	bzip2.c:2282
		total: 62.2%	
vortex	(ref input)	DL0: 15.7%	bmtobj.c:831
		DL1: 12.6%	mem10.c:752
		DL2: 11.5%	mem10.c:596
		total: 39.8%	
parser	ref.in	DL0: 10.4%	parser.c:194
	(ref input)	DL1: 18.6%	xalloc.c:122
		total: 29.0%	
vpr	(ref input)	DL0: 13.6%	place.c:2002
		total: 13.6%	

Table 6.2: Properties of significant DLs in the SPEC2000INT benchmark suite



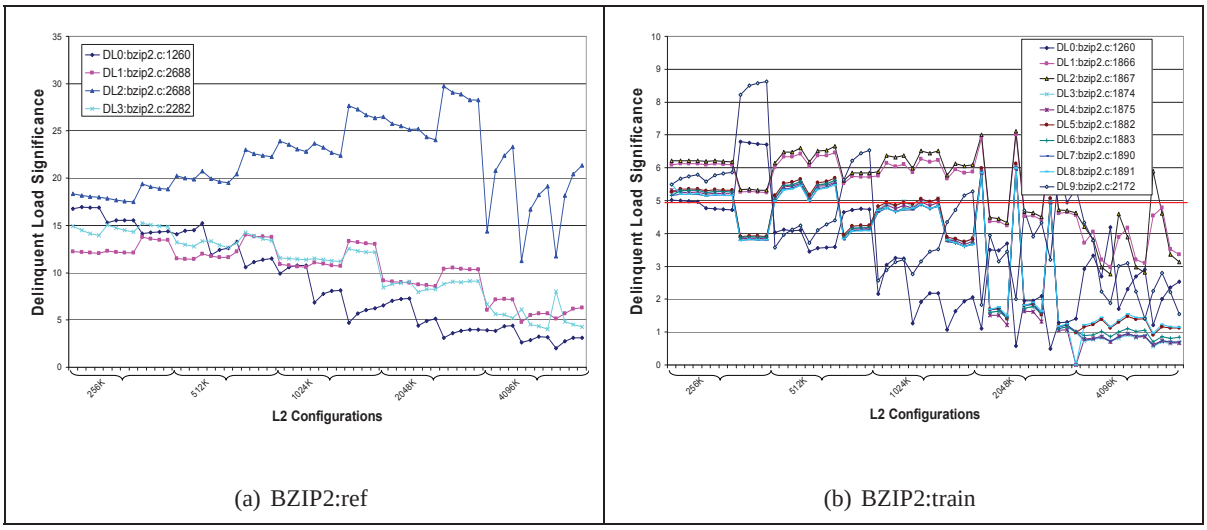
(a) MCF:ref (b) MCF:train

Figure 6.2: Persistence of DLs across architectures and benchmark inputs: MCF



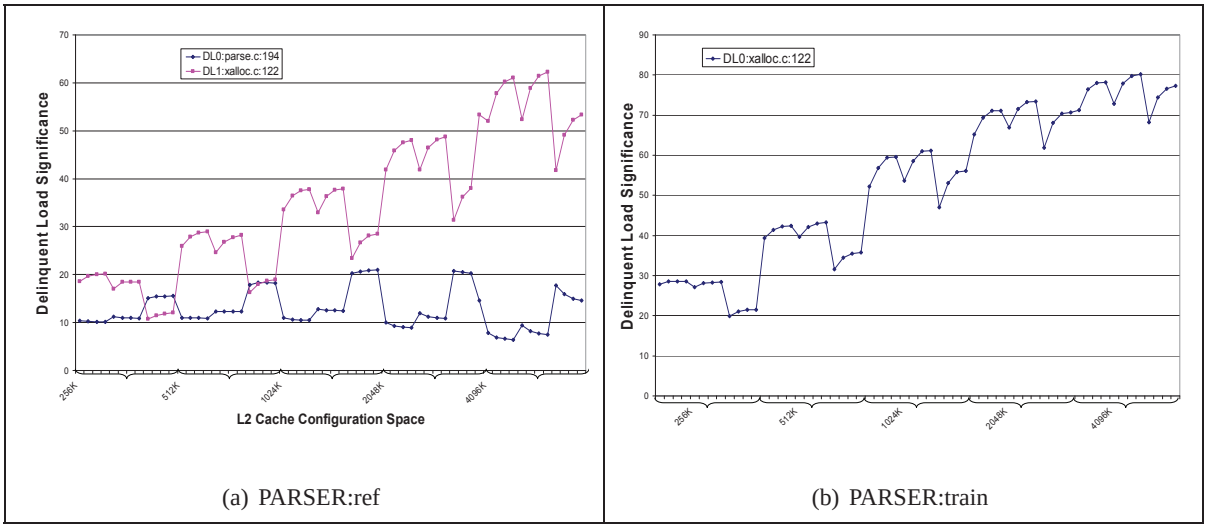
(a) VPR:ref (b) VPR:train

Figure 6.3: Persistence of DLs across architectures and benchmark inputs: VPR



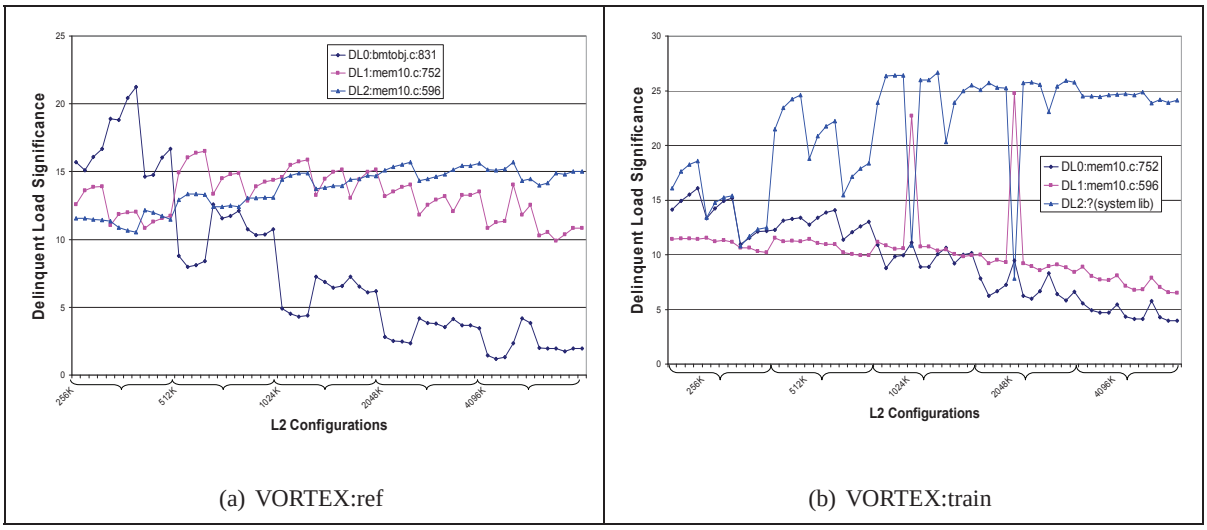
(a) BZIP2:ref (b) BZIP2:train

Figure 6.4: Persistence of DLs across architectures and benchmark inputs: BZIP2



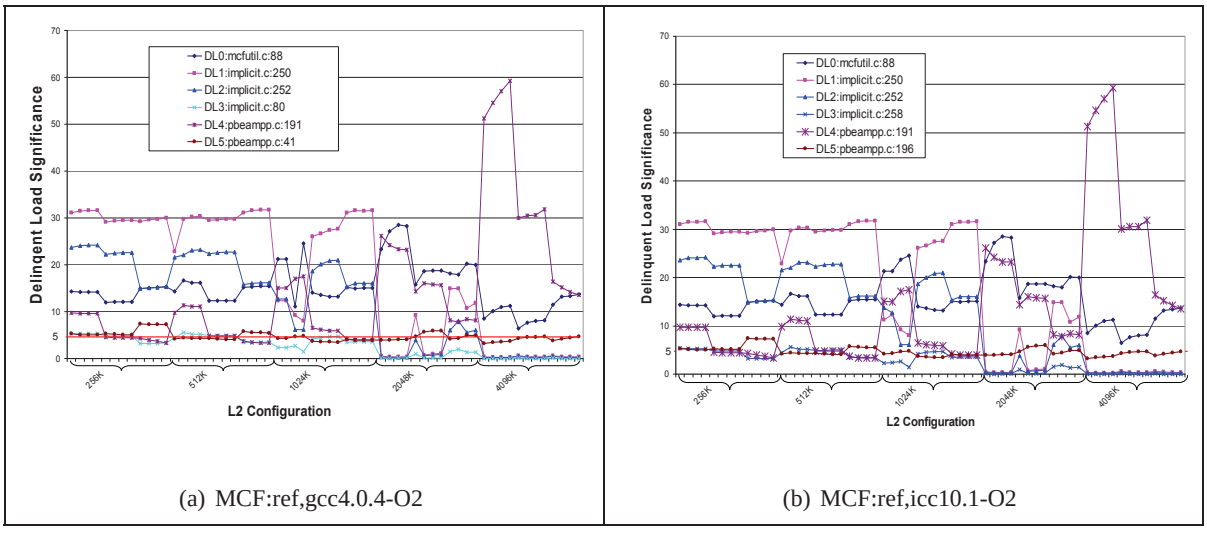
(a) PARSER:ref (b) PARSER:train

Figure 6.5: Persistence of DLs across architectures and benchmark inputs: PARSER



(a) VORTEX:ref (b) VORTEX:train

Figure 6.6: Persistence of DLs across architectures and benchmark inputs: VORTEX



(a) MCF:ref,gcc4.0.4-O2 (b) MCF:ref,icc10.1-O2

Figure 6.7: Persistence of DLs across compilers: MCF

to study DL locations and properties. We configure the PIN-based cache simulator with 2-level cache (separated data and instruction L1 cache and unified L2 cache) that covers large variations of cache size, cache line size and degree of associativity. We give the configuration details of the cache simulator in Table 6.1.

We conduct our initial investigation on all SPEC2000INT `C` benchmarks using both *training* and *reference* inputs. We find that only a subset of the applications contain DLs with significance of five percent or higher. We call a particular DL *persistent* if its offending source-code location remains unchanged when other conditions change. To consider optimizing DLs in a compiler, the DLs need to be persistent—they cannot be sensitive and shift positions due to a particular configuration of the L2 cache. In this section we measure the *persistence* of L2 load misses (DLs) in our benchmark applications across a broad range of L2 cache architectures. We also measure persistence across program inputs, as well as compiler vendors, versions and optimization levels.

In Table 6.2 we list existing SPEC2000INT applications with significant DLs, as well as individual DL with its associated significance inside each benchmark, all under the *reference* input. In this setup, we use a 256KB L2 cache configured with 32B cache lines and 2-way set associativity. As shown in the table, DLs from a small number of source-code locations are responsible for a very large fraction of all L2 cache misses in these applications, ranging from 13.6% (VPR) to 89.6% (MCF).

We present detailed results of our DL's persistence analysis in a sequence of figures: from Figure 6.2 for `MCF` to Figure 6.6 for `VORTEX`. We use Figure 6.2 as an example for explanation. Figure 6.2 has two sub figures: Figure 6.2(a) shows the MCF DL's significance for reference input and Figure 6.2(b) shows the MCF DL's significance for training input. When we zoom into Figure 6.2(a), it presents six `DL`'s significance curves, representing the six identified `MCF DLs` we provide earlier in Table 6.2. The X axis of Figure 6.2(a) is the configuration space of L2 cache and it has been partitioned into five sections, representing 256KB, 512KB, 1024KB, 2048KB and 4096KB of L2 cache size, respectively. Within each partitioned section, there are a total of 12 different L2 cache configurations. We provide individual configuration details in Table 6.1 (index 0-11). The Y axis of Figure 6.2(a) is the DL's significance value in percentage.

Figure 6.2(a) shows a general trend that a DL's significance gradually reduces when the size of L2 cache increases. With a larger cache, more loads will hit the cache rather than issue a long-latency main-memory fetch. Notice that most DL's significance remain above the range of five percent—our definition of a `significant` DL. For those SPEC2000INT applications with `DLs`, the nature of significance is persistent across different L2 cache size and configuration. We verify each SPEC2000INT benchmark

application with significant DLs, present their DL's significance graphs from Figure 6.2 until Figure 6.6, and testify that our claim actually holds.

We conduct one additional verification on DL's persistent natures over compilers with different vendor, version and optimization levels. We present one result in Figure 6.7 that compares MCF's DLs using gcc-4.0.4 and icc-10.1, both at optimization level O2. These two compilers were at the state-of-art state when we performed the measurement. Figure 6.7 shows that MCF DL maintains its persistence across different compilers.

According to our analysis and discovery using SPEC2000INT CPU benchmark suite, we find that not all applications in this benchmark suite have DLs with significance of five percent of higher. Within the subset of applications that contain significant DLs, the DLs have the following persistent properties:

- the DLs are persistent across various L2 cache configurations (size, line size, ways of associativity), as long as the working set doesn't entirely fit into the L2 cache;
- the DLs are persistent across different compilers, including vendors, versions and optimization levels;
- the DLs are persistent across inputs (training or reference).

Through our analysis [67, 70], we find that DLs are more likely to appear in unexpected locations that compilers cannot normally predict. E.g., for integer-intensive applications in SPEC2000INT, they often appear at pointer-dereference locations to a structure whose size is bigger than the current cache-line size, or in multiple levels of pointer deference sites. Modern CPU's cache architectures do excellent work in prefetching predictable access patterns, thus normal memory accesses whose access distances fall within the size of cache line will have good cache behavior. However, for those that do not follow this pattern, DLs will be more likely to appear. As long as the current program's working set does not entirely fit into the last-level cache, DLs appear persistently [1] even with different input data sets. We thus call this property *DL persistence*. As a result for DL's difficult-to-predict nature, existing compilers cannot statically identify DLs' locations at compile time or perform effective transformations to hide the long DL latency. Thus DLs keep *persistent* across different compiler's vendors, versions and even optimization levels.

[1] Programs with persistent DLs may slightly shift the locations where DLs appear because the associated significance may change when given different inputs. However, a DL's persistence remains provided the program's working set do not entirely fit into cache.

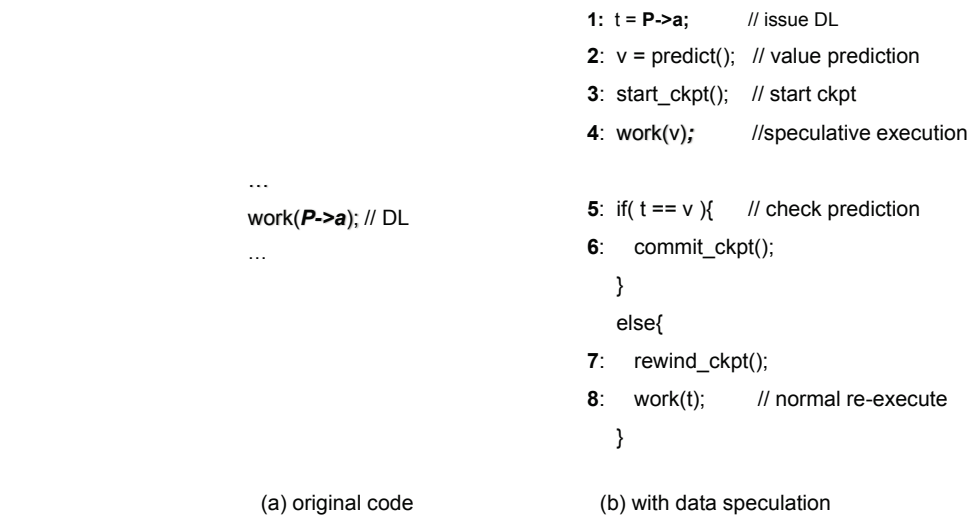


Figure 6.8: Data speculation

Given the DLs are persistent, we can further leverage compiler transformations to enable potential execution overlapping with the DL.

6.2.3 Data Speculation

The first method of tolerating DL latency is *data speculation* (*DS*) where we make a prediction on the result value of the DL and use it to continue execution speculatively, as illustrated in Figure 6.8. After issuing the DL as early as possible (*1*), predicting the DL's data value (*2*), starting the checkpoint (*3*), and performing speculative execution based on that predicted value (*4*), we then attempt to commit the speculation. The commit process first checks whether the prediction was correct by comparing the predicted value with the DL's value(*5*) : if so then the checkpoint is committed (*6*), otherwise the checkpoint is rewound (*7*) and the computation is re-executed non-speculatively using the correct DL result value (*8*).

6.2.4 Control Speculation

Whenever the result value of a DL is used *solely* within a conditional control statement (E.g., a branch), as shown in Figure 6.9(a), we have an interesting opportunity: rather than predicting the exact result value of the DL we can instead merely predict the boolean result of the conditional—condition taken or not taken, which ideally will more easily be an accurate prediction than predicting the exact result value. We call this form of speculation *control speculation (CS)*, which is essentially a special case of

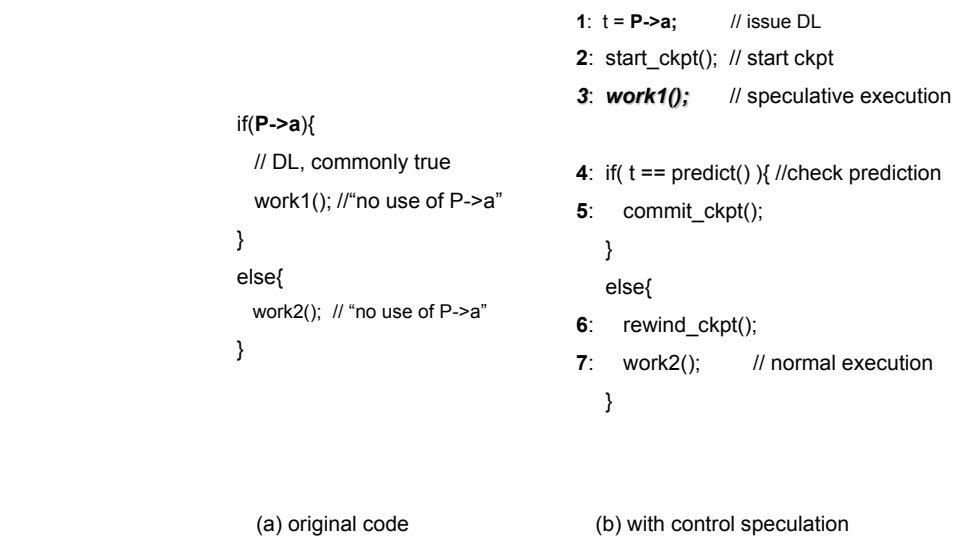


Figure 6.9: Control speculation

data speculation.

We present the control speculation's compiler transformations in Figure 6.9(b). When a DL resides on a control-flow branch, we have an opportunity to hoist the work carried within the frequently-used branch to be earlier than the DL, as suggested in Figure 6.9(b) for `work1`. Similar to the data-speculation case, we issue the delinquent load as early as possible (1), immediately followed with start checkpointing (2), and the speculatively hoisted work item (3). Note that in (1) we introduce a new temporary variable (`t`) to hold the return value of the delinquent load. At this point, we compare the predicted value with the DL's return value (4). If they are the same, the predictor makes a good prediction and we are ready to commit the speculation (*5*). Otherwise, if the predicted value is not the same as the DL's return value, the prediction has failed in this situation. The scheme will need to abort the current checkpoint (*6*) that effectively undoes `work1`, and execute the less-frequently executed branch (`work2`) in non-speculative mode (*7*).

Modern processors perform branch prediction and speculatively execute instructions beyond the branch—however this speculation is limited to the size and aggressiveness of the processor's issue window that can contain a small and limited number of instructions. The available independent instructions within this limited window are further constrained. With compiler-assisted control speculation, we can ideally speculate more deeply, allowing more instructions and greater opportunity for tolerating all of the latency of a DL.

6.2.5 Value Prediction

Both data speculation and control speculation need high-accuracy value predictions [13, 40, 55, 58, 60]. Since the value prediction's accuracy directly correlates to checkpointing's commit ratio, highly accurate value prediction is a must to guarantee high checkpoint commit rate—a vital step necessary for performance gains through speculative execution. However, the computations involved in generating the predicted value are on the critical path and thus must be treated as part of the speculation overhead. As a result, the competing goals of value prediction in the context of checkpointing-enabled applications are two-fold: (i) to maintain very high prediction accuracy, and simultaneously (ii) to minimize involved computational and storage overheads.

We study a wide range of value prediction methods, from simple constant-value prediction, last-known value prediction, constant stride value prediction, to more complex and storage-intensive table-based context-aware value predictions [40, 58], as well as adding confidence as a method to throttle the prediction result and increase prediction accuracy. To our surprise, we find that simple last-known value prediction and constant-stride value predictions work fairly well and satisfy most of our needs in value prediction for speculation. The more expensive, complex and computationally-intensive value predictors (e.g., table-driven context-aware predictors) do not necessarily deliver higher prediction accuracy, but at the expense of much more computation and storage overhead. As a result, we utilize only constant-stride value predictor and last-known value predictor to service the speculation's prediction demand for the rest of this chapter.

6.3 Theoretical Performance Modeling

Figure 6.10 illustrates the ideal speculative timing model for overlapping execution with DLs. Figure 6.10(a) is the normal sequential model where the total execution time is the accumulation of both DL's latency cycles and the work's latency cycles, and the continuation of work relies on DL. This represents the conditions where the DL's value is immediately needed to allow execution to proceed with inside work, thus the program stalls until the DL's value returns from the high-latency memory system.

Under the speculatively overlapped model given in Figure 6.10(b), the program continues with the predicted value while the memory system is simultaneously serving the DL. This resembles a form of memory-level parallelism though there is no explicit parallel thread needed to fetch the DL's value from main memory. Thus under this model, the total execution time is the *maximum* of the two individual

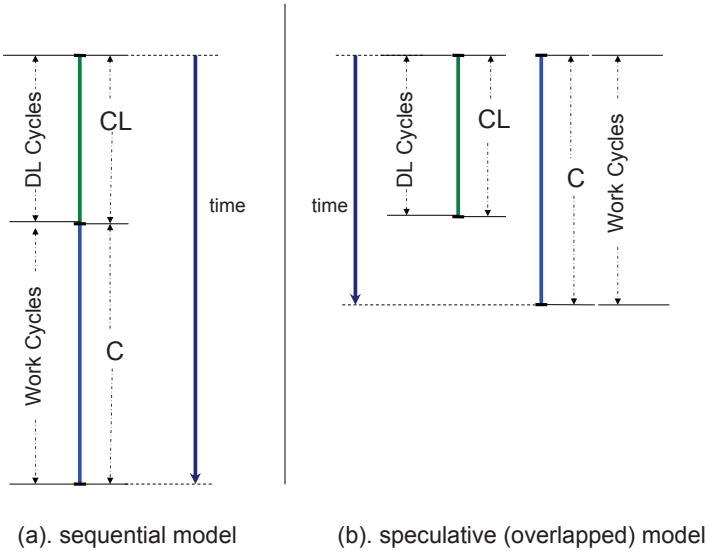


Figure 6.10: DL ideal timing model

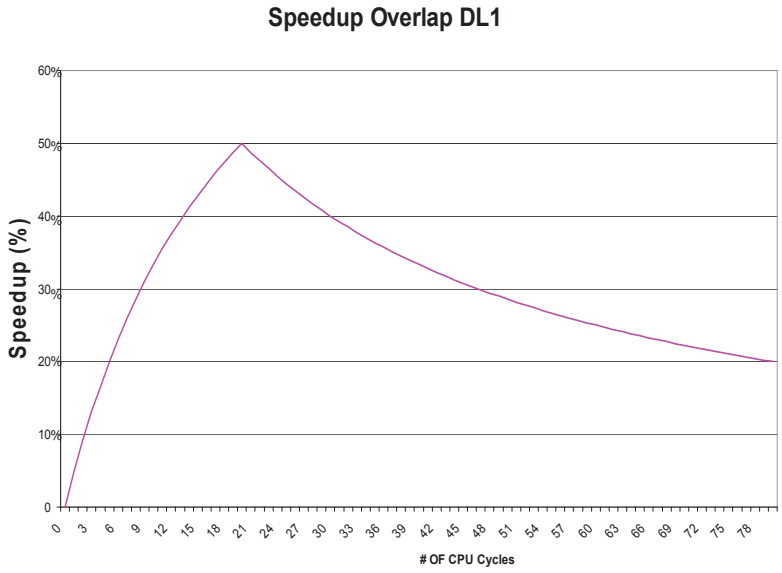


Figure 6.11: Overlap execution with L1 cache only (20 cycle L1 cache miss latency)

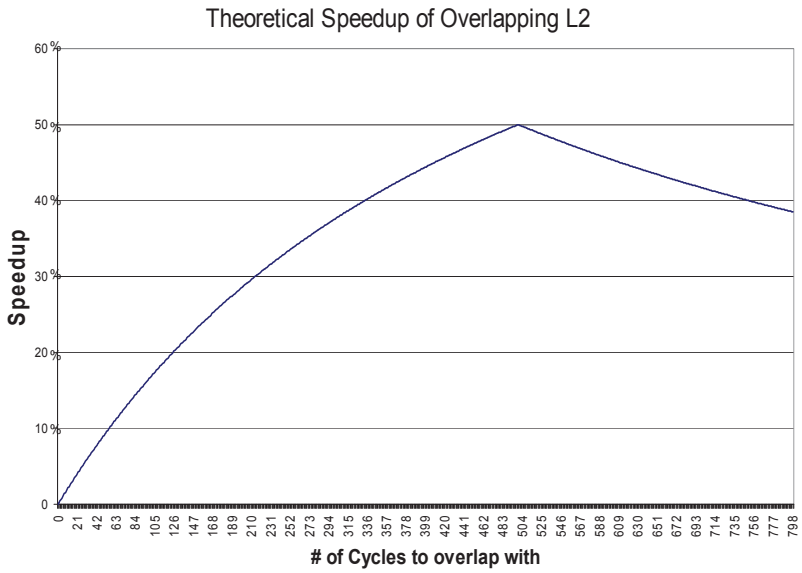


Figure 6.12: Overlap execution with L2 cache only (500 cycle L2 cache miss latency)

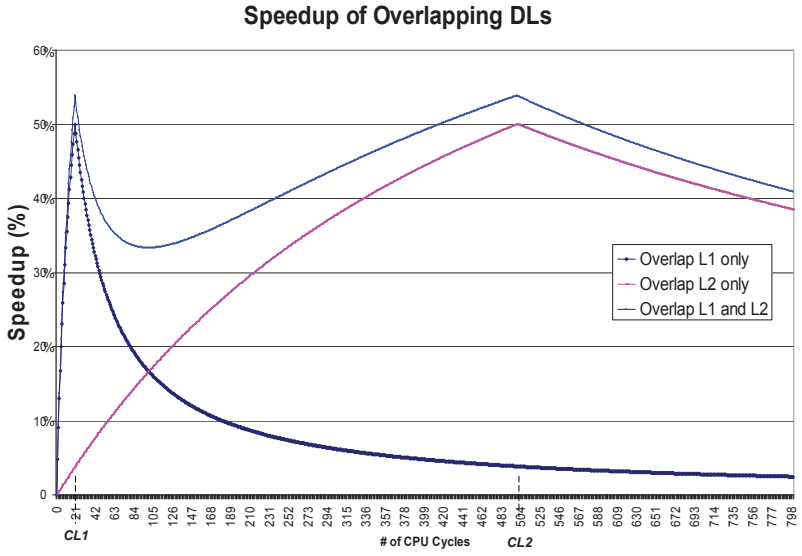


Figure 6.13: Overlap execution with both L1-and-L2 cache (all inclusive)

participating components. This models the cases when either the DL's value is not being immediately needed or the DL is being used to make a predictable control-flow decision and therefore its precise value is less important.

We now present a theoretical performance modeling of speculatively overlapping execution with up to two levels of cache.

Let *CL* denote the cycles of a cache miss (DL) and let *C* denote the cycles of work that overlaps with the DL, we have

$$T_{sequential} = CL + C$$

$$T_{speculate} = \max(CL, C)$$

Let *S* denote the relative speedup of overlapping execution with DL, we give the definition of *S* as

$$S = \frac{T_{sequential} - T_{speculate}}{T_{sequential}} = \frac{CL + C - \max(CL, C)}{CL + C} \tag{6.1}$$

Thus the ideal theoretical relative speedup for overlapping with only L1 cache is

$$\begin{aligned} S_1 &= \frac{CL_1 + C - \max(CL_1, C)}{CL_1 + C} \\ &= \begin{cases} \frac{C}{CL_1 + C}, & \text{if } C < CL_1 \\ \frac{CL_1}{CL_1 + C}, & \text{if } C \geq CL_1 \end{cases} \end{aligned}$$

Similarly the ideal theoretical relative speedup for overlapping with only L2 cache is

$$S_2 = \frac{CL_2 + C - \max(CL_2, C)}{CL_2 + C}$$
$$= \begin{cases} \frac{C}{CL_2+C}, & \text{if } C < CL_2 \\ \frac{CL_2}{CL_2+C}, & \text{if } C \geq CL_2 \end{cases}$$

In addition, we obtain the theoretical relative speedup for overlapping with combined L1 and L2 cache by aggregating individual speedups:

$$S = S_1 + S_2 = \begin{cases} \frac{C}{CL_1+C} + \frac{C}{CL_2+C}, & \text{if } 0 \leq C < CL_1 \\ \frac{CL_1}{CL_1+C} + \frac{C}{CL_2+C}, & \text{if } CL_1 \leq C < CL_2 \\ \frac{CL_1}{CL_1+C} + \frac{CL_2}{CL_2+C}, & \text{if } C \geq CL_2 \end{cases}$$

We present three theoretical relative speedup figures for the three possibilities of overlapping execution with DL: with a L1 cache miss only (Figure 6.11), with a L2 cache miss only (Figure 6.12), and with a combined L1-and-L2 cache miss (Figure 6.13), respectively. The figures show both overall similarity and individual differences. For ease of comparison, we fix L1 cache miss latency to 20 cycles ($CL_1 = 20$) and L2 cache miss latency to 500 cycles ($CL_2 = 500$). For each figure, on x-scale we give the number of work cycles that are suitable to speculatively overlap with the DL, and on y-scale we give the maximum theoretical relative speedup for the respective case of overlapping with a cache miss. Figure 6.13 is an all-inclusive figure that shows all three participating curves together. It presents the composition process that overlaps and interpolates Figure 6.11 and Figure 6.12, and produces Figure 6.13.

In Figure 6.11, the relative speedup that overlaps with L1-only workload goes sharply to its peak from zero to CL_1 (20) cycles in the beginning. Since the L1-miss-and-L2-hit cycles are relatively short, the curve has only limited room to stretch before reaching its theoretical peak, which is predicted to be 50% when the overlapped cycles (C) is equal to L1-miss-and-L2-hit cycles (CL_1). In Figure 6.12, the relative speedup that overlaps with L2-only work can be treated as horizontally scaling the curve in Figure 6.11 to match with L2-miss-and-memory-hit cycles (CL_2) and its theoretical performance upper-bound is also 50%. Given ideal workloads, the two theoretical speedups can further combine and

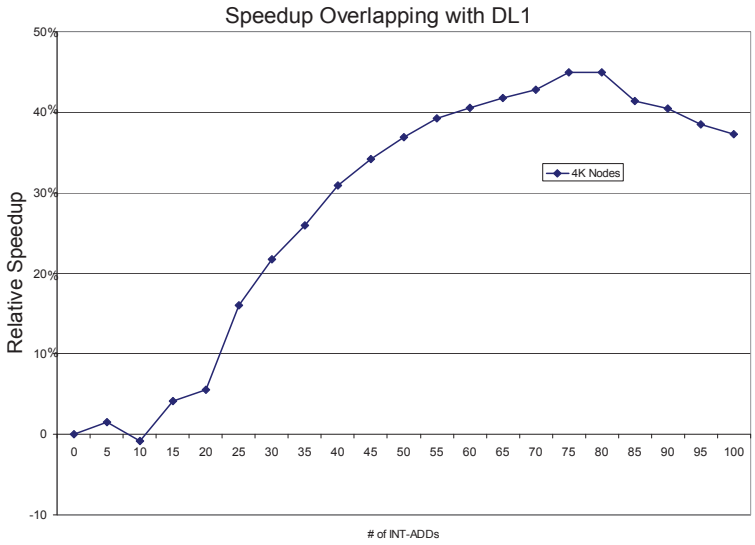


Figure 6.14: Real DL Speedup: L1

generate an aggregated effect that can cross the 50% threshold, presented as the CL_2-centered triangle-like area in Figure 6.13. Since multiple levels of cache work together in a real machine, one cannot easily separate and observe the L1 cache-miss only or L2 cache-miss only effects, the combined L1-and-L2 effect (presented in Figure 6.13) is the one that we can expect from fine-grain speculation in real-world workloads.

6.4 Micro Benchmark and Practical Performance

Software-only speculation that overlaps fine-grain execution with DLs is a field that doesn't have established or well-known workloads. We are thus in need of benchmark or micro-benchmark applications that have significant DLs as well as sufficient work items that are suitable for the fine-grain overlapping. This section is devoted to the efforts of building such applications.

6.4.1 Micro Benchmarks

We develop a set of synthetic benchmarks for real-machine evaluation. This includes a linked list (LINKLIST), a binary search tree, a B-tree, a red-black tree, an AVL tree, and a hashtable. They behave similarly in that accesses to dynamically allocated data structures result in frequent last-level cache misses (DLs). We use LINKLIST as the representative workload for this study. We make each node in the LINKLIST larger than the cache-line size on the machine we used to conduct the evaluation. Within

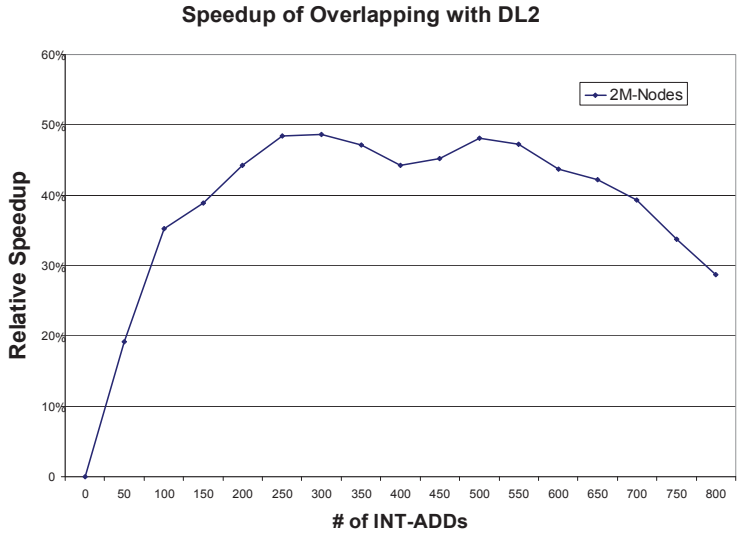


Figure 6.15: Real DL Speedup: L1 and L2

the nodes, we also make frequent accesses to fields whose offset distances to the begin of the node are larger than the cacheline size. As a result, most DLs will appear when traversing the LINKLIST—a step that simplifies the need to precisely identify DL locations. For the workload that can be used to overlap with DLs, we use a simple integer accumulator through a loop (*INT-ADDs*). By varying the loop's trip count, we can easily control the granularity of the workload to make it ideally suitable for tolerating L1-miss only latency, L1-hit-and-L2-miss latency, or L1-and-L2-miss latency. We make this workload independent of the DL's return value so the speculative overlapping has the potential to scale up to the limit of the respective cache-miss cycles. We make the loop's trip count an input-dependent parameter to destroy potential compiler optimizations applicable on this critical workload-control loop. To exacerbate the situation, we randomize the starting address of each node, which helps to undermine the hardware prefetcher. By adjusting the number of nodes in the LINKLIST, we achieve the effect of either polluting only the L1 cache (L1-DL), or polluting both L1-and-L2 caches (L1L2-DL) through a single linklist traversal. The empirical list size we use is 4K nodes for L1-DL and 2M nodes for L1L2-DL, respectively. We use *RDTSC* [1, 63] for fine-grain time measurement.

The machine used for evaluating the micro benchmarks has a single-core 3.0GHz Pentium-IV CPU, with a 16KB 4-way set-associative L1 data cache, a 12KB 8-way set-associative L1 instruction cache, and a 512KB 8-way set-associative shared L2 cache. The cache-line size is consistent at 64B across all levels of cache. Each measurement data point is the arithmetic average of at least five independent runs.

6.4.2 Performance of Micro Benchmark

Figure 6.14 shows the relative speedup of overlapping L1 DL using LINKLIST with 4K nodes. The workload to overlap with DL is a loop performing accumulation of integer adds (INTADDs, the loop's trip count is shown on the x-axis), while the y-axis gives the relative speedup. Figure 6.14 is very similar to the theoretical prediction of L1 speedup curve given in Figure 6.11. It reaches its maximum at around 45% relative speedup while overlapping roughly 70 INTADDs.

When performing testing on real machines, a workload that pollutes the L2 cache must already have the L1 cache polluted. It is difficult to obtain the performance figure with a workload that overlaps with only the L2 cache (L2 DL). We thus focus on workloads that overlaps with L1-and-L2 (L1-L2) DL.

Figure 6.15 shows the relative speedup result when overlapping with L1-L2 DLs using 2M nodes. The workload to overlap with DL is the same integer-accumulation loop as the one used in Figure 6.14. The difference is on the loop's trip count, which simulates the granularity of the workload that overlaps with the DL. Figure 6.15 roughly contains two stages. In stage one, the curve reaches around 35% speedup at roughly 70 INTADDs. This agrees with our own measurement given in Figure 6.14 and it is the effect of mostly overlapping L1 DL. In stage two, the speedup curve maintains its stability over 35% until roughly 750 cycles, with a maximum reaching very close to the 50% theoretical peak. This closely matches the L1-and-L2 prediction given in Figure 6.13 where a wide range of 35%+ relative performance is expected after stage one.

6.5 Challenge with Real-World Applications

We give theoretical analysis and predictions on speculative performance for overlapping execution with various levels of cache. We verify this claim with micro benchmarks that can reach very close to the theoretical peak and largely represent the performance trend that the theoretical model predicts. These results are obtained under ideal conditions that (i) there is no failed speculation because the involved predictor can yield 100% prediction accuracy, (ii) there are no cache misses within the simulated workload that is used to overlap with the original *DL*, (iii) the checkpointing framework generates minimal overhead because the speculative workload has no dependency on the predicted value of the *DL*, and (iv) the speculative workload is only fine-grain enough to sufficiently overlap with the targeting *DL*. However, such ideal situations may not always hold under non-synthetic benchmarks on real machines.

We further investigate the possibility and feasibility of applying control speculation and data speculation transformations we introduced earlier in this chapter to real-world applications (e.g., the

DL-intensive applications in the SPEC2000INT suite). We expect some major challenges. First, even with all checkpointing optimizations enabled, checkpointing overhead is non trivial and cannot be ignored. Second, the success rate from branch prediction or value prediction plays an important role because failed predictions will directly translate into failed speculation and trigger the expensive checkpointing abort and recovery process. Thus a relatively low speculation success rate can render the entire speculative scheme uninteresting. Third, the compiler needs to find work that is coarse-grain enough and can potentially overlap with the entire latency of the identified *DL*. This ideally suitable workload may not exist in real-world applications. Finally, the compiler needs to recognize an ideal sweet spot to terminate speculative execution and maximize potential speculation benefit. This is no clear indicator on the exact location for such sweet spots.

6.6 In-depth Study Using MCF

`MCF` is a benchmark application from the SPEC2000INT suite. `MCF` frequently operates over linklist-like type data structures and is known to have intensive DLs. We thus select `MCF` to conduct an in-depth case study that aims to explore manually enabled speculative execution overlapping with the significant MCF DLs.

6.6.1 Insights of Significant MCF DLs

Due to its pointer-intensive nature, MCF is known to have multiple static significant DLs across its entire code base. Using the same pintool-based cache simulator, we present the top six most-significant MCF DLs in Figure 6.16 and provide insights and analysis of their characteristics.

First, most DLs reside within a pointer access that attempts to fetch a field within a node structure. Examining source code indicates that most DLs belong to code that is part of a linklist traversal. All linklist nodes are dynamically allocated with node size bigger than the size of a cache line on the CPU architecture that MCF runs. This indicates little inter-node spatial locality and implies that conventional prefetching techniques will not likely be effective for these DLs. Second, the majority of DLs are within one level of pointer access (DL0 to DL4), with the only exception on DL5 whose behavior is through multiple levels of pointer indirections. This matches the style of single-level linklist where most actions happen within the single node that is currently being accessed. Third, DLs are more likely to happen within a frequently-accessed field of a big node structure whose size is larger than the cache-line size. Knowing the size of *arc* is 32B (DL1 to DL5) and the size of *node* in MCF is 60B (DL0), they are

```
while( node != root ){
    while( node ){
        if( node->orientation == UP )          // DL0
            node->potential = node->basic_arc->cost + node->pred->potential;
        else{
            node->potential = (node->pred)->potential -node->basic_arc->cost;
            checksum++;
        }
  …
}
```

(a) DL0: mcfutil.c:86

```
while( arcin ){
    tail = arcin->tail;                        // DL 1

    if( tail->time + arcin->org_cost > latest ){ // DL 2
        arcin = (arc_t *)tail->mark;
        continue;
    }
    …
}
```

(b). DL1: implicit.c:250, DL2: implicit.c:252

```
cost_t compute_red_cost( cost_t cost, node_t *tail, cost_t head_potential )
    cost_t cost; node_t *tail; cost_t head_potential;
{
    return (cost - tail->potential + head_potential);   // DL3
}
```

(c) DL3: mcfutil.c:80

```
for( ; arc < stop_arcs; arc += nr_group )
{
    if(arc->ident > BASIC ) { // DL4
        red_cost = bea_compute_red_cost( arc );
        …
    }
}
```

(d) DL4: pbeampp.c:191

```
cost_t bea_compute_red_cost( arc_t *arc ){
    return( arc->cost - arc->tail->potential + arc->head->potential); // DL5
}
```

(e) DL5: pbeampp.c:41

Figure 6.16: Significant DL locations in MCF

either equal-to or larger-than the cache-line size (32B) of the machine that we conduct analysis and evaluations. Loading a different linklist node is more likely to cause cache misses on such architectures because it is difficult for the cache sub system to predict and then prefetch the address of the *next* linklist node when the nodes are dynamically allocated. Finally, when there are multiple levels of pointer access (DL5), it is more likely to be a DL because such accesses are more unlikely to remain in cache.

Predictor	Accuracy	Speculative Performance
pred+ckpt+nousepred	100%	-0.14%
branch always taken	96.35%	-0.51%
const value(1)	94.66%	-1.33%
last value	91.3%	-1.36%
const value(0)	3.65%	-1.49%
const value(2)	1.69%	-2.43%
always predict wrong	0.0%	-2.45%

Table 6.3: Prediction accuracy and performance impact for MCF:DL4

6.6.2 Speculation over MCF DLs

We further investigate the potential for compiler-based data speculation and control speculation to tolerate DL latency, focusing on these significant DLs in MCF. Any speculation over DL that delivers performance benefits has to satisfy a critical condition: the DL's value must be highly predictable. Any low value prediction accuracy effectively renders the speculation unattractive due to the overwhelmingly expensive recovery overhead from failed speculations. Of the three DLs best-suited for data speculation in MCF (DL1, DL3, and DL5), their data values are too sparse and random. As an unfortunate result, all three have prediction accuracies that are too low to justify further exploration. For the three control speculation cases (DL0, DL2, and DL4), DL0 and DL2 are also too unpredictable; however DL4 presents an interesting case. Although it appears to be a control speculation, this DL has only three different integer data values ($0 : 3.65\%, 1 : 94.66\%, 2 : 1.69\%$) across the entire MCF execution. In Table 6.3, we present all branch predictions and value predictions we have explored on DL4, as well as their respective prediction accuracy. It is easy to see that a static branch predictor that always predicts taken (true) yields the highest DL4's prediction accuracy (96.35%) in reality.

The 100% accuracy (pred+ckpt+nousepred) represents an ideal case where prediction and checkpointing are both enabled and aggressively optimized, but the predicted value is not actually used. Hence this case merely measures the speculation (checkpointing and value prediction) overhead without benefiting from any speculative overlapping. This case results in only a tiny slowdown of 0.14%, highlighting the combined efficiency of our checkpointing framework and the value prediction. Various predictors yield vastly different prediction accuracies, ranging from 96.35% for a static always-taken branch predictor (always predicts true), to 94.66% for a constant value predictor (always predicts

a constant value one), to 91.3% last-known value predictor (always predicts the last value, with an immediate value update for any incorrect prediction), to 3.65% for a constant value predictor (always predicts a constant value zero), and finally to 1.69% of a constant value predictor (always predicts a constant value two). All predictions that yield realistic accuracies are from real predictors (prediction accuracy range between 0% and 100% in Table 6.3) embedded in fully functional speculative execution environments with error checking and failure recovery enabled. The *speculative performance* column shows steady performance degradations with the ever decreasing prediction accuracy. This matches our expectation that low prediction accuracy triggers failed speculations that can potentially be overwhelmingly expensive.

A small overall slowdown of 2.43% (with 1.69% prediction accuracy) is derived from a much larger slowdown factor within the function where the speculation occurs. This is the worst case of a realistic low-accuracy predictor can cause in MCF. The row with 0.0% prediction accuracy is achieved by constantly predicting a wrong value (data value of -1), which is neither in the distribution of available values (Table 6.3) for value prediction, nor contributes to any success in branch prediction (Figure 6.16, DL4 case). Thus the global slowdown of 2.45% represents the worst-case performance lower bound that an always-failing speculation over DL can possibly cause.

A few critical conditions need to be satisfied simultaneously in order for a speculative scheme to gain potential performance. This includes *(1)* highly-biased branch prediction or extremely accurate value prediction toward selected speculative region; *(2)* overlapping code region that is coarse-grain enough to closely match the speculation (DL) latency and compensate checkpointing overhead; *(3)* no control-flow terminating instructions within the overlapping code region which can prematurely terminate speculative execution, *(4)* no reuse of DL's value within the speculative region, and *(5)* no additional hidden DLs in the speculative region that are covered by a leading DL. Among the three MCF control-speculation cases (DL0, DL2 and DL4), DL4 is the only one that yields prediction accuracy high enough and worthwhile to conduct further investigation. Unfortunately, DL4 has only a small amount of computation to potentially overlap with (Figure 6.16). This code region is too fine-grain to completely hide the long-latency the DL caused while tolerating the software checkpointing overhead. In addition, DL4 is a *leading* DL—a DL that covers additional memory loads whose code distances to DL4 are within the cache-line size. Compiler transformations to enable speculative execution on DL4 breaks its delinquent-load nature, but exposes additional DLs that are otherwise hidden and covered by the leading DL (DL4). Leverage over control speculation or data speculation, we manage to eliminate the leading DL (DL4) which comes at the expense of exposing additional DLs that used to be hidden

under shadow. As a result, control speculation on DL4 gives no positive performance despite its high prediction accuracy and our highly efficient software checkpointing framework.

6.7 Summary

In this chapter, we conduct a thorough investigation of our first checkpointing application—overlapping speculative execution with delinquent loads. We present theoretical performance analysis that models speculative execution overlapping with DLs. Based on this model, we predict that the relative theoretical speedups will be around 50% for overlapping with L1-and-L2 cache misses. We verify this theoretical prediction with synthetic benchmarks that can achieve very close to the predicted peak performance. The verification results are obtained using a macro benchmark on real machines under ideal speculative conditions. Motivated by the feedbacks from the micro benchmark, we further conduct an in-depth study of real-world software-only speculation using all possible significant DLs in MCF, including various predictors, speculative transformations, and efficient software-only checkpointing. We find that not all DLs are suitable candidates for speculation. Within those suitable candidates, the amount of computation that can overlap for speculative execution is likely too fine-grain to compensate for checkpointing and value prediction overhead.

Chapter 7

Checkpoint-Enabled Debugging and Backtracking

In chapter 3 and chapter 4, we introduce a large and comprehensive checkpointing transformation and optimization framework that enables fine-grain checkpointing and reduces its associated checkpointing overhead through aggressive compile-time analysis and optimizations. Our compiler-based fine-grain checkpointing infrastructure provides a low-cost and software-only platform that is necessary to support many applications. In chapter 6, we give a detailed analysis of our first checkpointing application—overlapping execution with delinquent loads. In this chapter, we introduce two more interesting applications that leverage our checkpointing support to both gain distinct functionality and benefit from much reduced overhead. These applications include checkpoint support for debugging, and checkpoint-enabled automatic software backtracking.

7.1 Checkpoint-enabled Debugger

7.1.1 Overview

A debugger is a software program that helps programmers to identify and resolve software bugs. A normal debugging session begins with a user placing breakpoint(s) at designated program location(s) before launching a program inside a debugger. When the program starts its execution and stops at each breakpoint location, the programmer can examine the application's logic and states, trying to identify the root cause of the bug that is under investigation, as well as attempting a fix. However, once execution passes a certain breakpoint, it is usually difficult to rewind execution back to a previous

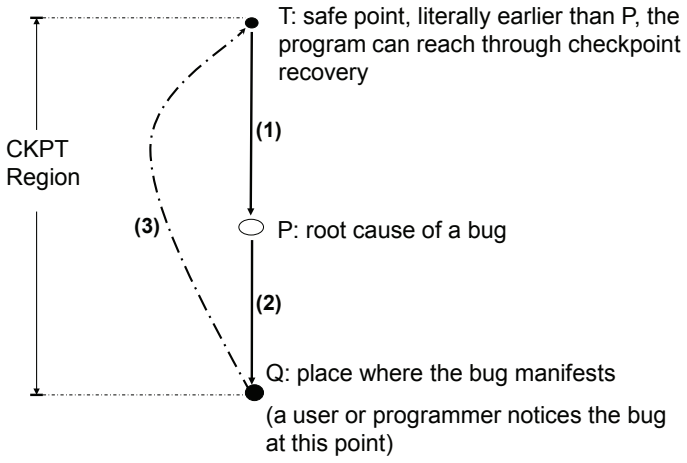


Figure 7.1: Overview: checkpointing support for debugging

program location, although a user may often find that the root cause of a bug is likely located close to the location of a previous breakpoint that execution has just passed by. Frequently restarting execution can be impractical in occasions where it may take an arbitrarily long time to again reach the suspicious bug location. Moreover, the bug may not be always reproducible under some certain execution environments.

Debuggers enhanced with our checkpointing support can help alleviate this situation. We expose the checkpointing APIs on the source-code level so that a programmer can selectively mark a checkpoint region that likely contains the root cause of the bug. The programmer first inserts an end-region marker slightly after the location where the bug manifests. This is an easy step because the bug's triggering location is well known to the programmer. This is usually the place where the programmer notices the application's abnormal behavior: generating a core dump, triggering an assert, issuing some error or warning messages to the console, printing some messages that are apparently wrong, etc. Properly identifying a start-region position that just includes the root cause of the bug requires some understanding of the code as well as an educated guess. The region needs to be big enough to contain the root cause of the bug, but at the same time cannot be overly large so that the programmer gets lost in an overwhelming amount of unrelated details. Even though an initial begin checkpoint-region marker placement may not always be ideal, we find through our own experience that it will quickly converge to the right position within very few iterations. In practice, we often place breakpoints overlapping with the checkpoint region boundaries. Once execution reaches the end of a checkpoint region, the matching

Index	Checkpoint Function	Explanation
1	start_ckpt()	begin a new checkpoint region
2	stop_ckpt(bool b)	finish current checkpoint region by either commit or abort checkpoint
3	commit_ckpt()	commit current checkpoint
4	abort_ckpt()	abort current checkpoint

Table 7.1: Checkpointing APIs exposed to debugger

breakpoint will stop the program's execution. At this moment, the programmer has the opportunity to decide the next action to take over the debugger's command interface: finish debugging this region by issuing a `stop_ckpt` command, or rewind execution to the beginning of the region by issuing an `abort_ckpt` command. This process iterates until the root cause of the bug is successfully uncovered.

Figure 7.1 shows the checkpointing-enhanced debugging process operating over a user-identified checkpointing region. When a normal program execution reaches program point `T` (the checkpoint begin position), checkpoint-enabled execution starts. Program point `T` needs to be a position earlier than the root cause of the bug. When program is executing at position `T`, we consider it is in a bug-free mode. Execution propagates along checkpoint-enabled path (1) covering the root cause of the bug `P`. Note that the root cause of the bug is triggered, but doesn't manifest immediately. Execution continues after the root-cause position along path (2), and finally reached the program position `Q` where the programmer notices the bug (bug manifestation point). The programmer can conduct any normal debugging activities along the execution paths after entering the checkpoint-enabled region. At program location `Q`, the programmer can selectively decide the next action. If the programmer decides to finish the current checkpoint and resume normal execution, a `commit_ckpt` command is issued. Alternatively, if the programmer plans to rewind execution to the begin of the checkpoint region (along path 3) and re-examine code in this section with more debugging activities, an `abort_ckpt` command is issued.

We present the set of relevant checkpointing APIs in Table 7.1. We provide these APIs in the form of `C` programing language source code prototype, thus a user can invoke a particular checkpointing service by calling its function's name on the debuger's command-line interface. Function `start_ckpt` starts a new checkpointing region by resetting all internal buffers and be ready to conduct backup actions. Function `stop_ckpt(bool)` finishes the current checkpoint region, with a boolean type argument indicating the proper action. When the boolean argument is *true*, the system will commit the current checkpoint; otherwise when the boolean argument is *false*, the system will abort the current checkpoint and rewind execution to the begin of the checkpoint region. The programmer is responsible for making the proper decision on whether to commit or abort the current checkpoint. We provide two additional

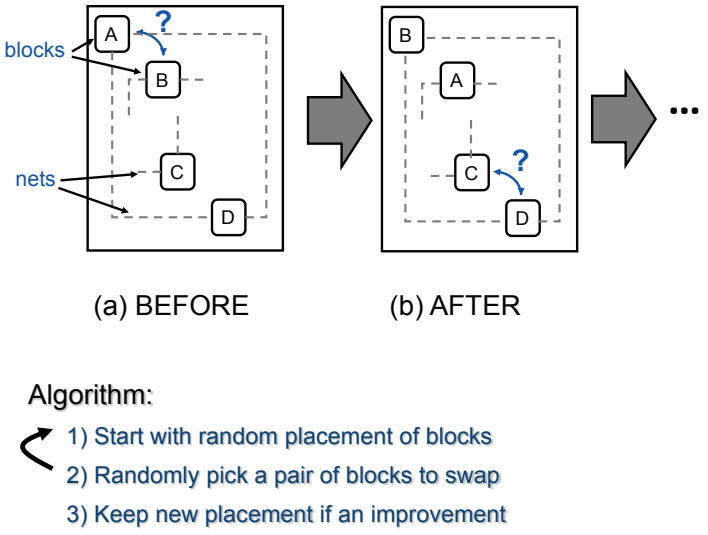


Figure 7.2: Overview: checkpoint-enabled software backtracking using VPR

API functions to further refine and simplify the `stop_ckpt` process: `commit_ckpt` commits the current checkpoint, and `abort_ckpt` aborts the current checkpoint, respectively.

7.1.2 Benefit

Debuggers enhanced with our checkpointing support gain the ability to rewind execution to a previously specified program location that a programmer identifies. It allows rewinding over regions of unconstrained size because our checkpointing scheme supports regions with arbitrary granularity and complexity. Our checkpointing scheme buffers fine-grain program changes into main memory and can dynamically grow the checkpoint buffer under demand. The checkpointing infrastructure also supports unlimited retries. This helps avoid all problems related with repetitively reproducing the bug under a precise bug-trigger environment, as well as the *long-latency* process related with restarting the application. A checkpoint-enhanced debugger helps to reduce develop-run-debug cycle time. This enhanced functionality and increased ease in debugging can easily convert into improved programmer's productivity in the process of identifying, isolating and fixing software bugs.

7.2 Automatic Backtracking Support for VPR

Backtracking refers to a set of algorithms that search for solutions in a given space of possible choices. The final solution of a backtracking algorithm is built upon a sequence of incrementally improved partial solutions, where each step either makes a guaranteed forward progress or maintains its current state

that is free of regressions. When evaluating an individual step that can potentially make a positive contribution to the final solution, the partial result is either committed (incorporated into existing partial solution) or discarded, depending on the evaluation result based on this individual step. Because the backtracking may perform intensive access to global data structures, the implementation of a backtracking algorithm usually cannot be contained in any stack-based design. We conduct our 3rd case study of checkpointing-supported applications using automatic software backtracking algorithm in *VPR*.

7.2.1 Overview

Versatile Placement and Route (*VPR*) [9, 54] is a software CAD tool for generating high-quality circuit layouts on array-based FPGAs. VPR places and routes on a wide variety of FPGAs and facilitate comparisons among different architectures. VPR's placement phase denotes to the process of placing various circuit components at different locations on the circuit board, where the routing phase is the process of connecting placed components through wiring while respecting all limitations and constraints in a given FPGA board. VPR implements a software backtracking algorithm in its placement phase and we present a simplified view of this backtracking process in Figure 7.2.

VPR's simulated annealing-based placement begins its processing with a set of circuit blocks in their original locations. The algorithm involves choosing a pair of blocks at random, swapping their positions, and evaluating the impact of this swap on a chosen cost function. E.g., Figure 7.2-(a) presents a given set of circuit board that has a total of four circuit blocks (`A` to `D`) requesting a placement. Among these available circuit blocks, VPR's placement algorithm randomly selects two circuit blocks (circuit `A` and circuit `B`) for swapping. If these two circuit blocks happen to reside on different nets, the pending swapping (backtracking) action will have a higher possibility of success, although the placement algorithm is not aware of this fact. The algorithm proceeds by evaluating the newly generated circuit based on this attempted swap. Evaluation criteria include estimating new placement cost when trying to fit the new circuit, power and heat dissipation based on the swap, as well as any impact on clock frequency and resource constraints. Depending on the impact (evaluation result), the swap may either be accepted or rejected. We show in Figure 7.2-(b) that the attempted swap of circuit blocks `A` and `B` is accepted. Thus this temporary swap becomes permanent by incorporating this partial order into the current solution. This process repeats until the cost function converges to a satisfactory value.

Current implementation of VPR's backtracking algorithm needs to **manually** save all necessary program states before attempting a swap, as well as backup all temporary results along all program paths when evaluating the attempted swap. Shall a discard happen, it **manually** restores all saved

program states from various complex data structures to undo the attempted swapping action. The code that implements VPR's backtracking algorithm is within the `try_swap` function—segments of `C` source code that are reasonably big and span across roughly 300 lines, with access to global variables, loops, pointer-chasing data structures and user-defined function callsites. Each user-defined function callsite within `try_swap` function needs to invoke its checkpoint-enabled version, which coexists with its non-checkpoint enabled version that will be called from outside of the checkpoint region.

Under the current `VPR` implementation, it is not an easy task to manually enable instrumentation over such a coarse-grain code region while providing correctness guarantees. It can easily become a maintenance nightmare for future development if a developer merely has the code but lacks sufficient training or background on the existing algorithm and its related data-structure implementations. VPR designers need to understand not only the placement algorithm, data structures, but also pay close attention to the details of manually saving and restoring necessary program states. Maintaining this manually-instrumented backtracking code will easily become a major productivity and backward-compatibility bottleneck when a programmer attempts to revise the evaluation algorithm or improve the associated data structures. This is a tedious and error-prone process that often has a negative impact on productivity, especially when improving the algorithm that results in necessary data structure changes.

7.2.2 Benefit

By exposing the compiler-friendly checkpointing APIs at the source-program level, our fine-grain checkpointing framework releases VPR designers from caring about fine details of conducting manual checkpointing instrumentation over the backtracking code region that can potentially be arbitrarily large and complex. VPR designers can ignore all details of manual checkpointing and instead simply mark the entire back-tracking function or region as a checkpointing program construct. Our checkpointing framework will then automatically transform the code to enable and optimize checkpointing over the region. This is a simple and straightforward process of manual region marking followed by automatic tool transformation. It greatly simplifies the previously tedious actions that a programmer has to manually instrument over all possible code paths that the checkpoint-enabled VPR program may exercise. VPR designers can instead focus on improving the algorithm itself and leave all tedious details to our compiler—a step that simplifies application programming interface, reduces programming difficulties, lowers the possibilities of introducing bugs due to the overwhelming amount of details and complexity that a programmer has to face when conducting manual instrumentation, thus improving overall end-user productivity.

7.3 Test Environment

Our compiler-based checkpointing framework builds on the LLVM [37, 38] open-source compiler infrastructure release 2.9 [1]. All analyses, transformations, and optimizations are organized as LLVM passes with explicitly specified dependencies using LLVM's built-in pass manager. This guarantees not only the proper ordering among the passes from our checkpointing framework, but also the smooth interactions with a large number of existing LLVM passes that are designed for general optimizations and have no particular design goal for checkpointing. We use debug builds for development and sanity test and release builds for the performance test. For evaluating checkpointing support on debugging, we use BugBench [41]—a collective suite containing various known software bugs with program inputs that trigger individual bugs. We select five BugBench applications that are both single-threaded programs and contain only buffer-overflow bugs. To evaluate applications with checkpoint-enabled software backtracking support, we use the most recently released version of VPR-5.02 [9], as described in section 7.2. We conduct all measurements on an Intel platform, with a Core i7 – 920 CPU, 4GB of DDR3-1600 RAM, running a fully patched Debian-6-i386 (kernel 2.6.32) with g++ version 4.4.5. All other prerequisite packages for LLVM are on their highest levels of supported versions.

7.4 Program Partition for Checkpointing Regions

The automatic checkpointing process relies on a manual step to partition programs for checkpointing regions. We manually convert each of our test application into Single-File Application (*SFA*) form and create checkpointing regions with respect to individual programs.

7.4.1 Checkpoint Region Partition

We partition each suitable application into three levels of granularity for checkpointing: small (S) region, medium (M), and large (L). After converting each selected `BugBench` application into its `SFA` form, we enclose the root cause and manifestation point of each bug in a region setup with minimal code span. We call this the small (**S**) region. We then grow the small region by both forward extending and backward extending the region boundaries, covering increased granularity and complexity of the source code. The result is a medium (**M**) region that contains a significant portion of the program, and a large (**L**) region

[1] We started our checkpointing work when LLVM was on its 2.7 release. LLVM has two new releases every year, thus we have kept upgrading our work when a new release became available. We finish the intensive coding stage and stabilize the checkpinting framework when LLVM was in its 2.9 release.

Applications	Region	Avg. insts	Avg. source lines	Execution Entries
bc-1.05	S	2.2 K	3	3
	M	208 K	430*	1
	L	305 K	1200*	1
gzip-1.24	S	0.9 K	1	1
	M	2.7 K	89*	1
	L	194 M	119*	1
man-1.5h1	S	1.4 K	14	1
	M	1.6 K	30*	1
	L	645 K	89*	1
ncompress-4.2	S	0.8 K	2	1
	M	149 K	149*	1
	L	231 K	163*	1
polymorph-0.4.0	S	1.5 K	2	1
	M	3.1 K	49*	1
	L	148 K	76*	1
VPR-5.02	M & L	67.1 K	268*	371 K

Table 7.2: Benchmarks and Checkpoint Region Properties

that is even coarser grain and can potentially cover the entire application. `VPR` has only one checkpoint region for properly implementing the back-tracking algorithm within its `try_swap` placement function. However, for `VPR` in particular, we have two checkpointing regions: a medium *(M)* region and a large *(L)* region, depending on whether the region is marked from the function callee's perspective (*M*) or the caller's perspective (*L*), respectively.

7.4.2 Checkpoint Region Properties

Table 7.2 summarizes the checkpoint regions' properties for each benchmark application after its region partition. Checkpoint regions are vastly different in size. For example, a small region usually contains around 1000 instructions and spans two or three lines of source code, while a large region can contain up to 195 million instructions (the entire gzip-1.24 when running under its given input) and covers 1000+

Apps	Region	Inline	RRE	FPRE	HRE	Hoist	Aggr	NRESE	DynOpti	ArrayOpti
	S	0	0	0	0	0	0	0	0	0
bc-1.05	M	17	21	56	3	2	9	0	70	0
	L	63	68	63	12	2	9	0	114	0
	S	1	0	0	0	0	0	0	0	0
gzip-1.24	M	2	35	16	9	10	0	1	0	0
	L	2	35	16	10	10	0	1	0	0
	S	7	0	0	0	0	0	0	0	0
man-1.5h1	M	8	0	0	0	0	0	0	0	0
	L	79	2	19	0	0	0	0	18	0
	S	1	0	0	0	0	0	0	0	0
ncompress-4.2	M	3	0	0	0	0	0	0	0	0
	L	18	7	3	0	0	0	0	0	0
	S	1	0	0	0	0	0	0	0	0
polymorph-0.4.0	M	1	0	0	0	0	0	0	0	0
	L	6	2	3	0	0	0	1	0	0
VPR-5.02	M	0	20	0	7	0	0	0	0	0
	L	0	20	0	7	0	0	0	0	0

Table 7.3: Compile-time statistics of individual checkpointing optimizations

lines of source code (bc-1.05).[2] Our region partition scheme is flexible because we are capable of supporting program regions with arbitrary size and complexity. Users can extend or shrink checkpoint regions to match the requirement, provided the begin-region marker (`start_ckpt`) always *dominates* the stop-region marker (`stop_ckpt`). More details on checkpoint region-partition requirements are available in section 3.3.1.

7.5 Static Evaluation of Checkpointing Optimizations

Once a compiler optimization identifies an opportunity that satisfies the condition(s) it is examining, it will perform the designated transformation(s) on that opportunity at compile time. Each optimization pass keeps a number of counters to track important optimization principles and will increase the corresponding counter each time when the optimization matches a suitable principle after completing a designated transformation. Thus when a optimization pass finishes the processing of a SFA application

[2]Note that *M* and *L* regions always contain user-defined functions within our selected BugBench applications, thus the number of source lines presented in Table 7.2 marked with * only indicates the lower bound of possible source-code span.

at its input, it can quickly discover whether this optimization is effective by examining its internal counters. We call these compile-time counter statistics *hits*. After compiling a test application, a LLVM pass can simply check the respective counter's value and rapidly discover the effectiveness of the optimization. The higher the counter value(s), the more frequent the compiler caught opportunities from the given input program and performs desired transformations. When the counter shows value zero (no hits), it indicates the respective transformation doesn't catch any opportunity (not effective) after processing the given test input.

Table 7.3 gives the *hits* values for each testing application under all available checkpointing partitions. These are the compiler's compile-time statistics when our checkpointing infrastructure independently performs individual optimizations on SFA inputs with checkpointing region partitions. It provides a quick statistical preview of whether an optimization is effective on certain benchmarks and sets expectations of different optimizations on input applications. We consider a particular optimization a *success* if the *hit* counts in any given column is not always zero. Even a low counter value could indicate a significant transformation event (e.g., a transformation that happens within a loop could be potentially significant, however we cannot tell from the hit counter statistics alone.). The only always-zero column is array optimization (*ArrayOpti*). Despite its aggressive algorithm by design, *ArrayOpti* doesn't catch any suitable opportunity from the given BugBench programs on the provided input data. All other checkpointing optimizations effectively perform transformations and we can quantitively measure their results through testing.

7.6 Comparison with Existing Checkpointing Solutions

In this section we compare our compiler-based checkpointing solution with two alternative software-only approaches to checkpointing that are both considered state of the art in their own respect: a coarse-grain checkpointing library, and a fine-grain software transactional memory scheme supported by a commercial compiler.

7.6.1 Comparison with libCKPT

Library-based checkpointing schemes backup all memory used by the running process, thus the checkpointing overhead closely correlates to the size of memory the process uses (memory footprint) at checkpointing time. We use libCKPT [53] as a representative of a recent library-based software-only checkpointing solution.

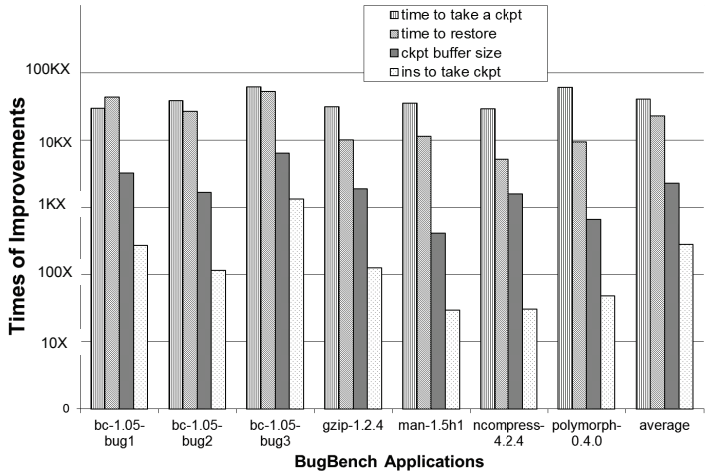


Figure 7.3: Overall Coarse-grain comparison: our base checkpointing solution vs. libCKPT

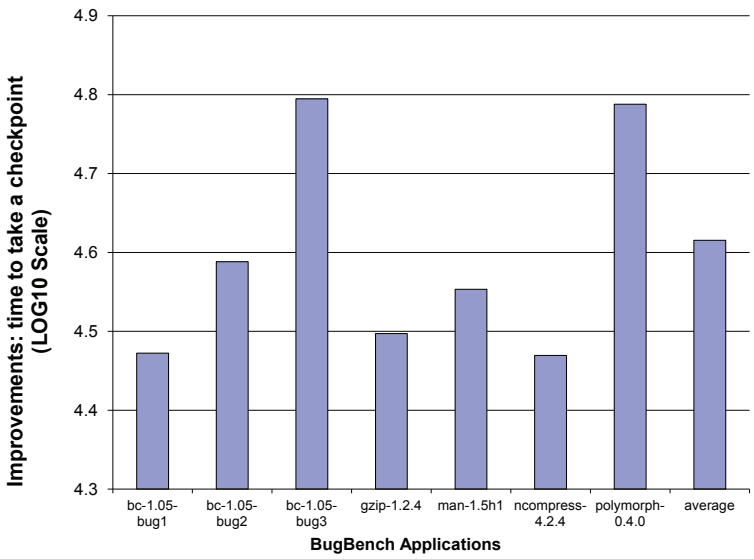


Figure 7.4: Improvement on time to take a checkpoint

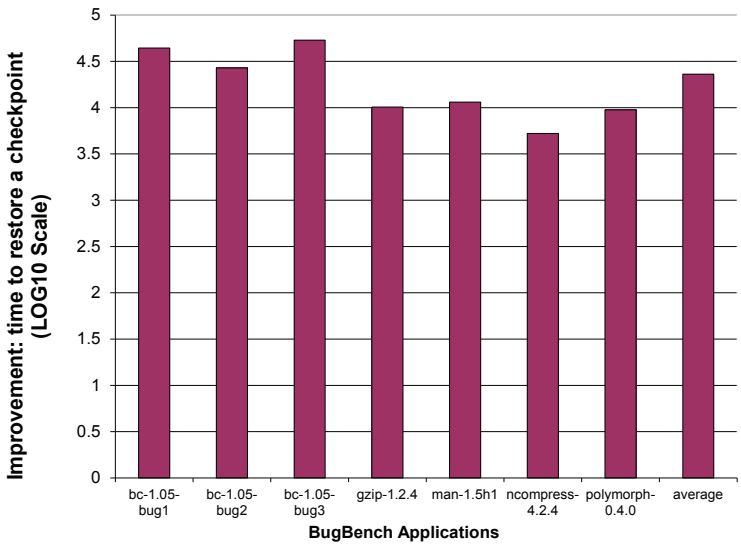


Figure 7.5: Improvement on time to restore a checkpoint

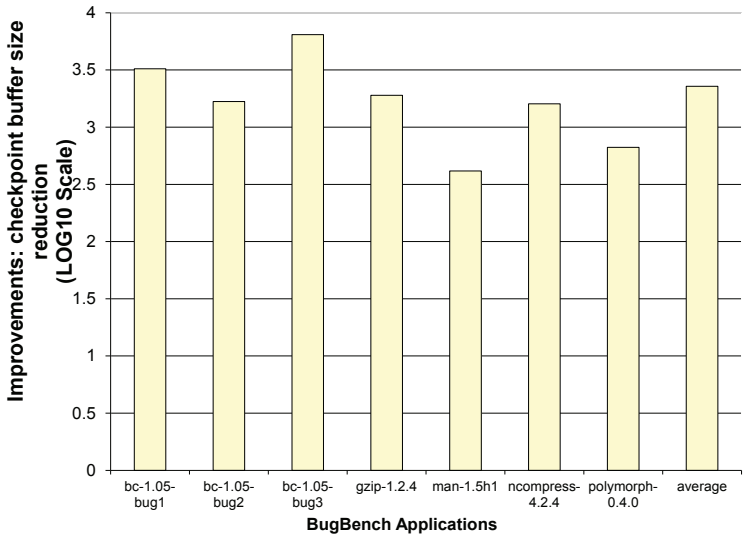


Figure 7.6: Improvement on checkpoint buffer-size reduction

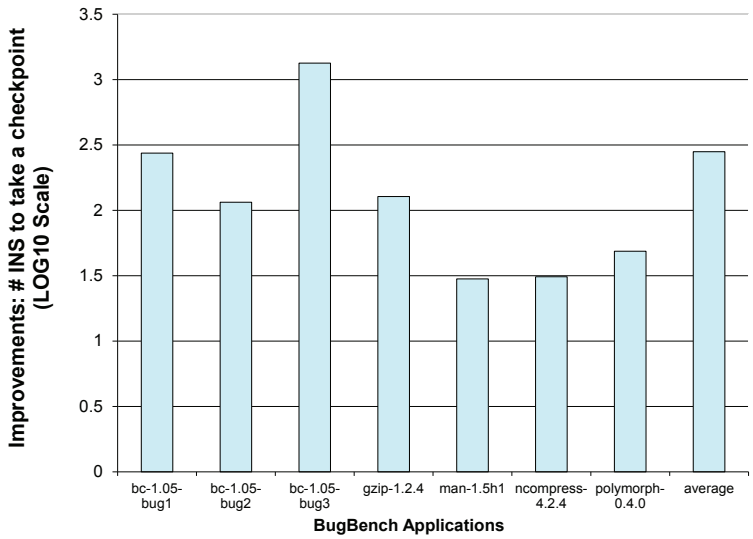


Figure 7.7: Improvement on number of instructions to take a checkpoint

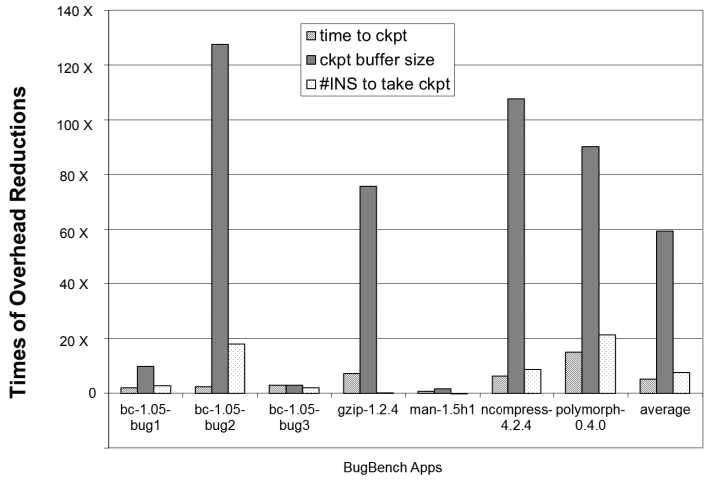


Figure 7.8: Fine-grain comparison: with ICCSTM

Figure 7.3 shows an overall comparison between our compiler-based fine-grain checkpointing scheme and *libCKPT's* library-based coarse-grain checkpointing approach. We place the selected BugBench applications on x axis, and examine on y axis improvements over a total of four different checkpointing performance-related aspects from each application: (i) time to take a checkpoint, (ii) time to restore a checkpoint, (iii) checkpointing buffer size, and (iv) number of instructions needed to conduct a checkpoint. Figure 7.3 presents improvements made from our fine-grain checkpointing scheme over the *libCKPT* baseline. Due to the significance of the performance improvement, all data presented in Figure 7.3 are on logarithmic scale. Note that since we are conducting checkpointing performance analysis between a fine-grain scheme and a coarse-grain approach, Figure 7.3 represents only our checkpointing compiler transformations without activating any checkpointing optimization(s). Figure 7.3 also provides arithmetic averages for each category for a quick overall comparison. We further separate Figure 7.3 into four individual figures (Figure 7.4 - Figure 7.7 respectively), each representing its own category for detailed zoom-in views.

Figure 7.3 shows that our fine-grained checkpointing approach can provide over 1000X improvement when comparing with coarse-grain libary-based checkpointing, for both the time-to-take a checkpoint and the time-to-restore a checkpoint. The improvement in checkpoint buffer size and the number of instructions needed to service a checkpoint are within the range of 100X to 1000X.

This huge improvement on overhead reduction is mainly from two aspects. First, it is the difference between checkpointing to permanent storage (hard disk, in the case of *libCKPT*) and checkpointing to main memory (our fine-grain checkpointing). The performance difference between the two storage systems reflects into the difference of checkpointing performance. Second, it is the difference between coarse-grain checkpointing (*libCKPT*) and fine-grain checkpointing (our scheme). Coarse-grain checkpointing has high overhead by copying entire memory range or complete objects. In *libCKPT*, it copies the entire range of the running process's memory footprint. In our fine-grain checkpointing, we only copy a memory location to our backup buffer if and only if there is at least one write into it. Thus we checkpoint only the bare minimum to facilitate a loss-free recovery. A comprehensive optimization framework aims to further reduce this overhead. The huge overhead reduction shows great performance potential for conducting checkpointing over fine granularity.

7.6.2 Comparison with ICCSTM

We further compare fine-grain software checkpointing overhead through supporting single-threaded speculative optimization between Intel's Software Transactional Memory [2, 57] (*ICCSTM*) and our

compiler-based checkpointing solution. *ICCSTM* is a software solution for supporting optimistic parallelism and bases its support on Intel's production-quality `C/C++` compiler. Just like other STM systems, *ICCSTM* supports speculative parallel execution through write-buffering and dependence tracking on read set and write set from individual thread for multiple threads at run-time. The differences in performance between the two software packages are expected to come from the different focus and specialization on their respective main use cases.

Figure 7.8 compares *ICCSTM* to our baseline compiler-based checkpointing solution (with no optimizations). We find that our solution outperforms *ICCSTM* in almost all cases. On average, our solution outperforms the time-to-take a checkpoint for *ICCSTM* by 5X, and the number of instructions needed to take a checkpoint by 8X. The largest difference is in improving (reducing) checkpoint buffer size, as *ICCSTM*'s buffer is almost 60X larger than that used for our scheme.

ICCSTM is mainly optimized to support program parallelization based on relatively short transactional regions while our checkpointing scheme is optimized to support single-thread speculation, or debugging for larger program regions. Based on the limited description available [2, 57], *ICCSTM* uses only basic compiler optimizations such as inlining and a very simple form of partial redundancy elimination while our checkpointing scheme employs a comprehensive optimization framework, trying to reduce overhead from all possibilities. Furthermore, to the best of our knowledge, *ICCSTM* doesn't optimize for the single-threaded speculative execution case. In this special case of speculation support, tracking of a single thread's read-set could be safely omitted. In contrast, our checkpointing scheme benefits from being specialized for the single-thread case. Specifically, we track only the write set for the speculative thread via an efficient implementation based on undo-logging. In the common case where speculation is successful, undo-logging avoids expensive lookups on reads for matching prior writes, and also the copies of writes to shared memory on commit.

Overall, it is expected that our fine-grain checkpoint support will have lower overheads, and/or better cache behavior than a write-buffering STM such as *ICCSTM*. Due to our undo-log design and aggressive compiler optimizations under the single-thread application environment, our fine-grain checkpointing scheme outperforms Intel's *ICCSTM* by a large margin.

7.7 Effectiveness of Checkpointing Optimizations

To evaluate our checkpointing optimization framework, we run each individual optimization over every testing application's *M* and *L* regions. We gradually increase the number of optimizations on each

```
BEGIN
  bool bCont = true;
  while (bCont) do
    // preparation stage:
    bCont = false; bCont |= doInline (); bCont |= doPreOpti ();
    // Redundancy Eliminations:
    bCont |= doRRE (); bCont |= doFPRE (); bCont |= doHRE ();
    // Hoisting:
    bCont |= doHoist ();
    // Aggregations:
    bCont |= doSimpleAggr (); bCont |= doComplexAggr ();
    // NRESE:
    bCont |= doNRESE ();
    // Dynamic Optimization and Array Optimization:
    bCont |= doDynOpti (); bCont |= doArrayOpti ();
    // Post Optimizations:
    bCont |= doPostOpti ();
  end
END.
```

Figure 7.9: Algorithm of checkpointing optimization ordering

checkpoint region until all available optimizations are exhaustively applied. We focus our evaluation on the effectiveness of checkpointing overhead reduction as measured by the following three metrics: checkpoint buffer size reduction, reduction in the number of `backup` service calls, and optimization(s)' impact on redundancy rate.

7.7.1 Optimization Ordering

$$Inline \rightarrow PreOpti \rightarrow RRE \rightarrow FPRE \rightarrow HRE \rightarrow Hoist \rightarrow Aggr \\ \rightarrow NRESE \rightarrow DynMeOpti \rightarrow ArrayOpti \rightarrow PostOpti \quad (7.1)$$

When conduct testing on checkpointing optimizations, we always incrementally perform applicable optimizations in a known-good order. This order needs to not only intuitively satisfy all implicit and explicit dependencies among different checkpointing optimization passes, but also naturally cooperate with LLVM's existing non-checkpointing optimizations. We present the optimization's ordering algorithm in Figure 7.9.

As shown in Figure 7.9, the optimization ordering algorithm contains a key `while` loop, where all available checkpointing optimizations appear inside this loop in a pre-determined order. Checkpointing optimization passes perform their designated activities respecting the order they appear inside the while loop. Each individual optimization will return a boolean *true* value if it hits any opportunity and

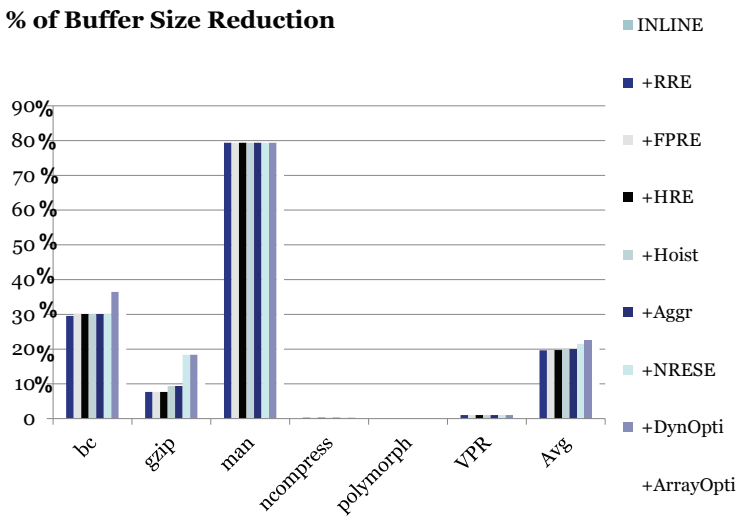


Figure 7.10: Optimization impact on checkpointing buffer size: M region

performs a designated transformation, and *false* otherwise. Thus the key `while` loop in Figure 7.9 will break only if none of the existing checkpointing optimization conducts any effective transformation. We also present the explicit single-pass checkpointing optimization order in formula 7.1 (the explicit order within the key `while` loop).

Formula 7.1 is the default order to apply checkpointing optimizations and it naturally resolves all implicit or explicit dependencies among optimization passes. This order is generated by LLVM's pass manager, but it is by no means the only applicable order. The optimization framework is flexible to adopt any optimization order that resolves existing dependencies. For ease of comparison, the rest of the evaluation on checkpointing compiler optimizations will respect this simple optimization ordering.

7.7.2 Checkpoint Buffer Size Reduction

Figure 7.10 and Figure 7.11 shows the compiler optimization impact on checkpoint buffer size when all optimizations are incrementally and accumulatively applied while we follow the default optimization order. The effectiveness of the optimizations depends on the region size, program structure (especially loops and user-defined functions that have `backup` operations), as well as store intensity within the region. A larger region usually has more opportunities for optimization, thus we normally observe that overall optimizations are more effective on large checkpointing regions than on medium regions. We notice that **RRE** is the most-effective single optimization among all available optimizations in the entire framework. This is mainly due to its aggressive algorithm enhanced by the leader-hoisting step. As

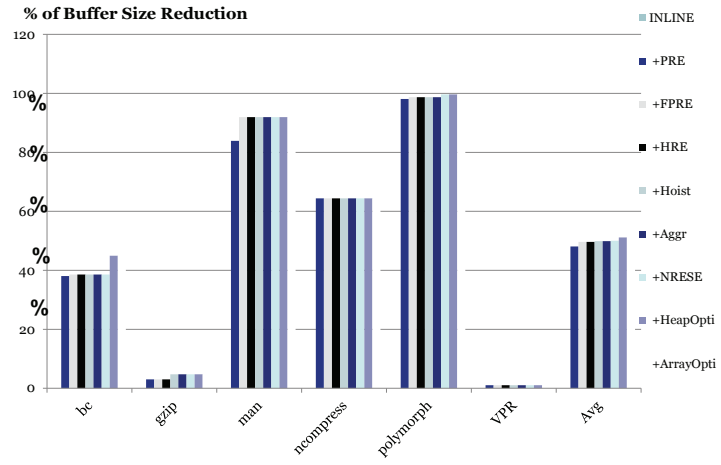


Figure 7.11: Optimization impact on checkpointing buffer size: L region

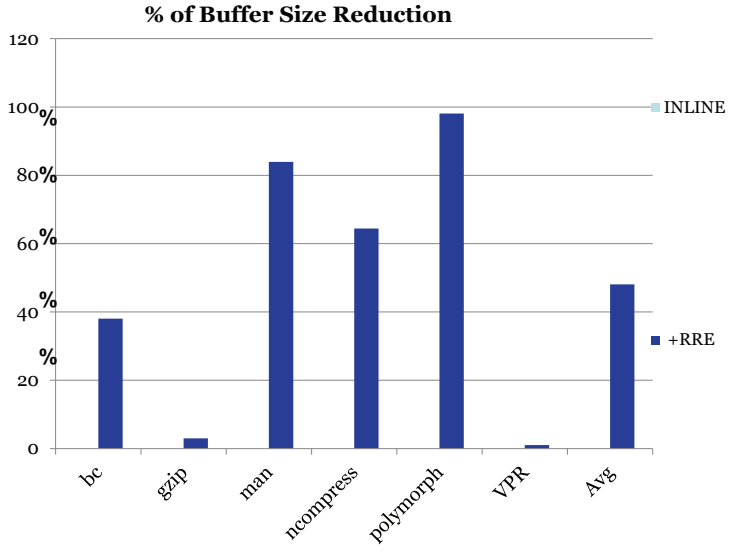


Figure 7.12: Optimization impact zoom (L region): before FPRE (Inline + RRE)

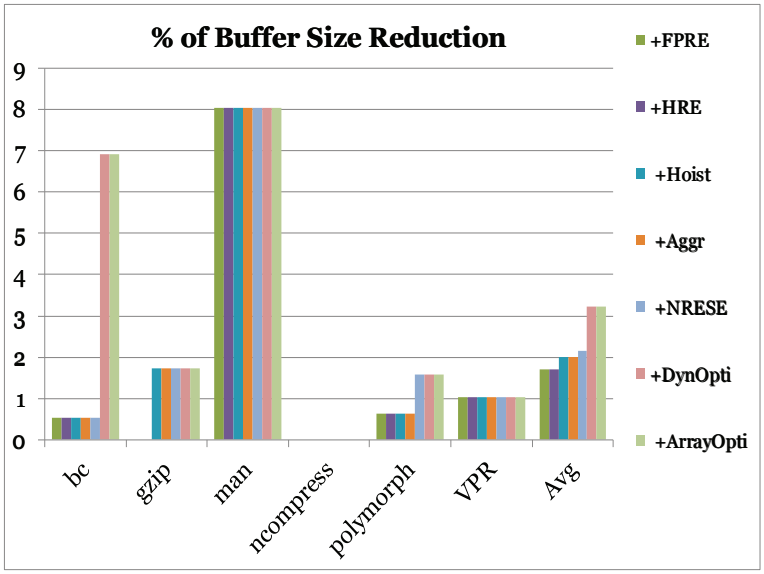


Figure 7.13: Optimization impact zoom (L region): after FPRE (FPRE + HRE + . . . + ArrayOpti)

shown in Figures 7.10 and 7.11 respectively, *RRE* reduces the checkpoint buffer size by almost 80% in `man` (Medium region) and 92% in `polymorph` (Large region). When optimizations are incrementally applied, we observe a stable trend of buffer size reduction for both *M* and *L* regions. The performance results show that our compiler optimizations either exploit opportunities for optimization and hence improve checkpointing efficiency after the optimization transformation, or keep existing checkpointing performance without introducing negative effects that undermine the established gains (regressions). We investigate and fine tune all checkpointing optimization algorithms to guarantee the regression-free property. Overall, the optimizations reduce checkpoint buffer size by an average of 52% for the *L* regions and 22% for the *M* regions.

Since *RRE* is the single most-effective optimization, we are interested in observing its individual contribution, as well as separating it from other optimization passes and analyzing the individual and combined behaviors of the rest of the available optimizations. Figure 7.12 shows a *zoom-in* effect that isolates *RRE* from the remaining optimizations. By grouping the optimizations into two segments, Figure 7.12 shows the optimization impact by only performing *Inline* and *RRE*, while Figure 7.13 shows the *post-RRE* impact for all remaining optimizations (from *FPRE* to *ArrayOpti*). Notice that we group *Inline* and *RRE* together and make *Inline* precede *RRE* because *Inline* is a prerequisite of *RRE*. We thus make them reside in the same group while respecting the pre-defined optimization ordering discussed in section 7.7.1.

Figure 7.12 confirms our previous observation that *RRE* is the single most-effective optimization

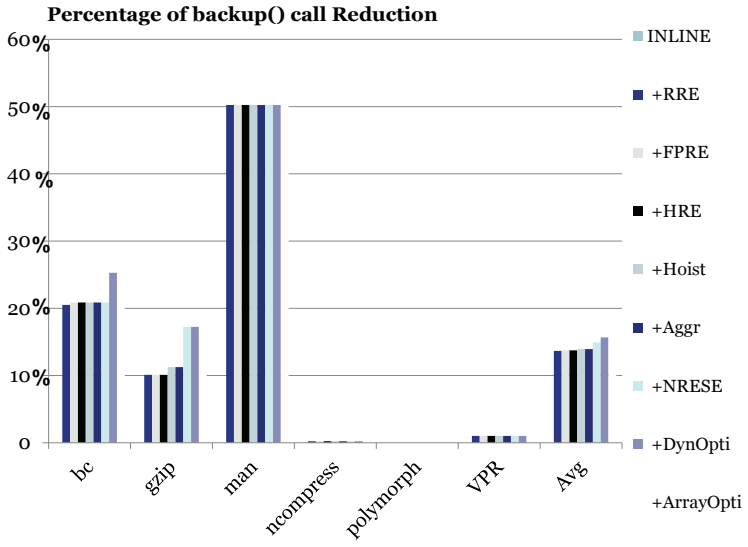


Figure 7.14: Optimization impact on backup call reduction: M region

among all available optimizations. *RRE* alone reduces checkpointing buffer size by up to 92% in `polymorph` (L) and more than 80% in `man` (L). On average, *RRE* reduces around 50% of checkpointing buffer redundancy in `L` regions. Figure 7.13 presents the post-*RRE* effect. It shows the behaviors and performance of all remaining optimizations and the results are normalized after *RRE*. In post-*RRE* era, all optimizations matter. However, we cannot find any single optimization that has a similar impact. E.g., *DynMemOpti* delivers around 6.5% in `bc`, while *FPRE* delivers almost 8% in `man`. The potential impact largely depends on the nature of the testing application and the optimization opportunities the checkpointing region exposes. When an optimization discovers a matching opportunity, it transforms the code and makes the performance impact (e.g., *DynMemOpti* in `bc` and *FPRE* in `man`). Otherwise it maintains its regression-free property. Combined post-*RRE* optimizations deliver an accumulative average of slightly more than 3% for `L` regions. It has similar impact on `M` regions.

7.7.3 Backup Operation Reduction

In addition to buffer size reduction, our compiler optimizations also reduce the total number of `backup` calls—another metric for estimating and evaluating the optimization impact on checkpointing overhead reduction. Reduction on the total number of `backup` calls closely correlates with the reduction of checkpointing buffer size, since the optimizations achieve both goals when reducing checkpointing overhead. Though the correlation may not be precisely linear, we expect that they will be close enough to present similar behaviors in optimization impacts. Figure 7.15 and Figure 7.14 show that

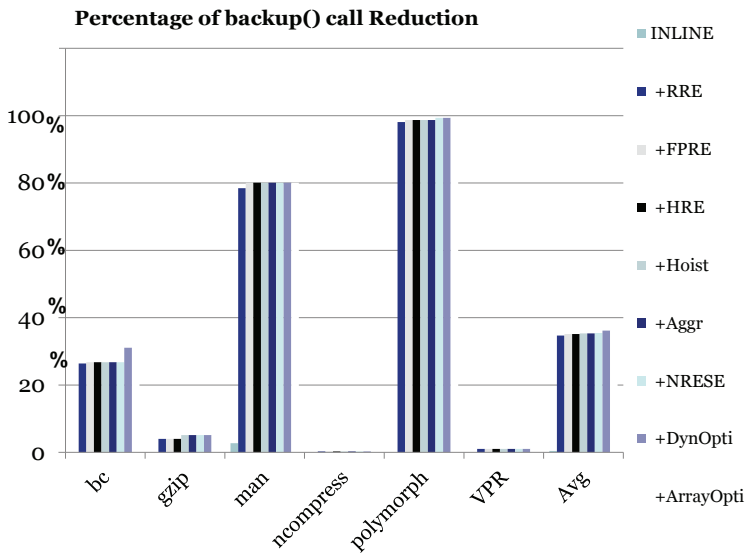


Figure 7.15: Optimization impact on backup call reduction: L region

our optimizations reduce the total number of `backup` calls by an average of 36% for the *L* regions and an average of 15% for the *M* regions respectively, after all optimizations have been applied following the same scheduling order discussed in section 7.7.1. We notice that *RRE* is the most effective single optimization within the entire optimization framework in both cases. This confirms with our observation in section 7.7.2. The `backup` call reduction closely correlates to the optimization impact on buffer size reduction presented in Figure 7.11 and Figure 7.10.

7.7.4 Impact on Redundancy Rate

After exhaustively applying all available optimizations, it is important to understand the amount of remaining redundancy in the checkpoint buffer. This is a measurement to predict potential future optimization opportunities that may remain. We quantify this by studying the optimizations' impact on the region's *redundancy rate* (*RR*), as defined earlier in section 5.4. Figure 7.16 and Figure 7.17 illustrate the changes of *RR* made by the optimization framework for both *M* and *L* regions while incrementally applying available optimizations.

Figure 7.16 indicates that our optimizations are more effective in reducing redundancy rate in *M* regions, since the *RR* reductions in *M* regions are more significant and the highest *RR* in *M* regions is around 18% (`man`) after all optimizations. This is because there are fewer `backup` calls (and less redundancies) in *M* regions. Thus an optimization that removes a smaller number of `backup` calls can potentially generate a bigger impact on redundancy rate reduction. Comparing with *M* regions, the

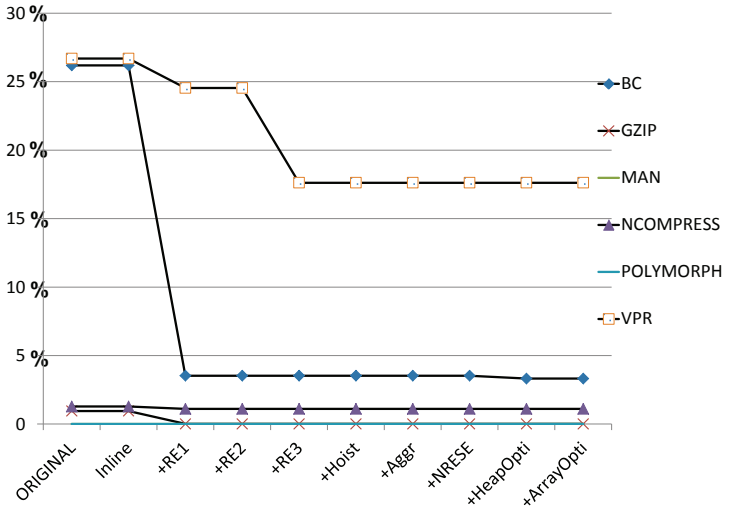


Figure 7.16: Optimization impact on redundancy rate: M region

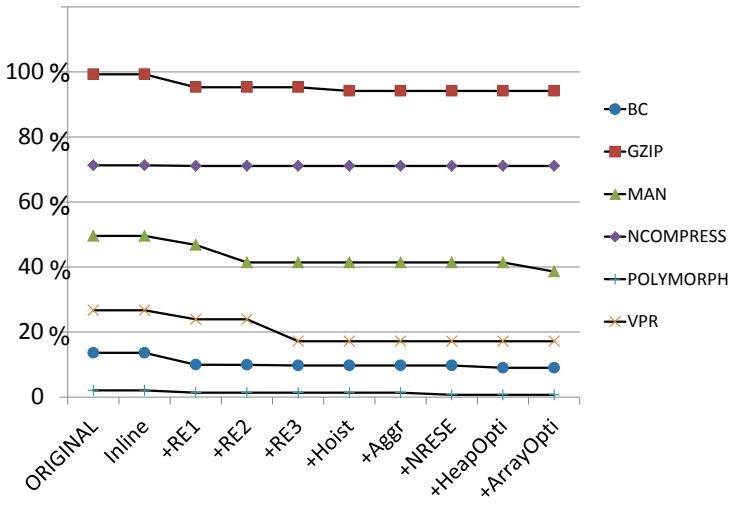
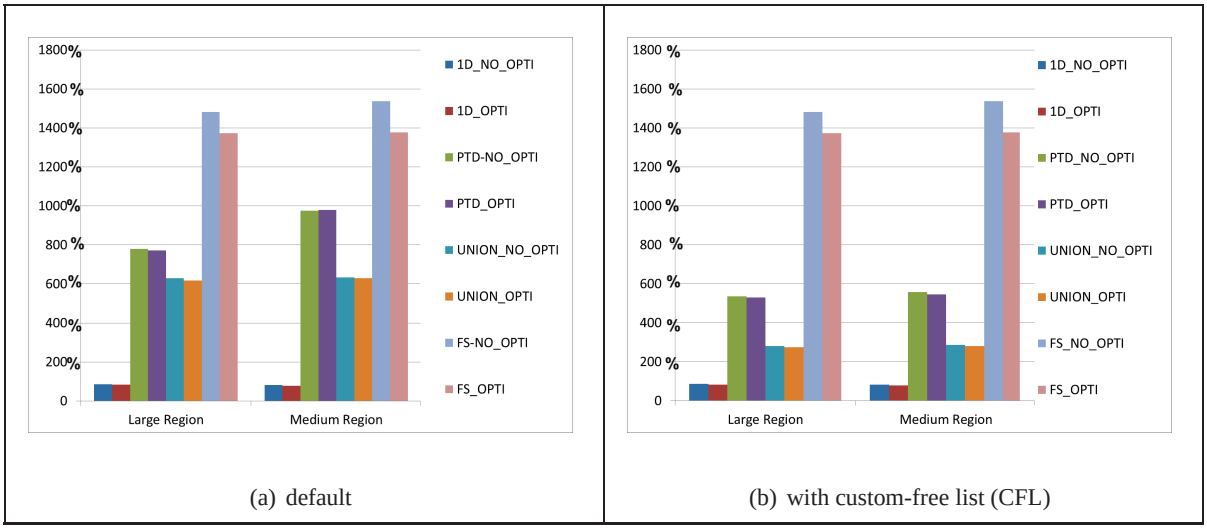


Figure 7.17: Optimization impact on redundancy rate: L region



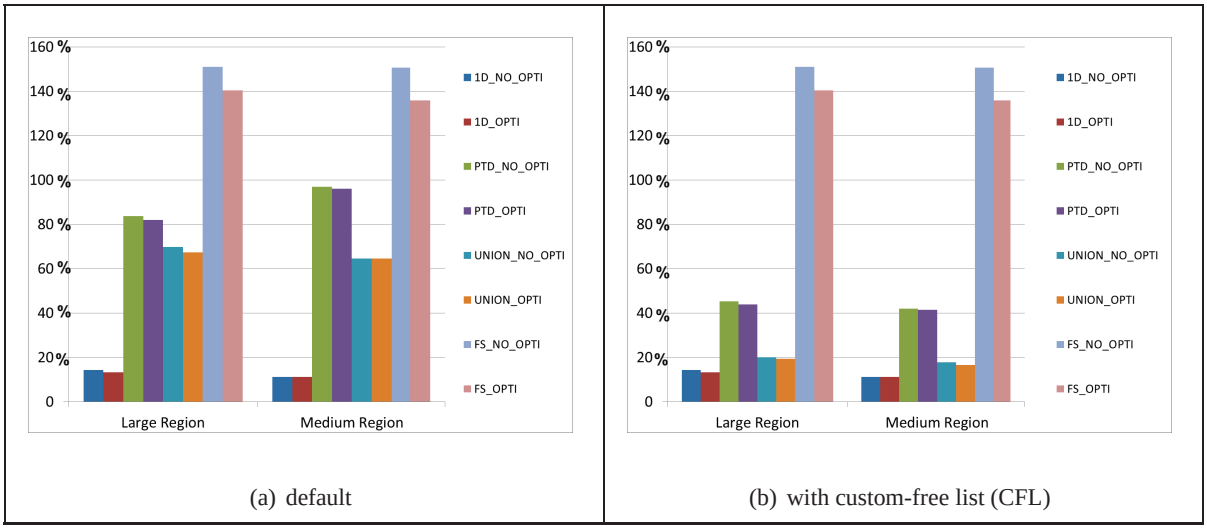
(a) default (b) with custom-free list (CFL)

Figure 7.18: Checkpoint buffering overhead on VPR's try_swap function

optimizations can normally remove more `backup` calls in *L* regions. However, since there are many more redundancies to eliminate, the optimization's effectiveness on reducing redundancy rate is not as significant as those in *M* regions. Notice that in the case of *L* regions, three applications (`gzip`, `ncompress`, and `man`) still have very high *RR* even after applying all available optimizations (92% for `bzip`, 72% for `ncompress`, and 39% for `man`). For each case, we manually examine the `LLVM` produced code after applying all available optimizations and attempt to analyze the root cause of this high redundant rate. We conclude that this remaining-high *RR* is due to redundancy caused by extensive use of pointers that our current optimization framework is incapable of handling. Effectively handling `backup` operations over memory regions passed by pointers strongly suggests the need of a precise pointer analysis over a checkpointing region. The remaining high *RR* also suggests potentially abundant work that future research may continue to explore. We will highlight this as a potential future direction in section 8.

7.8 Checkpointing Performance on VPR

Software-only checkpointing is not free: it comes with a cost on conducting memory `backup` operations that can potentially have negative impact on an application's performance when the checkpointing service is enabled. In this section, we report the performance result on `VPR` with activated checkpointing service.



(a) default (b) with custom-free list (CFL)

Figure 7.19: Checkpoint buffering overhead across entire VPR

7.8.1 Performance of Different Buffer Schemes

Software checkpointing has negative performance impact on VPR because checkpointing incurs software-only overhead that is not part of the original application. We present the negative performance impact on checkpointing-enabled VPR on the granularity of its placement function (`try_swap`) and over the execution of the entire VPR in Figure 7.18 and Figure 7.19 respectively. For both of VPR's *M* region and *L* region, we report the slowdowns of VPR's application performance after enabling checkpointing when using all available buffering schemes, with and without checkpointing optimizations for each possible buffering. We examine a total of four available buffering schemes: one-D array, *PTD* hash table, *Union* hash table, and *fixed-size* (*FS*) hash table. Please refer to chapter 5 for more details on individual buffering schemes. Under each available buffering scheme, we measure performance once with all checkpointing optimizations turned off (*NO_OPTI*), and measure performance again with all checkpointing optimizations turned on (*OPTI*). Thus the values presented on y-axis are slowdowns (in percentage), indicating the amount of relative overhead caused by checkpointing using different buffering schemes. Figure 7.18 shows the performance normalized to VPR's backtracking algorithm (`try_swap` function) only, and Figure 7.19 shows the same overhead normalized across the entire VPR.

It is clear that the one-D array buffer delivers the best overall performance (the smallest amount of checkpointing overhead) for checkpoint-enabled VPR among all available buffering schemes. This confirms with our previous analysis in section 5.5 that one-D array performs the best when comparing with hashtable-based schemes. When normalized to VPR's backtracking function (`try_swap`), one-D array causes the mildest slowdown of around 50% (without optimization) and 45% (with full

optimizations). Optimizations have a relatively small impact on VPR because the performance difference between unoptimized VPR and fully optimized VPR is relatively insignificant. Among the three hashtable-based buffering schemes, the one with Union-type hash node performs the best. This again complies with our former report on buffer-scheme analysis in section 5.5 that Union hashtable delivers the best overall performance among all hashtable-based implementations. Notice that though Union-hash table outperforms the other two hashtable-based buffer implementations, its raw performance still lags far behind one-D array buffering (600% vs. 50%).

Figure 7.19 presents the same information as those given in Figure 7.18, but under different scaling. Figure 7.19 shows that the checkpointing overhead is normalized globally (across the entire VPR's execution), while Figure 7.18 shows the same checkpointing overhead that is normalized locally (over the try_swap backtracking function only). Figure 7.19 shows that under global scale, one-D array buffer indefinitely and universally exhibits the best overall checkpointing performance after applying all available optimizations: a mild slowdown of 15% over both medium and large regions.

7.8.2 Fine Tuning on Hash-Table Schemes

Our checkpointing optimization framework shows non-uniformly distributed optimization effects: its impact on VPR is less significant comparing with most applications in the BugBench suite. We intend to explore further potential on reducing checkpointing buffer overhead by injecting application-specific knowledge and conducting fine-grain and manual application-specific performance tuning.

We qualify that hashtable-based buffer schemes suffer from two major performance bottlenecks: (i) cache misses, and (ii) dynamic-memory management overhead. We thus focus our application-specific fine-tuning efforts on these two aspects. First, we perform prefetching on hash table. We gain checkpoint region-specific knowledge on the number of unique backup memory addresses from profiling. By prefixing the number of hash buckets to slightly more than the total number of unique backup addresses, we guarantee that the average length of each bucket chain will not be more than one (one or less). Second, we develop a custom dynamic memory manager: managing an in-memory custom-free list (*CFL*) through the use of a doubly linked list (*linklist*) of hash nodes. This custom memory manager intercepts all dynamic memory management calls (malloc, calloc, realloc, and free), and redirects them to the CFL whenever possible. Freeing an existing hash node will insert it at the head of the CFL, while allocating a new hash node will always check for an available and suitable node from the head of the CFL. If there is any available node in the CFL, that node is returned instead of mandating a real dynamic memory allocation at the runtime. By building a custom dynamic memory

allocator, we eliminate most of the overhead related with dynamic memory management (allocation and free).

Figure 7.18(b) and Figure 7.19(b) present the results of exhaustive manual efforts for application-specific performance tuning. The `union` hashtable benefits the most from fine tuning: overhead reduction from 600% to 300% (improves around 50% in Figure 7.18(b)). The `PDT` hashtable also benefit from this tuning effort: 970% to 550% (improves around 43% in Figure 7.18(b)). However, `fixed` hashtable doesn't benefit from the custom tuning because such tuning efforts have already been built into its implementation details. By design, `fixed` hashtable already employs a custom memory manager to complete its operations as necessary. Further more, its implementation detail is necessarily and unavoidably conducting a form of the bucket-array prefetch operation. Notice in Figure 7.19(b) that for the most efficient hashtable-based scheme (`Union`-node hash table) enhanced with all possible application-specific fine-tuning efforts, its performance still lags behind one-D array buffering by a fair margin: 19% to 15%, respectively.

7.9 Summary

In this chapter, we introduce two additional interesting applications, namely checkpointing support for debugging (*Debugger*) and checkpointing-enabled software backtracking using VPR (*VPR*). With our efficient software-only checkpointing support, the applications are either obtaining improved performance (*Debugger, VPR*), gaining new functionality that is not normally available otherwise (*Debugger*), or improving programmer's productivity (*VPR*). With our fine-grain checkpointing support, *Debugger* now supports reverse execution, with unlimited number of retires and no need for program restart. With our software-only checkpointing support that naturally adapts to program changes, *VPR* developers can instead focus more on the algorithm and program logic, thus convert the automatic compiler support for reduced development cycle time into improved productivity.

We present the checkpointing region partitioning scheme and give details on the checkpointing region properties. We compare our software-only checkpointing work with existing solutions, including a coarse-grain approach based on `libCKPT` and a fine-grain approach based on `ICCSTM`. We show significant gains through overhead reductions by comparing with `libCKPT`. We also show that by focusing on aggressive compiler optimizations, we can outperform the highly competitive `ICCSTM` by a large margin.

We select `BugBench` and `VPR` as our representative benchmarks and evaluate checkpointing

performance through our compiler transformations and optimizations. We show that the optimizations are effective to reduce up to 98% of checkpointing buffer size and remove up to 95% of `backup` service calls from `BugBench`, at expense of 15% checkpointing overhead from `VPR`. We further conduct detailed comparison among all available buffer schemes and conclude that one-D array buffer is the most efficient buffer scheme at expense of data redundancy. In most cases, one-D array buffer significantly outperforms all hashtable-based buffers even after the latter receive heavy custom optimizations and application-specific fine tuning.

Chapter 8

Conclusion and Future Work

Checkpointing is conventionally used as a system-level scheme to improve failover, and to enhance reliability and security. The vast majority of checkpointing implementations are implemented in software to checkpoint the entire application's memory into persistent storage (normally hard disks) and restore from the saved checkpoints to facilitate error recovery. Traditional checkpointing schemes incur prohibitive overhead that render them inapplicable for any performance-sensitive application.

In this thesis, we propose a fine-grain software-only checkpointing scheme that is based on per-store instrumentation enabled by compiler transformations. In addition, we design and implement a comprehensive compiler optimization framework that takes checkpoint-enabled code and aggressively optimizes it from many different perspectives, aiming to reduce checkpointing overhead to its minimum. Our fine-grain software-only checkpointing scheme outperforms a state-of-the-art library-based coarse-grain software-only approach by exhibiting 1000X+ less overhead. Our compiler optimizations further improve our baseline checkpoint-enabled code by eliminating up to 98% checkpointing overhead.

We further explore three key applications that leverage our fine-grain and compiler-based checkpointing framework to enable unique functionality. These checkpoint-enabled applications expose checkpointing APIs to user level to gain detailed user control, allow program backtrack to a previously specified location for unlimited number of retires and free of restart, remove programmers from tedious details of manual checkpointing instrumentation and better focus on improving the application itself. The new functionality and enhancements made for programming and debugging tools can easily convert into improved programmer productivity.

This thesis lead to the publication of a number of papers [67–70] and the public release of *ChuckPoint*—source code of our fine-grain and compiler-driven checkpointing transformation and

optimization framework based on `LLVM`, together with a PIN-based runtime library. The PIN-based runtime library focuses on checkpointing functionality verification, profile-driven redundancy analysis, and provides a handy platform for new opportunity exploration. The package release also includes full source codes of benchmark applications (*BugBench* and our micro applications) instrumented with various checkpoint-region partitions.

8.1 Contributions

This dissertation makes the following contributions.

1. **A comprehensive compiler-based checkpointing framework.**

   We present a software-only fine-grain checkpointing framework that is based on compiler transformations and optimizations using *LLVM*. This system is completely independent of any checkpointing hardware support. Compiler transformations enable checkpointing on any user-annotated program region with arbitrary size and complexity. We pay close attention to details on handling corner cases including function-pointer callsites and premature returns from checkpointing region. A large and comprehensive compiler optimization framework operates on the checkpointing-enabled user program, attempting to aggressively reduce checkpointing overhead from many possible perspectives. To our best knowledge, this is the first checkpointing work that is based on compiler-driven program instrumentation and optimization for fine-grain checkpointing.

2. **Effective optimizations for reducing checkpointing overhead.**

   We demonstrate that the checkpointing framework can be used to support iterative reverse execution for debugging purposes. By exposing checkpointing APIs on application's level, a programmer can directly involke checkpointing APIs as user-level commands to conduct respective checkpointing activities inside a debugger. This naturally extends a debugger's support to checkpointing and allows an easy entry for programmers who need to try out or utilize the checkpointing functionality without prepackaged OS or hardware support. Comparing with a library-based coarse-grain checkpointing approach, we achieve significant overhead reduction: **1000X+** times less overhead. We also consistently outperform *Intel*'s state-of-the-art software transactional memory solution (*ICCSTM*) by up to **60X**.

3. **Demonstration of the limitations of an overly fine-grain and performance-sensitive application of compiler-based checkpointing.**

   We illustrate some inefficiency of the checkpointing framework when trying to support an ultra fine-grain application—overlapping program execution with delinquent loads in `MCF`. We conduct detailed analysis on identifying significant delinquent loads through PIN-based dynamic instrumentation. We discover the *persistent* properties of `MCF`'s significant DLs. We explore value predictions under the context of software speculation and find that simple last-known value predictor and constant-value predictor perform extremely well to satisfy the demand of the speculation scheme. We further perform manual program transformation to enable speculative execution utilizing both control-speculation and data-speculation schemes, while performing all analysis and testing on real machines. We conclude that even after heavy optimizations, the checkpointing overhead is still considered too high for fine-grain micro architectural level events such as delinquent loads. Lack of sufficiently coarse-grain workload that is suitable for fine-grain speculation in the real-world test application (`MCF`) exacerbates the situation.

4. **Demonstration of the potential for compiler-based checkpointing in providing automated support for backtracking.**

   We show that by exposing simple checkpointing APIs to source-code level, a user-initiated and compiler-driven automatic checkpointing scheme can remove programmers from the overwhelming and tedious details of manual checkpointing. We use `VPR` as the case study to checkpoint its backtracking algorithm in placement phase. We show that our checkpointing scheme is useful in dealing with checkpointing across a large chunk of source code with complex program constructs that is otherwise difficult and error-prone to handle manually. This immediately benefits a programmer who can instead better focus on improving the key algorithm and its supporting data structures—a step that converts enhanced programming efficiency, increased ease of use and reduced develop-run-debug cycle time into improved programmer's productivity.

5. **An LLVM-based infrastructure and implementation of compiler-based checkpointing.**

   To assist and encourage collaborative research, we release a complete source-code package that contains all building-block components that we developed over the course of the thesis. This includes a `LLVM`-based compiler checkpointing transformation and optimization framework, a PIN-based runtime toolset for instrumentation verification, opportunistic exploration, and redundancy analysis, as well as testing benchmark suites (selected `BugBench` programs and our

own micro benchmarks) with various granularity of checkpointing region partitions. We will further investigate the possibility and contribute to integrate our checkpointing work with top-of-tree `LLVM` release. This will make it easier to leverage over our existing contributions made in this thesis and help the research community for potential future exploration and collaboration.

8.2 Conclusions

In this thesis, we design, implement and evaluate a compiler-driven software-only checkpointing infrastructure that operates independently without any hardware support. The checkpointing scheme works on per-store based fine granularity. A user identifies any region with arbitrary size and complexity, a compiler then automates the remaining process to enable and optimize checkpointing on the user-identified region.

Comparing with a recent library-based coarse-grain checkpointing approach (`libCKPT`), our compiler-driven fine-grain checkpointing achieves more than **1000X** improvement on overhead reductions. Even when comparing with a state-of-the-art fine-grian STM solution including Intel's *ICCSTM*, we show significant improvement due to our undo-log checkpointing buffer design and aggressive and effective compiler optimizations. We present full details of the compiler optimizations and show that the optimizations can further reduce up to **98%** of checkpointing buffer size and eliminate up to **95%** of `backup` service routines over our fine-grain baseline checkpointing.

In principle, similar compiler-based optimizations as those that we present in this thesis could be applied to write-buffer based solutions, including Intel's STM compiler/run-time solution. In this respect, our work is a proof of concept that pioneers the exploration that effective compiler optimizations can further reduce STM overhead. Since the STM overhead is currently recognized as the main bottleneck that prevents wide-spread uses of STM, effectively reducing STM overhead will make a big impact in STM-based parallelization of realistic applications.

It may also be possible to design an STM which for sequential portions of transactions switches to an undo-buffer approach, potentially improving STM overhead, wherever such sequential portions are part of long-running transactions.

The techniques presented in this thesis can be further applied in conjunction with any application that needs support for speculative execution, including for the purposes of I/O prefetching, value-prediction, control-flow prediction, and so on. Particularly with the (simple) addition of multiple roll-back points, such applications stand to benefit tremendously from both the efficiency and the simplicity of this

lightweight checkpointing approach.

We leverage support from the checkpointing infrastructure to enable applications to obtain unique features or functionalities. We show that a checkpointing-enabled debugger gains new functionality of reverse execution, with added benefits including unbounded reverse-execution window size, unlimited number of retries, and free from application restarts. We show that a checkpointing-enhanced application (`VPR`) frees programmers from tedious details of conducting manual backtracking and instead allow programmers to better focus on improving the backtracking algorithm and associated data structures. This improvement on rapid-application development process easily converts automatic checkpointing into improved programmer's productivity. Enabling automatic software-only checkpointing on `VPR`'s backtracking algorithm comes with a mild performance overhead of only 15%.

Even with aggressive compiler optimizations, the fine-grain checkpointing overhead may still be considered too high in certain speculative applications. We further show that overlapping execution with delinquent loads generate no performance gains in a DL-centric real-world application (`MCF`), mainly due to lack of suitable workload that are coarse-grain enough to amortize the checkpointing and value-prediction overhead.

8.3 Future Work

The work on compiler-based checkpointing described in this dissertation could be improved and extended in the following ways.

8.3.1 Support for Incremental Checkpointing

One immediate future work is to support and evaluate the potential for incremental checkpointing by allowing multiple rollback points (sub checkpoints) within a single checkpoint region, as well as allowing a user to selectively rollback to one of the multiple sub checkpoints. Under this multiple sub-checkpoint scenario, both of the two original requirements for single-restore point checkpointing still hold: (i) the `start_ckpt` marker must dominate the `stop_ckpt` marker, and (ii) both `start_ckpt` and `stop_ckpt` markers need to be on the same lexical level. The multiple rollback point scenario can be implemented as a sequence of single rollback points, where each individual rollback point needs to satisfy the previously established requirements. Since the rollback action now has multiple potential targets (sub checkpoints), the precise rollback target needs to be identified as additional argument over the `stop_ckpt` API.

The checkpoint buffer scheme proposed in this dissertation needs minimal change to support incremental checkpointing. When utilizing the one-dimensional array buffer, the meta buffer needs a sub-checkpoint counter to identify individual incremental sub-checkpoint, while the data buffer remains unchanged. This minimal design change allows the `backup` operations to proceed as normal other than incrementing the sub-checkpoint counter when reaching a new sub-checkpoint location. This minimal change would allow a checkpoint abort operation to rollback to any prior sub-checkpoint.

8.3.2 Allowing More User Control

An alternative immediate future work is to allow direct and detailed user control over an otherwise checkpoint-enabled program region. During program development of this thesis research, we find multiple cases where a relatively small code block is not suitable for checkpointing (e.g., program initialization phase) within a big checkpoint-enabled program region. A suitable solution will allow user direct control to programmably disable checkpointing within a sub region over a large checkpoint-enabled program region.

Identifying a non-checkpointable section will be similar to the process of delimiting a checkpointing region, except we will need a different pair of region delimiters. The analysis, transformations and optimizations participating in the checkpointing framework will need to make necessary adaptions and adjustments to accommodate this new requirement.

8.3.3 Exploiting More-Precise Pointer Analysis

Our current checkpointing optimization framework is incapable of further optimizing `backup` operations that operate on data stored through pointers, as suggested in section 7.7.4. Most existing optimizations cannot decide whether the `backup` operation into a pointer-based address is a suitable candidate for its optimizing scenario, and thus are forced to make a conservative assumption and not to perform optimizations on such cases.

This places an immediate need for a precise pointer analysis, whose results can be used to disambiguate between the current `backup` address and the address or address range that an existing optimization is interested in. Because the pointer analysis may need to populate across both non-checkpoint region and checkpoint region, initially we suggest a flow-sensitive and context-sensitive analysis. Since the checkpointing optimizations happen within the compilation process, we have relatively more tolerance for longer compile time. Notice that the checkpoint region is often relatively

small and contains known program constructs, we can leverage this knowledge and reduce the complexity of the potential pointer analysis.

Once the pointer analysis results become available, most existing optimizations can immediately benefit from this improved memory disambiguation. E.g., *ArrayOpti* can use the result and decide whether the pointer-based `backup` operation actually access a valid array range, and thus eliminates the original `backup` operation if the array address range covers the `backup` memory address through pointer access. *Hoist* can also leverage this knowledge and decide whether the pointer-based `backup` operation is accessing an address that is loop invariant, and thus can make an informed decision to better optimize `backup` operations within the loop. We may even leverage the pointer-analysis result to invent novel optimizations that are otherwise impossible given the current situation.

8.3.4 Extending Checkpointing to I/O Devices

In future, we will investigate the possibilities and techniques that extend our in-memory checkpointing framework to operate on input-output (`I/O`) devices. Comparing with in-memory operations, `I/O` device operations usually have much longer latency. Micro-architectural level events such as delinquent loads usually have a latency of few dozen to few hundred cycles, the matching speculative workload will ideally have similar latency. Typical `I/O` events (including disk `I/O`, NUMA memory `I/O`, or network `I/O`) often have latency of thousands or even millions of cycles. This much longer latency can ideally be used to tolerate more aggressive coarser-grain workloads, thus have better performance potentials. One way to enable checkpointing over `I/O` devices is to use double buffering—the participating `I/O` device won't consider the current operation complete until receiving a commit command issued from the `I/O` controller.

Appendix A

Checkpointing APIs

```
/*
mark the begin location of a checkpoint region
*/
void start_ckpt();

/*
mark the end location of a checkpoint region
integer conditional code: c
1: commit checkpoint
0: abort checkpoint
*/
void stop_ckpt(int c);

/*
force to commit the current checkpoint
*/
void commit_ckpt();

/*
force to abort the current checkpoint
*/
```

```
void abort_ckpt();

/*
perform backup action, starting from addr location, for the length of len bytes
*/
void backup(char * addr, int len);
```

Appendix B

Special System Handling Functions

Our current checkpointing system supports a total of 10 special system functions. We show the exhaustive list with code skeleton on each supported function. [1]

B.1 Memcpy

```
/* ------------------------------------------------------------------
*/ // void handleMemcpy(char * dst, char * src, int len);
// Compile-time resource to perform backup before a call to memcpy();
// char * memcpy(char * dst, char * src, int len);
/* ------------------------------------------------------------------ */
void handleMemcpy(char * dst, char * src, int len){
  // 1. do memory backup:
  bkp_memory(dst, len);

  // 2. do profile/runtime tracking:
  // ...
}
```

B.2 Memset

```
/* ------------------------------------------------------------------ */
//void handleMemset(char * addr, char val, int rep);
// Compile-time resource to perform backup before a call to memset();
// void memset(char * dst, char val, int len);
/* ------------------------------------------------------------------ */
void handleMemset(char * addr, char val, int len){
```

[1] In order to reduce code verbosity, profiling related codes are not included.

```
  // 1. do memory backup:
  bkp_memory(addr, len);

  // 2. do profile/runtime tracking:
  // ...
}
```

B.3 Memmove

```
/* ------------------------------------------------------------------- */
//void handleMemmove(char *dst, char *src, int len);
// Compile-time resource to perform backup before a call to memmove();
// void memmove(char * dst, char *src, int len);
/* ------------------------------------------------------------------- */
void handleMemmove(char *dst, char *src, int len){
  // 1. do memory backup:
  bkp_memory(dst, len);

  // 2. do profile/runtime tracking:
  // ...
}
```

B.4 Strcpy

```
/* ------------------------------------------------------------------- */
//void handleStrcpy(char * dst, char * src);
// Compile-time resource to perform backup before a call to strcpy();
// char * strcpy(char * dst, char * src);
/* ------------------------------------------------------------------- */
void handleStrcpy(char * dst, char * src){
  // 1. obtain the str length:
  int len = strlen(src);

  // 2. perform the backup, start from dst's address:
  bkp_memory(dst, len);

  // 3. do profile/runtime tracking:
  // ...
}
```

B.5 Strncpy

```
/* ------------------------------------------------------------------- */
//void handleStrncpy(char * dst, char * src, int len);
```

```
// Compile-time resource to perform backup before a call to strncpy();
// char * strncpy(char * dst, char * src, int len);
/* ------------------------------------------------------------------ */
void handleStrncpy(char * dst, char * src, int len){
  // 1. Setup:
  // min_len: the min between len and strlen(src)
  int min_len=0;
  int src_len = strlen(src);
  if (src_len > len){ min_len = len; }
  else{ min_len = src_len; }

  // 2. perform the backup, start from dst's address:
  bkp_memory(dst, min_len);

  // 3. do profile/runtime tracking:
  // ...
}
```

B.6 Strcat

```
/* ------------------------------------------------------------------ */
//void handleStrcat(char * dst, char * src);
// Compile-time resource to perform backup before a call to strcat();
// char * strcat(char * dst, char * src);
/* ------------------------------------------------------------------ */
void handleStrcat(char * dst, char * src){
  // 1. figure out how big the buffer will be used, after the format:
  int src_len = strlen(src);
  int dst_len = strlen(dst);

  // 2. perform the backup, start from dst's ending address:
  bkp_memory(dst + dst_len , src_len);

  // 3. do profile/runtime tracking:
  // ...
}
```

B.7 Strncat

```
/* ------------------------------------------------------------------ */
//void handleStrncat(char * dst, char * src, int len);
// Compile-time resource to perform backup before a call to strncat();
// char * strncat(char * dst, char * src, int len);
/* ------------------------------------------------------------------ */
void handleStrncat(char * dst, char * src, int len){
```

```
  // 1. figure out the actual length of src that will be copied:
  int min_len=0;
  int dst_len = strlen(dst), src_len = strlen(src);
  if (src_len > len){ min_len = len; }
  else{ min_len = src_len; }

  // 2. perform the backup, start from dst's ending address:
  bkp_memory(dst + dst_len, min_len);

  // 3. do profile/runtime tracking:
  // ...
}
```

B.8 Sprintf

```
/* ------------------------------------------------------------------- */
//void handleSprintf(char * buf, char *format);
// Compile-time resource to perform backup before a call to sprintf();
//
// The handling function provided is an approximate to the real string length
// for sprintf(...) with var args.
//
// int sprintf(char * buf, char * fmt, ...);
/* ------------------------------------------------------------------- */
void handleSprintf(char * buf, char *format){
  // 1. figure out the actual size of the buffer used, after the format:
  // USE a new heuristic:
  int len = strlen(buf) * 1.25;
  //int len = (strlen(buf) + strlen(format) ) * 1.25;

  // 2. perform the backup:
  bkp_memory(buf, len);

  // 3. do profile/runtime tracking:
  // ...
}
```

B.9 Vsprintf

```
/* ------------------------------------------------------------------- */
// void handleVsprintf(char * buf, char* fmt);
// Compile-time resource to perform backup before a call to vsprintf();
// int vsprintf (char * str, const char * format, va_list arg );
/* ------------------------------------------------------------------- */
void handleVsprintf(char * buf, char* fmt){
```

```
  // 1. figure out the actual size of the buffer used, after the format:
  // USE a new heuristic:
  //int len = strlen(buf) * 1.25;
  int len = (strlen(buf) + strlen(format) ) * 1.25;

  // 2. perform the backup:
  bkp_memory(buf, len);

  // 3. do profile/runtime tracking:
  // ...
}
```

B.10 Snprintf

```
/* ------------------------------------------------------------------- */
//void handleSnprintf(char * buf, int len, char* fmt);
// Compile-time resource to perform backup before a call to snprintf();
// int snprintf( char *buffer, int buff_size, const char *format, ... );
//
// write into target buffer through snprintf(), at the most, buff_size
// bytes can be written
/* ------------------------------------------------------------------- */
void handleSnprintf(char * buf, int len, char* fmt){
  // 1. figure out how big the buffer will be used, after the format:
  // known from the cmd-line option already

  // 2. perform the backup:
  bkp_memory(buf, len);

  // 3. do profile/runtime tracking:
  // ...
}
```

Bibliography

[1] Using the rdtsc instruction for performance monitoring. In *Pentium II Processor Application Notes, Intel Corporation* (1997).

[2] ADL-TABATABAI, A., LEWIS, B. T., MENON, V. S., MURPHY, B. R., SAHA, B., AND SHPEISMAN, T. Compiler and runtime support for efficient software transactional memory. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)* (2006).

[3] AHO, A. V., SETHI, R., AND ULLMAN, J. D. Compilers: Principles, techniques, and tools, first edition. In *The Red Dragon Book* (1986).

[4] AHO, A. V., AND ULLMAN, J. D. Compilers: Principles, techniques, and tools. In *The Green Dragon Book* (1977).

[5] AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. Checkpoint processing and recovery: An efficient, scalable alternative to reorder buffers. In *IEEE Computer Society* (2003).

[6] ALFRED V. AHO, RAVI SETHI, J. D. U., AND LAM, M. S. Compilers: Principles, techniques, and tools, second edition. In *The Purple Dragon Book* (2006).

[7] AMARASINGHE, S. P., ANDERSON, J. M., LAM, M. S., AND TSENG, C. W. The suif compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (February 1995).

[8] ANANIAN, C., ASANOVIC, K., KUSZMAUL, B., LEISERSON, C., AND LIE, S. Unbounded transactional memory. In *High-Performance Computer Architecture (HPCA)* (2005).

[9] BETZ, V., AND ROSE, J. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications* (1997).

[10] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. A practical automatic polyhedral parallelizer and locality optimizer. In *Programming Language Design and Implementation (PLDI)* (Jun. 2008).

[11] BRONEVETSKY, G., MARQUES, D., PINGALI, K., AND RUGINA, R. Compiler enhanced incremental checkpointing. In *Language and Compilers for Parallel Computing (LCPC)* (2007).

[12] BRONEVETSKY, G., MARQUES, D., PINGALI, K., RUGINA, R., AND MCKE, S. A. Compiler-enhanced incremental checkpointing for openmp application. In *International Conference on Supercomputing* (June 2008).

[13] CALDER, B., REINMAN, G., AND TULLSEN, D. M. Selective value prediction. In *International Symposium on Computer Architecture archive* (1999).

[14] CHOI, S., AND DEITZ, S. Compiler support for automatic checkpointing. In *High Performance Computer Architecture (HPCA)* (2002).

[15] COLLINS, J., WANG, H., TULLSEN, D., HUGES, C., LEE, Y.-F., LAVERY, D., AND SHEN, J. Speculative precomputation: Long-range prefetching of delinquent loads. In *ACM SIGARCH Computer Architecture News* (May 2001).

[16] COLOHAN, C. B., AILAMAKI, A., STEFFAN, J. G., AND MOWRY, T. C. Tolerating dependences between large speculative threads via sub-threads. In *International Symposium on Computer Architecture (ISCA)* (June 2006).

[17] CORPORATION, S. P. E. Spec2000 integer benchmark suites.

[18] DAMRON, P., FEDOROVA, A., LEV, Y., LUCHANGCO, V., MOIR, M., AND NUSSBAUM, D. Hybrid transactional memory. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006).

[19] DING, C., SHEN, X., KELSEY, K., TICE, C., HUANG, R., , AND ZHANG, C. Software behavior oriented parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June. 2007).

[20] ELNOZAHY, W., JOHNSON, D., AND ZWAENEPOEL, W. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems, pp. 39-47* (October 1992).

[21] FELDMAN, S. I., AND BROWN, C. I. Igor: A system for program debugging via reversible execution. In *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging* (1989).

[22] FOUNDATION, F. S. In *GDB: the Gnu Debugger Manual 7.0* (2009).

[23] GROSSER, T., ZHENG, H., ALOOR, R., SIMBRGER, A., GRLINGER, A., AND POUCHET, L.-N. Polly - polyhedral optimization in llvm. In *International Symposium Code Generation and Optimization (CGO)* (March 2011).

[24] H., A., R., D., AND E.:, S. An execution-backtracking approach to debugging. In *IEEE Software, vol. 8, no. 3, pp. 21-26,* (May-June 1991).

[25] H., A., R., D., AND E., S. Debugging with dynamic slicing and backtracking. In *Software: Practice and Experience* (October 2006).

[26] HALL, M. W., ANDERSON, J. M., AMARASINGHE, S. P., MURPHY, B. R., LIAO, S.-W., BUGNION, E., AND LAM, M. S. Maximizing multiprocessor performance with the suif compiler. In *IEEE Computer* (December 1996).

[27] HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data speculation support for a chip multiprocessor. In *ACM SIGOPS Operating Systems* (December 1998).

[28] HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B., DAVIS, J., HERTZBERG, B., PRABHU, M., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional memory coherence and consistency. In *CM SIGARCH Computer Architecture News* (March 2004).

[29] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, W. N. Software transactional memory for dynamic-sized data structures. In *The Twenty-Second Annual Symposium On Principles Of Distributed Computing* (2003).

[30] HERLIHY, M., AND MOSS, J. E. Transactional memory: architectural support for lock-free data structures. In *international symposium on computer architecture (ISCA)* (1993).

[31] HWU, W., AND PATT, Y. Checkpoint repair for out-of-order execution machines. In *Computer Science Division, University of California at Berkley, ACM, 1987* (1987).

[32] JAGADISH, H. V., SILBERSCHATZ, A., AND SUDARSHAN, S. Recovering from main-memory lapses. In *Procs. of the International Conf. on Very Large Databases (VLDB)* (1993).

[33] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *Annual USENIX Technical Conference* (2005).

[34] KINGSLEY, G., BECK, M., , AND PLANK, J. Compiler-assisted checkpoint optimization using suif. In *First SUIF Compiler Workshop* (1995).

[35] KINGSLEY, G., BECK, M., AND PLANK, J. Compiler-assisted checkpoint optimization using suif. In *First SUIF Compiler Workshop* (1995).

[36] KRAWCZUK, V. Distributed debugging based on deterministic re-execution – methology and design of a working prototype. In *Master of Computer Science Thesis* (September 1992).

[37] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO)* (March 2004).

[38] LATTNER, C., AND ADVE, V. The llvm compiler framework and infrastructure tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures* (Sept. 2004).

[39] LI, C., STEWART, E., AND FUCHS, W. Compiler-assisted full checkpointing. In *Software-practice and Experience, Vol 24(10), 871-886* (October 1994).

[40] LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. Value locality and load value prediction. In *ACM SIGOPS Operating Systems Review* (December 1996).

[41] LU, S., LI, Z., Q, F., TAN, L., ZHOU, P., AND Y., Y. Z. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[42] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI 05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM, pp. 190–200.

[43] MCDONALD, A., CHUNG, J., CARLSTROM, B. D., MINH, C. C., CHAFI, H., KOZYRAKIS, C., AND OLUKOTUN, K. Architectural semantics for practical transactional memory. In *ACM SIGARCH Computer Architecture News* (2006).

[44] MINH, C. C., TRAUTMANN, M., CHUNG, J., MCDONALD, A., BRONSON, N., CASPER, J., KOZYRAKIS, C., AND OLUKOTUN, K. An effective hybrid transactional memory system with strong isolation guarantees. In *International Symposium on Computer Architecture (ISCA)* (2007).

[45] MOORE, K., BOBBA, J., MORAVAN, M., HILL, M., AND WOOD, D. Logtm: Log-based transactional memory. In *High-Performance Computer Architecture (HPCA)* (2006).

[46] MOSHOVOS, A., AND KOSTOPOULOS, A. Cost-effective, high-performance giga-scale checkpoint/restore. In *Computer Engineering Group Technical Report* (November 2004).

[47] MOSS, J. E. B. Log-based recovery for nested transactions. In *Proceedings of the 13th International Conference on Very Large Data Bases* (1987).

[48] NG, W., AND CHEN, P. The symmetric improvement of fault tolerance in the rio file cache. In *Proceedings of 1999 Fault Tolerance Computing (FTC)* (1999).

[49] PANAIT, V., SASTURKAR, A., AND WONG, W.-F. Static identification of delinquent loads. In *International Symposium on Code Generation and Optimization* (March 2004).

[50] PLANK, J., BECK, M., AND KINGSLEY, G. Compiler-assisted memory exclusion for fast checkpointing. In *IEEE Technical Committee on Operating System and Application Environments, Special Issue on Fault-Tolerance* (1995).

[51] PLANK, J., AND ELWASIF, W. Experimental assessment of workstation failures and their impact on checkpointing systems. In *28th international conference on fault tolerant Computing* (1998).

[52] PLANK, J., LI, K., , AND PUENING, M. Diskless checkpointing. In *IEEE Transactions on parallel and distributed systems* (Oct. 1998).

[53] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under unix. In *Usenix Winter Technical Conference* (1995).

[54] ROSE, V. B. J., AND MARQUARDT, A. Architecture and cad for deep-submicron fpgas. In *Kluwer Academic Publishers* (2005).

[55] RYCHLIK, B., FAISTL, J., KRUG, B., AND SHEN, J. Efficacy and performance impact of value prediction. In *Parallel Architectures and Compilation Techniques (PACT)* (1998).

[56] S., C. An evaluation of recovery related properties of software faults. In *Ph.D. thesis* (2004).

[57] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., AND AND, C. C. M. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Principles and Practice of Parallel Programming(PPOPP)* (2006).

[58] SAZEIDES, Y., AND SMITH, J. E. The predictability of data values. In *30th International Symposium on Microarchitecture* (1997).

[59] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. A scalable approach to thread-level speculation. In *International Symposium on Computer Architecture (ISCA)* (June 2000).

[60] WANG, K., AND FRANKLIN, M. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (1997).

[61] WANG, Y., HUANG, Y., VO, K., CHUNG, P., AND KINTALA, C. Checkpointing and its applications. In *25th Int. Symp. On Fault-Tol. Comp., pp. 22-31* (June 1995).

[62] WHALEY, J. System checkpointing using reflection and program analysis.

[63] WORK, P., AND NGUYEN, K. Measure code sections using the enhanced timer. In *Intel(R) Software Network* (2008).

[64] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. Retrace: Collecting execution trace with virtual machine deterministic replay. In *3rd Workshop on Modeling, Benchmarking and Simulation, ISCA* (2007).

[65] YEN, L., BOBBA, J., MARTY, M., MOORE, K., VOLOS, H., HILL, M., SWIFT, M., AND WOOD, D. Logtm-se: Decoupling hardware transactional memory from caches. In *High Performance Computer Architecture (HPCA)* (2007).

[66] ZHANG, C., DING, C., GU, X., KELSEY, K., BAI, T., AND FENG, X. Continuous speculative program parallelization in software. In *Principle and Practise of Parallel Programming (PPOPP)* (January 2010).

[67] ZHAO, C., STEFFAN, G., AMZA, C., AND KIELSTRA, A. Tolerating delinquent loads with speculative execution. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA)* (June. 2009).

[68] ZHAO, C., STEFFAN, G., AMZA, C., AND KIELSTRA, A. Compiler support for fine-grain software only checkpointing. In *Compiler Construction 2012 (CC-12)* (March. 2012).

[69] ZHAO, C., STEFFAN, G., AMZA, C., AND WU, Y. Lengthening traces to improve opportunities for dynamic optimization. In *Workshop on Improving Interactions between Compilers and Computer Architecture (Interact), HPCA* (Feb. 2008).

[70] ZHAO, C. C., STEFFAN, G., AND AMZA, C. Compiler-based checkpointing and the potential for tolerating delinquent loads. In *Technical Report, Department of Electrical and Computer Engineering, University of Toronto* (2009).