Deterministic Execution for Arbitrary Multithreaded Programs

Joseph Devietti

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2012

Reading Committee:

Luis Ceze, Chair

Daniel Grossman

Henry Levy

Program Authorized to Offer Degree:

Department of Computer Science and Engineering

University of Washington

Abstract

Deterministic Execution for Arbitrary Multithreaded Programs

Joseph Devietti

Chair of the Supervisory Committee:

Associate Professor Luis Ceze

Computer Science and Engineering

Nondeterminism is one of the main reasons that parallel programming is so difficult. Bugs can vanish when programs are rerun or run under a debugger, thwarting attempts at their removal. Stress-testing is a common practice to flush out rare defects though it consumes extensive processing power and offers no real guarantees of discovering bugs. Deployment can similarly expose new issues that are difficult to reproduce. Finally, nondeterminism frustrates replicating multithreaded programs for fault-tolerance or performance as the replicas can diverge silently. Determinism eliminates these problems, making debugging and replication possible and making testing more valuable and efficient.

Previous efforts at providing determinism required programs to be (re-)written in restrictive languages. In contrast to these language-level determinism schemes, this dissertation shows how execution-level determinism can be provided for arbitrary parallel programs, even programs that contain concurrency errors. First, we employ a hardware-based approach to provide determinism for unmodified binaries running on a deterministic multiprocessor system. Second, we show that memory consistency relaxations both enable a pure software-based implementation of execution-

level determinism for arbitrary programs and also admit a simpler deterministic multiprocessor design. Finally, we describe a hybrid mechanism that integrates execution-level and language-level determinism techniques to provide determinism for arbitrary programs with higher performance than an execution-level approach alone.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Getting a PhD is not a solo sport. Over the past five years I have benefited from the love and support of an amazing team of people. Two pages are not enough to describe the immense impact they have had on my life and on this work, but I shall try.

Let me start by thanking my advisers Luis Ceze and Dan Grossman. Your complementary backgrounds and research perspectives combine to form an environment that I, and many others in our group, have found incredibly fruitful. Luis, thank you for your energy and for always pushing me to achieve my full potential. Dan, thank you for your diligence and clear thinking that has helped me better understand so many of my own ideas. A special thanks is also due to you both for putting up with my incessant dabbling in new technologies from simulators to presentation software to revision control systems, often at the expense of forward progress. Without your guidance I would surely still be polishing my first simulator to a brilliant but useless sheen.

I owe a tremendous debt of gratitude to my parents for supporting me through college and encouraging me to add a second major that turned out to be a defining event in my life. Throughout grad school their unwavering support and advice have helped me grow, as a researcher and as a person. I am also indebted to my sister for her support and for a steady supply of silliness and amazing baked goods. Many a critical deadline push has been fueled by her amazing éclairs.

At Microsoft Research I had the opportunity to work with Karin Strauss and Shaz Qadeer. I am grateful for their insights, patience in working with me to find a successful project, and for continuing our collaboration long after my internship ended. At UW I had the opportunity to work with many great faculty members. I am grateful to Mark Oskin for helping me get started with research at UW and for career advice as I finished. I also deeply appreciated Susan Eggers' encouragement and honesty throughout my time at UW.

During my time at UW, my fellow graduate students in the SAMPA group have made my life technically, intellectually and socially rewarding. I and the other lab members deeply benefited

from Jacob Nelson's friendship, grace under pressure, and ability to order the chaos of shared computing infrastructure. Brandon Lucia brought a creativity and energy to everything he worked on that I wish I could emulate; the many late-night breaks we took from debugging to laser his cat were also very useful. I am grateful for Tom Bergan's rigorous thinking and incredible coding abilities; the scope of our projects would have been much less without Tom's help. Ben Wood was always great fun to work with, though, given that almost none of our collaborations resulted in publications, perhaps too much so. Ben's skills at creative acronyming are second-to-none. In addition to their friendship and humor, I must also recognize Hadi Esmaeilzadeh for bringing real microarchitecture research to our group, Adrian Sampson for an impeccable presentation style and always ensuring the vegetarians had something to eat, and Owen Anderson for his incredible knowledge of LLVM.

My time in grad school wouldn't have been nearly as rich without the amazing friendship of Kayur, Mike, Tomas, John, Yaw and Ben. From early morning workouts to all-day LOTRO to late-night conversations, with an endless stream of nerdy humor throughout, you guys made me feel at home at UW and in Seattle, and kept me from taking grad school too seriously.

And finally, to my wife Sylvia, a few private words addressed to you in public. You have been a constant source of love, vitality and great cooking throughout my degree. Your honesty and openness have indelibly improved my life, even more than your reliable stream of song parodies. You have additionally contributed to my research in many ways, from helping clarify my presentations and visualizations to many parts of this document as well. I feel so lucky to have you as a partner, and I look forward to all the joys of life we'll share together.

DEDICATION

To my parents, for their love, generosity and support throughout my life.
I would never have made it here without you.

Chapter 1 Introduction

Over the last several years, multicore processors have steadily displaced their single-core counterparts: first in server rooms, then in desktops and laptops, and most recently in phones and tablets as well. It is already impossible to buy a new machine with only one core in many market segments. Multicores' swift invasion of the general-purpose processor market has been spurred by the need to meet three simultaneous constraints: high performance, energy efficiency, and limited cooling capacity. Increasing processor frequency has been one of the traditional approaches to improving single-threaded performance, but higher frequencies require higher processor voltages, driving total chip power usage up per Power = Capacitance × Voltage^2 × Frequency. Higher power usage also generates more heat that places more stress on cooling. Ultimately, higher power usage and limited cooling capacity defined a "power wall" that placed a practical limit on CPU frequency, reached in 2006 for desktop and server processors (Figure 1, left axis, blue filled circles).

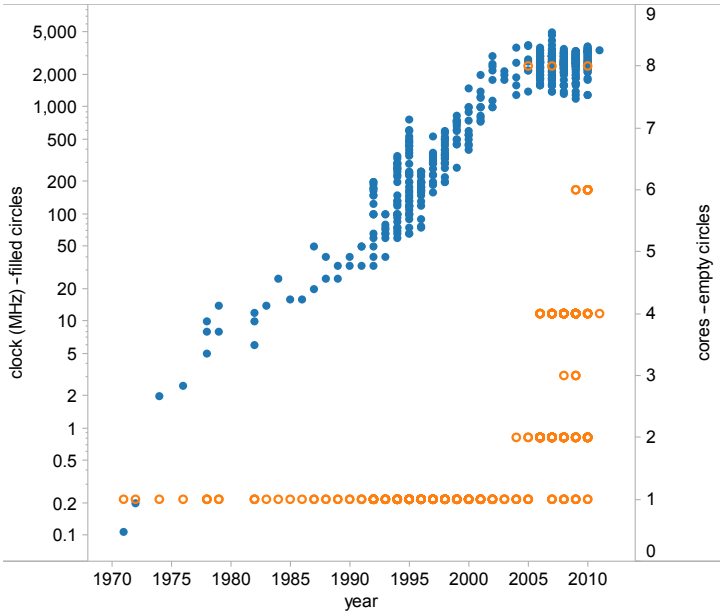

Figure 1: CPU frequency (blue filled circles) and core count (orange empty circles) of production desktop/server microprocessors from 1971-2011. Power supply and cooling limitations imposed a ceiling on frequency improvements in 2006 that sparked the rapid transition to multicore. Data from the Stanford CPUDB project [1].

Multicore processors offered a solution to the power wall, as two cores running at N GHz provide the same aggregate performance as one core running at 2N GHz, but with substantial power and cooling savings over the single-core processor. These efficiencies have driven the popularity of multicore hardware (Figure 1, right axis, orange empty circles). Unfortunately, the spread of

parallel software has not kept pace with that of parallel hardware. Efficient use of now-ubiquitous multicore processors requires that software be parallelized to take advantage of extra computing resources. Manually parallelizing code requires substantial engineering effort, and automatic parallelization remains an open research challenge.

1.1 The Challenges of Parallelism

Like all software, parallel software must satisfy criteria like correctness, performance, usability and cost. Parallelism presents special challenges in terms of performance and correctness since parallelism inherits all of sequential programming's difficulties and adds many new ones. The focus of this dissertation will be on techniques to improve the correctness of parallel programs. However, even if it were trivial to write correct parallel programs, many open challenges would remain in ensuring those programs ran efficiently. Much recent research has examined the difficulties of scheduling [2], automatically tuning [3] and performance debugging [4] parallel applications. The work presented in this dissertation strives to improve correctness while having a minimal negative impact on performance.

Parallelism poses many correctness challenges, from dealing with concurrency errors like data races [5,6], atomicity violations [7] and ordering violations [8], to reasoning about the complexities of memory consistency models [9,10], to coping with the nondeterminism inherent in most parallel systems. Nondeterminism is orthogonal to these other challenges – a deterministic system does not eliminate concurrency bugs, nor does eliminating all concurrency bugs eliminate nondeterminism. Nevertheless, we believe that nondeterminism is a more fundamental challenge because the presence of nondeterminism exacerbates parallelism's other challenges. Nondeterminism destroys reproducibility, introducing a host of issues throughout the software development process. Bugs can appear or disappear from run to run, defeating attempts to understand and remove them. Testing's guarantees are weakened in the presence of nondeterminism, because a passing test says little about future behavior. Deployed software can similarly expose new issues that are difficult to reproduce. Finally, nondeterminism frustrates attempts at replicating multithreaded programs as the replicas can easily diverge, nullifying the reliability or availability benefits of replication.

1.2 The Benefits of Reproducibility and Determinism

Reproducibility does not directly improve the correctness of a parallel program, but it is a key enabler of correctness. Figure 2 shows the benefits reproducibility brings to the process of developing multithreaded software, from debugging through testing and deployment. We distinguish between two flavors of reproducibility: a weak flavor achievable via either determinism or record and replay mechanisms that improves the debugging process, and a strong flavor possible only via determinism (the shaded region in Figure 2) that improves testing and deployed code as well.

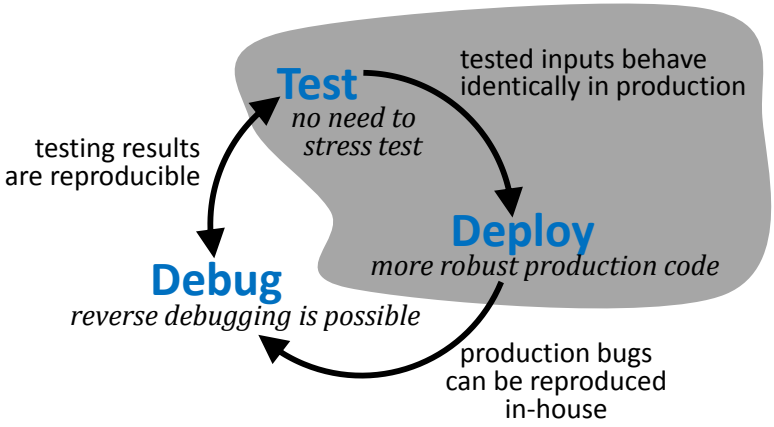

Figure 2: The value of reproducibility throughout the software development cycle. The shaded region indicates benefits attainable *only* through determinism; other benefits can be achieved via determinism or record and replay mechanisms.

Reproducibility is crucial for improving the process of **debugging** parallel software. Nondeterminism is the source of timing-sensitive "heisenbugs" which can disappear when run under a debugger or manifest only infrequently when run natively. Nondeterminism also thwarts any use of reverse debugging [11], a technique that helps to triage bugs. In contrast, deterministic execution allows buggy executions to be replayed, both forwards and backwards, in the same straightforward manner in which single-threaded programs can be.

Testing a multithreaded program often relies on stress testing: running a program with the same input many times to haphazardly exercise a variety of timing conditions, some of which might expose bugs. Determinism eliminates the need to perform stress testing since program execution is guided solely by its input.

Determinism can amplify the power of static analysis tools like model checkers [12] by reducing the size of the state space they must explore. Determinism can also assist dynamic tools (e.g., data race

detectors [13,14]) that focus on verifying a single execution with lower overhead, but suffer acutely from nondeterminism that makes detected errors hard to reproduce and guarantees about future executions impossible to provide. Deterministic execution acts as a powerful lever for dynamic analysis, by providing repeatability, and also ensuring that once a given input has been verified as safe, all future executions of that input will be safe as well. Such a guarantee about future executions can also be leveraged to enable new forms of incremental re-verification in response to code changes.

Code tested and then **deployed** on a deterministic platform has two significant advantages over code running on a nondeterministic platform. First, any tested inputs used in production are guaranteed to have the same behavior in both environments. This allows developers to have high confidence in their software; nondeterminism can no longer introduce a dramatically new set of behaviors once their software is deployed. Second, bugs manifesting on a machine in the field can be reproduced back in a development environment.

Finally, deterministic execution makes it simple to **replicate** multithreaded programs for fault-tolerance: broadcasting inputs to all replicas ensures they remain synchronized.

Weak reproducibility makes reverse debugging possible, and allows bugs that appear during testing or in production to be reproduced. Determinism, however, brings the benefits of reproducibility full circle by 1) eliminating the need to stress test, since a parallel program always behaves the same way for a given input, 2) ensuring that no new behaviors for a given input are possible in production, increasing the assurance provided by testing, ultimately 3) making deployed code more reliable. The value of even weak reproducibility can be seen in the commercial support for tools that make single-threaded executions repeatable, such as VMWare's replay debugger [15] and gdb's recent support for reverse execution [16]. Support for multithreaded execution is less common; one example is Corensic's Jinx tool [17] that offers limited amounts of reproducibility in testing environments. Next-generation tools that offer deterministic execution would be of even greater value.

1.3 Previous Work on Deterministic Parallelism

Prior to our work on execution-level determinism, a language-based approach was required to achieve deterministic parallelism. We briefly outline the most relevant language-based work here, and defer a fuller discussion of other related work in deterministic parallelism to Section 2.5.

Currently-proposed **deterministic languages** favor reduced expressiveness over runtime instrumentation. StreamIt [18] is a deterministic stream processing language that restricts programmers to a non-Turing-complete language without loops in exchange for static guarantees about the buffer sizes needed to implement a stream pipeline. NESL [19] and Deterministic Parallel Java (DPJ) [20] offer a similar bargain in supporting only fork-join parallelism. NESL is a purely functional language, and its lack of mutability avoids issues with interference – when concurrent threads read and write the same data. DPJ adds a sophisticated type-and-effect system to Java to statically prove non-interference for concurrent threads, while still supporting Java's fully mutable semantics. NESL and DPJ's static approaches provide determinism with zero runtime overhead, running programs as efficiently as nondeterministic languages. In addition to performance, deterministic languages provide strong safety guarantees in the form of sequential semantics: because the operations of concurrent threads have been proved non-interfering, parallelism does not affect the program's outcome. Deterministic languages are a safe and performant approach to parallelism for programs that fit within their restrictions.

Other approaches to deterministic parallelism have embraced existing languages but support only restricted **programming models**, such as requiring data race freedom or forbidding the use of complex synchronization mechanisms like locks. Task parallel frameworks like Cilk [21] (see Section 2.2.1) have exactly these restrictions. Writing a data race or resorting to the use of richer synchronization primitives can silently result in nondeterministic execution. Grace [22] is a notable exception: it explores support for a C/C++ nested fork-join programming model with isolation enforced via virtual memory protection. Isolation, plus the lack of synchronization idioms other than fork and join, guarantees the determinism of Grace programs. Concurrently with our initial

work on determinism, Kendo [23] proposed deterministic execution for data-race-free C/C++ programs that use arbitrary synchronization. Kendo exploits data race freedom to avoid the need for expensive thread isolation. Kendo also takes advantage of hardware performance counters to provide very reasonable overheads, at roughly 20% slowdown with 4 threads.

1.4 Dissertation Goals and Contributions

The goal of this dissertation is to evaluate the feasibility and efficiency of techniques that provide determinism for arbitrary parallel programs. Toward this goal we furnish three specific contributions, in the design and evaluation of three deterministic execution systems. The first, DMP or Deterministic MultiProcessor [24,25], is a pure-hardware design that leverages hardware's ability to perform low-overhead speculative execution. The second, RCDC or Relaxed Consistency Deterministic Computer [26], proposes a hybrid hardware-software design for reduced complexity with equivalent performance to DMP. The final system is MELD [27], a pure-software approach for Merging Execution-level and Language-level Determinism that reduces the runtime overhead of execution-level determinism by integrating static, language-based techniques.

Despite the variety of implementation technologies used in these three systems – from hardware transactional memory to type systems – DMP, RCDC and MELD share a common approach to providing determinism (Figure 3). Initially, each thread of a multithreaded program is placed in an *isolated* memory space, wherein an update to shared memory is visible only to the thread that performed the update. Effectively, the original multithreaded program is converted into a collection of single-threaded programs. Single-threaded programs naturally execute in a deterministic fashion; thus the determinism of the entire program is ensured. The mechanism used to ensure thread isolation varies depending on the deterministic execution system, from processor-

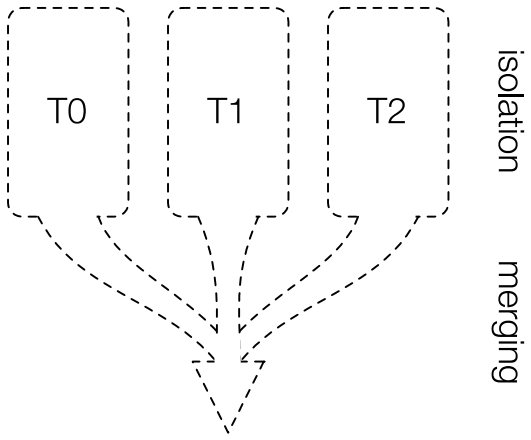

Figure 3: Our basic approach to determinism is precise control of thread communication, with alternating periods of isolated execution and deterministic merging of accumulated outputs.

private hardware caches to runtime instrumentation to static analysis that proves isolation at compile-time.

Of course, threads must periodically *communicate* with one another in order for the program to execute correctly. Thus, inter-thread communication is allowed at regular intervals, subject to two restrictions. First, these communication points must arise at deterministic places in the execution. Second, the communication itself, i.e., *merging* updates that have been performed by remote threads, must be performed deterministically. Again, the mechanisms used to implement deterministic communication differ by system. Counting fixed numbers of machine or bytecode instructions is a persistent approach for identifying deterministic points in an execution. Mechanisms for merging updates deterministically range from hardware techniques for thread-level speculation [28] to order-independent geometry rendering algorithms [29] from computer graphics.

1.4.1 DMP: A Deterministic MultiProcessor

DMP ([24] and Chapter 3) is a novel hardware architecture that runs programs deterministically, even if they use arbitrary synchronization constructs or contain data races. The DMP design built upon existing proposals for hardware transactional memory [30], a vibrant research area that has made the transition into shipping products from IBM [31,32] and Intel [33] (see Section 3.2.3). DMP divides the execution of each thread into a series of transactions and ensures that these transactions commit in a deterministic order. The size of transactions can be adjusted to match the limitations of a hardware transactional memory system, eliminating the need for unbounded transactions. To evaluate DMP, we built an architecture simulator demonstrating that DMP incurs approximately 20% overhead compared to nondeterministic execution.

1.4.2 RCDC: A Relaxed Consistency Deterministic Computer

To bring the benefits of deterministic execution to today's systems, we then designed the CoreDet algorithm [34] for deterministic execution. The main insight behind CoreDet was that DMP's need for always-on speculation could be exchanged for a non-speculative approach coupled with a more

relaxed memory consistency model based on Total Store Order [35,36]. This new design admitted a realizable software implementation of execution-level determinism. The CoreDet deterministic compiler [34] provides determinism for arbitrary C/C++ programs. The CoreDet compiler is open-source and has been downloaded by researchers at over 20 institutions worldwide. Our experiments show that CoreDet exacts a 5x slowdown on average, but brings the benefits of determinism to existing programs running on today's hardware.

To maintain determinism, CoreDet required that synchronization operations be serialized – a clear scalability bottleneck for programs that use frequent synchronization. The RCDC deterministic execution algorithm ([26] and Chapter 4) eliminates this bottleneck via additional consistency relaxations coupled with a more scalable deterministic synchronization algorithm from prior work [23]. RCDC affords a natural hardware-software implementation, leveraging private caches to cheaply enforce thread isolation while leaving software in control of making updates visible. To show the generality of the RCDC approach, we built a hardware simulator to evaluate our hybrid hardware-software design and extended the CoreDet compiler to evaluate a pure-software design as well. Both in simulation and in software, RCDC significantly improves the performance of programs with fine-grained synchronization. Moreover, RCDC's hybrid hardware-software implementation provides fully deterministic execution with performance comparable to that of the original DMP hardware proposal, without the need for always-on speculation.

The source code of our implementations of CoreDet and RCDC, as well as the raw data presented in the RCDC paper, are available from http://sampa.cs.washington.edu.

1.4.3 MELD: Merging Execution-level and Language-level Determinism

MELD leverages both our experience building execution-level determinism systems and the rich and deeply complementary body of work on deterministic parallel languages. On the one hand, execution-level determinism enforces determinism *dynamically* for arbitrary programs but with runtime costs. On the other hand, deterministic languages enforce determinism *statically* without

runtime overhead and with sequential semantics but only support fork-join programs expressible in sophisticated but limiting static type systems.

To combine the generality of dynamic determinism with the performance of static determinism, we designed the MELD system ([27] and Chapter 5) that merges execution-level and language-level determinism in a sound, i.e., determinism-preserving, manner. To maintain generality, MELD sacrifices the sequential semantics guarantee typical of deterministic languages, as general parallel programs have no natural sequential semantics. MELD uses a simple qualifier-based type system for Java to partition a program's data into regions operated on by either execution-level or language-level determinism. MELD also supports a dynamic privatization mechanism to transition data from one region to the other during program execution. To evaluate MELD, we have built a Java-based execution-level determinism compiler and runtime system, and are working on integrating this with the Deterministic Parallel Java [20] language. Our initial results suggest that static determinism can substantially accelerate deterministic execution's performance, while retaining generality for the remainder of the program.

1.5 Reading Guide

Chapter 3 and Chapter 4 are updated versions of the DMP [24] and RCDC [26] conference papers, respectively. These chapters have been revised to include a unified nomenclature, distinguishing between system implementations (DMP, CoreDet and RCDC) and deterministic execution algorithms (DET-*) which are typically implemented both in a hardware and a software system. Table 1 translates from the terminology of the conference papers to that of this dissertation. We also consistently refer to software implementations of deterministic algorithms as CoreDet, though this results in a slight anachronism in Section 3.4.4 as the original DMP paper [24] proposed a compiler-based implementation before the term CoreDet was later introduced in [34]. For completeness, Table 1's shaded cells translate the terms used in the CoreDet paper [34], though the CoreDet system is not discussed at length in this dissertation. Section 3.2.3 includes a new discussion about using upcoming hardware transactional memory support to accelerate the DMP-TM proposal.

Table 1: Terminology used in previous conference papers and in this dissertation.

Paper	Paper Term	Dissertation Term
DMP [24]	DMP	DMP
	DMP-Serial	DET-SERIAL
	DMP-ShTab	DET-SHTAB
	DMP-TM	DET-TM
	DMP-TMFwd	DET-TMFWD
	Hw-DMP	DMP
	Hw-DMP-Serial	DMP-SERIAL
	Hw-DMP-ShTab	DMP-SHTAB
	Hw-DMP-TM	DMP-TM
	Hw-DMP-TMFwd	DMP-TMFWD
	Sw-DMP	COREDET
	Sw-DMP-ShTab	COREDET-SHTAB
RCDC [31]	RCDC	RCDC
	DMP-TSO	DET-TSO
	DMP-HB	DET-HB
	RCDC-DMP-TSO	RCDC-TSO
	RCDC-DMP-HB	RCDC-HB
	CoreDet DMP-TSO	COREDET-TSO
	CoreDet DMP-HB	COREDET-HB
CoreDet [34]	DMP-O	DET-SHTAB
	DMP-TM	DET-TM
	DMP-B	DET-TSO
	DMP-PB	-

Chapter 5 represents a more developed version of the MELD workshop paper [27]. Though the results presented are still preliminary, we envision a subsequent publication to include these and further revisions.

Chapter 2 Related Work

The ideas behind execution-level determinism draw inspiration from a large body of research in making parallelism simpler to express and safer to use. This research ranges from automatically identifying and repairing concurrency defects, to enabling simpler memory consistency models, to providing alternative programming models for parallelism. Directly related to our work on determinism is previous work on managing parallelism implicitly, either via automatically extracting parallelism from sequential programs or via deterministic and implicitly-parallel languages. Also directly related are techniques for record and replay of multithreaded programs, and execution-level determinism systems from other researchers inspired by our systems.

2.1 Concurrency Bug Detection and Survival

There have been many proposals for automatically detecting, and in some cases avoiding, bugs in parallel programs. So-called concurrency bugs come in many different classes with overlapping definitions, as shown in Figure 4. Additionally, some classes of errors, e.g., data races and sequential consistency violations, have precise definitions and detectors can be built that will identify all such violations that arise during an execution. Richer classes of concurrency errors have also been identified, such as atomicity violations [7] and ordering violations [8], that do not have such precise definitions. Atomicity and ordering violations are not axiomatic – they are defined only with respect to some programmer-provided specification that says what operations should be atomic or how operations should be ordered.

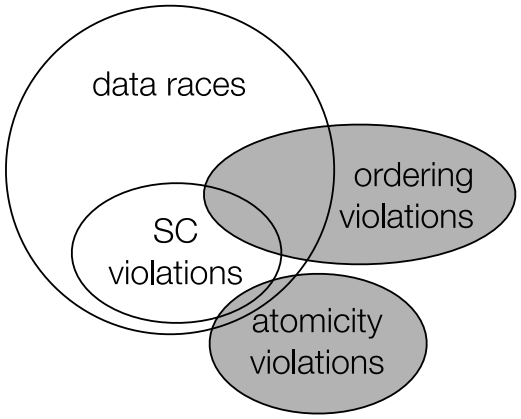

Figure 4: The overlapping definitions of different classes of concurrency errors.

2.1.1 Data Races

Data-race-freedom (DRF) has emerged as a safety property that can be usefully applied to general parallel programs. A variety of schemes have been proposed to verify or enforce DRF with a spectrum of performance/precision trade-offs. The gold standard for enforcing DRF is a race detector that is both sound (misses no races) and complete (reports no false races). The core algorithm for fully precise data race detection is the vector-clock algorithm [5,6,37]. Vector-clock data race detection uses the **happens-before relation** to order events within and between threads in a parallel program. The happens-before relation is a partial order, and thus two events from different threads may or may not be ordered with respect to one another: events that are not ordered are said to be concurrent. For events occurring in a single thread, the happens-before relation includes program order. Figure 5 gives an example of an inter-thread happens-before edge, which arises between release and acquire operations of the same lock (the dashed arrow); the operations on location A are ordered by the happens-before edge (plus program order in each thread) while the operations on location B are not. A data race arises if two memory accesses from different threads are concurrent and access the same memory location in a conflicting way, i.e., at least one of the accesses is a write. In Figure 5, the operations on location B form a data race. Programs that are DRF provide three important properties. First, they execute in a sequentially-consistent manner [9,10]. Second, *synchronization-free regions* of code execute atomically and in isolation [38]. Third, for an access A to a variable V, the value of V cannot change due to remote operations within A's *interference-free region* – the region of code from A's immediately preceding lock acquire to A's immediately subsequent lock release [39,40]. Taken together, these properties make DRF programs much easier to write and to reason about than non-DRF programs.

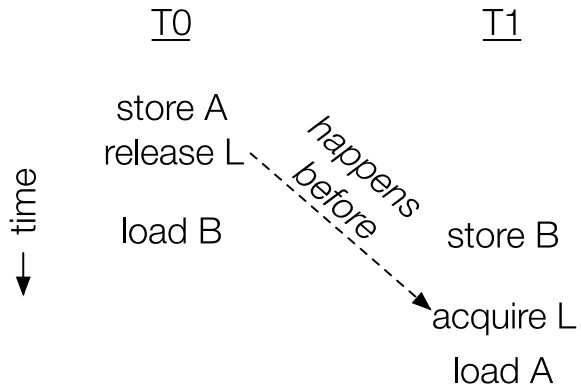

Figure 5: The happens-before relation. The accesses to location B form a data race.

In order to verify DRF, researchers have proposed techniques to detect data races. Many static race detection algorithms have been explored (e.g., [41]) though they are invariably unsound or incomplete due to undecidability. Dynamic race detection algorithms have been proposed as early as 1989 [42], and since then researchers have steadily improved the time and space performance of race detection while exploring the trade-offs of various soundness and completeness guarantees. Eraser [43] proposed **lockset** violation detection, an approximation of race detection which assumes that a consistent set of locks are used to protect each shared variable. Eraser automatically infers the association between locks and variables, and then checks for violations of this inferred locking discipline. As not all programs hold a consistent set of locks when protecting a given variable – e.g., the set of locks can change when a variable is privatized by different threads – Eraser can report spurious races where none exist, though it never misses races. HARD [44] is a hardware-based implementation of the lockset algorithm that uses bloom-filters per cache line to encode which locks should be held when accessing the corresponding data. Goldilocks [14] is a lockset-happens-before hybrid race detection algorithm that uses locksets to accelerate race detection while maintaining precision. Goldilocks uses sound (but incomplete) static race detectors [45,46] to reduce runtime overheads further, and also proposed throwing a language-level exception when a race is detected, an idea that would be subsequently explored in [38,47]. While the lockset algorithm is imprecise, it remains useful because, until recently [13], it held a significant performance advantage over precise forms of race detection.

The latest advance in fully precise race detection is the FastTrack race detector [13], which reduces the per-shared-variable space needs of a traditional vector-clock algorithm from O(n), where n is the number of threads in the program, to O(1) in many cases. FastTrack leverages the observation that programs typically protect all accesses of a location x – both reads and writes – with a single lock. Thus, there is typically only 1 previous reader for a given location, and there is no need to store information for each of n potential readers. For correctness, FastTrack can dynamically transition to a O(n) representation when necessary to support more complicated sharing patterns from, e.g., reader-writer locks. These space savings translate into time savings as well, as race

checks need to examine fewer vector clock entries. The intuition behind FastTrack's optimization is the same as that which motivated lockset race detectors, which assume that programs adhere to a one-lock-per-location invariant and report violations thereof. Lockset race detection was originally proposed as a faster, though incomplete, alternative to vector-clock race detection [43]. Leveraging lockset's key idea allows FastTrack to provide performance comparable to that of lockset-based race detection, without the latter's tendency to detect false races. However, even with FastTrack the cost of fully sound and complete race detection remains high, incurring a roughly 8x runtime overhead.

Other researchers have improved the performance of data race detection via sampling: turning data race detection off for most of the program to avoid its runtime overheads. In theory, a sampling-based approach can miss arbitrary numbers of races, completely invalidating soundness guarantees. In practice, principled approaches to sampling can perform quite well. LiteRace [48] proposes detecting races based on the frequency with which code paths execute: hot code paths are rarely subject to analysis while cold paths are regularly analyzed. Since the race detection analysis is expensive, but run infrequently, overall performance overhead is around 50%. The consequence is that on average only 70% of races are detected. PACER [49] proposes a different sampling strategy that guarantees that races will be detected in proportion to the sampling rate. PACER divides an execution into time windows, and performs race detection within these windows at a specified rate. To uphold the probabilistic detection guarantee, a limited form of race detection is enabled during non-sampled windows, catching all races where one of the racing accesses is inside a sample and the other access is outside. For a 1% sampling rate, PACER adds runtime overhead of around 50%, and PACER's sampling strategy allows space and time overheads, as well as detection probabilities, to scale up and down with the sampling rate.

While DRF is a useful safety property for multithreaded programs, techniques to fully enforce DRF are too expensive for production use. Sampling strategies decrease runtime overhead substantially, at the cost of reducing DRF to a bug-finding heuristic, instead of a safety property a programmer can rely upon.

2.1.2 Atomicity Violations

Atomicity violations were first described in [7], where they were called "high-level data races" and defined as the following situation: two threads access a shared variable, and while the accesses *should have been performed atomically* one of the threads performs a series of non-atomic accesses to the shared variables due to programmer error. Figure 6 illustrates an atomicity violation: the load-increment-store operations on V should occur atomically but due to a lack of synchronization each thread performs them non-atomically. This results in T0's update being lost, so that after both threads attempt to increment V only one increment is recorded.

T0	T1
load r0 ← V	
add r0 ← r0, 1	load r0 ← V
store r0 → V	add r0 ← r0, 1
	store r0 → V

Figure 6: An assembly code example of a single-variable atomicity violation. The lack of atomicity causes T0's increment to be dropped.

One important property of atomicity violations is that they violate an *atomicity specification* that states which accesses need to be performed atomically. This atomicity specification is, unfortunately, often unavailable and is only implicitly represented by the program's code. While there is no way to know for certain what accesses should be performed atomically in the absence of the actual specification, a number of useful heuristics have been identified. These heuristics inevitably entail both false positives, when an atomicity violation is detected but the program is in fact abiding by its (missing) atomicity specification, and false negatives, when the program's (missing) atomicity specification is deviated from but no atomicity violation is detected.

[7] also generalizes the definition of atomicity violations to accesses of a set of shared variables. This has resulted in two sub-classes of atomicity violations: *single-variable* atomicity violations and *multi-variable* atomicity violations. Figure 6 shows an example of a single-variable atomicity violation involving just the shared variable V. A multi-variable atomicity violation arises when accesses to a *set* of variables should be performed atomically, e.g., when updating the real and imaginary components of a complex number, but due to a programming error the variables are not updated atomically. Detecting multi-variable atomicity violations is inherently more complicated as it requires grouping variables into meaningful sets whose atomicity can then be examined. This

grouping process is again driven by heuristics and can be another source of both false positive and false negative atomicity violation reports.

Several atomicity violation detection schemes sidestep false positive/false negative issues by requiring the programmer to provide an atomicity specification. Atomizer [50] requires such a specification, and then dynamically verifies that the specification is adhered to during execution. Atomizer's dynamic analysis is based on lockset race detection, and lockset's intrinsic false positives make Atomizer's atomicity analysis imprecise as well. Velodrome [51] improves upon Atomizer by providing sound and complete atomicity violation detection, again with respect to a programmer-provided specification. In the absence of a provided atomicity specification, e.g., when dealing with legacy programs, both Atomizer and Velodrome evaluate the implicit specification that assumes all functions should execute atomically. While clearly leading to false positives, both schemes report no more than a few dozen warnings per program, making these heuristics tractable.

The Serializability Violation Detector [52] proposed a more sophisticated heuristic for discovering atomicity violations, effective enough to be used in the absence of an actual atomicity specification. Starting from a read access A to a shared variable, SVD infers an atomic region for subsequent accesses that are data- or control-dependent on A. SVD can thus discover some forms of single- and multi-variable atomicity violations automatically. AVIO [53] generalizes the SVD result, proposing a comprehensive framework for single-variable atomicity violations. Using this framework, AVIO uses training runs to infer the atomicity specification of the program: pairs of accesses that are atomic during training runs, but non-atomic during production runs, are likely atomicity violations. AVIO is able to automatically identify four real atomicity violation bugs in Apace and MySQL, with only five false positive reports on average for each program. The AtomAid system [54] automatically repairs atomicity violations by coupling AVIO's techniques for identifying single-variable atomicity violations with the BulkSC architecture [55]. BulkSC groups the dynamic instruction stream into chunks that execute atomically, as a low-overhead way of providing sequential consistency (Section 2.1.4 below). AtomAid adjusts chunk boundaries such that pairs of

accesses representing likely atomicity violations are placed in the same chunk, and, since chunks execute atomically, the atomicity violation will be repaired.

CTrigger [56] is a system that increases the manifestation probability of atomicity violations, making bug detection and other testing tools more effective. CTrigger collects a set of potential unserializable access interleavings (based on the AVIO [53] framework) and uses dynamic analysis to prune away those interleavings that are unreachable due to the program's synchronization. The remaining interleavings are ranked by their likelihood of being an actual bug, and small delays are inserted into the program's execution to make the unserializable interleavings more likely to occur. Using CTrigger makes atomicity violation bugs manifest orders of magnitude more quickly than with regular stress testing.

As researchers made steady progress finding and fixing single-variable atomicity violations, attention turned toward the more challenging problem of multi-variable atomicity violations. While SVD detected some kinds of multi-variable atomicity violations with its data- and control-dependence analysis, MUVI [57] pioneered a more general analysis that learns correlations among sets of variables from a series of training runs, and uses these correlations to automatically detect multi-variable atomicity violations. MUVI was able to identify five real-world multi-variable atomicity violations and discovered four new such bugs in Firefox. ColorSafe [58] is another proposal for multi-variable atomicity bug detection and survival. ColorSafe uses memory allocation information to correlate variables, and then uses this information to identify and also proactively avoid likely multi-variable atomicity violations.

2.1.3 General Concurrency Bug Detection

More recent work on concurrency bug detection has focused on new classes of concurrency bugs such as *order violations* [8] and on building concurrency bug detectors that work for many classes of errors. Order violations arise when a program relies on two events, e.g., two critical sections, always occurring in a particular order when the events can occur in either order. Order violations are distinct from both data races and atomicity violations. Unlike data races, the two events in an

order violation may be individually well-synchronized. Unlike atomicity violations, which are inherently unordered, order violations arise due to an expected (but unenforced) ordering. Order violations represent a detection challenge as the presence of an order violation depends on the (often missing) ordering specification of a program.

To cope with rich classes of errors like atomicity and order violations, many researchers have focused on properties common to all concurrency bugs and started building general concurrency bug detectors. These tools focus on detecting problematic patterns among communicating load and store instructions. For example, Bugaboo [59] records the context surrounding inter-thread communication events and uses the context to help differentiate correct from buggy communication patterns. The Interleaving-Constrained Multiprocessor [60] uses testing runs to learn ordering invariants among instructions. During production runs these invariants are enforced by the processor, automatically avoiding problematic interleavings. DefUse [61] learns invariants about which definition each use of a variable should read from, and reports violations of these invariants as errors. The errors uncovered by DefUse can be indicative of concurrency bugs and also sequential bugs like memory errors. Recon [62] focuses on making concurrency bug detection more useful to programmers by automatically identifying the root cause of a bug. Recon's bug reconstructions demonstrate to the programmer not just that a likely error occurred, but also how the program entered the problematic state.

2.1.4 Simplifying Memory Consistency Models

The memory consistency models of modern languages and architectures [9,10,63] are notoriously complex. Despite years of scrutiny, subtle errors have been found in both the Java Memory Model and its JVM implementations [64]. Even assuming the correctness of the model and its implementation, the sequentially consistent (SC) semantics [65] offered by these languages for DRF programs is often void in practice as the DRF precondition is not verified (see Section 2.1.1). Thus, programmers cannot rely on SC behavior. Furthermore, others have noted that DRF implies much stronger properties than SC, e.g., interference-freedom [40]. The resulting "gap" between SC and DRF can be exploited to provide SC much more cheaply than DRF. Motivated by the high overheads

of fully sound and complete race detection, researchers have proposed variations on race detection that enforce SC or detect SC violations. These SC-based schemes have proven to be substantially more performant than full data race detection.

Many have proposed using speculation to increase the scope for memory reordering [66–70]. We focus on contemporary approaches, which support relaxing all instruction reorderings across a wider window than previous work. BulkSC [55] and ASO [71] are pioneering architectures that provide high-performance sequential consistency instead of the weaker consistency models common in modern multiprocessors. BulkSC extends the Bulk [72] architecture to provide sequential consistency. The original Bulk architecture offered a mechanism to atomically execute chunks of thousands of instructions, using Bloom filter-based [73] signatures to efficiently encode read and write sets. Bulk's chunk mechanism was used to implement both transactional memory (also called TM, see Section 2.2.2) and thread-level speculation (also called TLS, see Section 2.3) in a unified way. Building on Bulk, BulkSC groups a program's entire dynamic instruction stream, instead of just TM or TLS accesses, into chunks. Speculative execution of each chunk allows processors to execute chunks in parallel for high performance, and the ability to aggressively reorder instructions within a chunk eliminates many potential consistency-related processor stalls. Pipelined chunk execution reduces the latency of cross-chunk consistency stalls as well so that BulkSC incurs negligible overhead despite providing strong memory ordering. The Atomic Sequence Ordering (ASO) processor [71] also performs memory ordering speculation at coarse granularity, using a scalable FIFO store buffer to guarantee memory ordering and to enable speculation across a large window of instructions. Much like BulkSC, ASO breaks up execution into a series of atomic sequences and guarantees memory ordering only across, not within, sequences. Speculative state is buffered in the L1 cache, avoiding the need for associative search during store-load forwarding. Pristine store values are held in the FIFO store buffer and are accessed only during infrequent commit operations or rarer abort operations. Similarly to BulkSC, ASO provides SC yet outperforms a conventional relaxed-consistency RMO [35] machine. Finally, InvisiFence [74] provides high-performance sequential consistency in the context of a conventional multiprocessor,

unlike BulkSC's dependence on the Bulk architecture, and without the need for a large FIFO store buffer as in ASO. InvisiFence uses the L1 cache for speculative state, and the L2 cache for pristine state, and ensures fast, atomic commit/abort operations via flash-clearing speculative/valid bits, respectively, in the L1 cache. As with BulkSC and ASO, InvisiFence provides SC memory ordering with better performance than a conventional RMO machine.

Subsequent work has looked at exporting sequential consistency farther up the programming stack. Conflict Exceptions [38] verifies the atomicity and isolation of all synchronization-free regions, a weaker property than DRF but one that can be enforced more cheaply using a simplified version of unbounded hardware transactional memory. Conflict Exceptions guarantees both SC and that all synchronization-free regions execute serializably; otherwise an eponymous "conflict exception" is thrown that terminates the program.

DRFx [47,75] weakens Conflict Exceptions' guarantee in exchange for reduced hardware requirements and better performance. DRFx verifies atomicity and isolation of short compiler-defined regions that are guaranteed not to exceed the resources of a best-effort hardware TM system. Similarly to Conflict Exceptions, DRFx verifies that these regions execute serializably or raises a runtime exception. By allowing only intra-region compiler reorderings, an exception-free DRFx execution is guaranteed to be sequentially consistent. However, the boundaries of these smaller regions are completely governed by the compiler, providing no source-level atomicity guarantees to the programmer.

The primary use of the sequential-consistency-else-exception guarantees of Conflict Exceptions and DRFx is simplifying the memory models of modern languages like Java [9] and C++ [10]. In the absence of these hardware mechanisms, a sequential consistency guarantee is obtainable only for programs that satisfy the stringent DRF requirement, and in conventional systems there is no checking of whether programs have in fact satisfied it. The stronger atomicity and isolation guarantees of Conflict Exceptions may be useful to compiler writers or even application programmers, though the performance implications of the need to support unbounded

synchronization-free regions have not been clearly evaluated. A recent in-depth performance evaluation of the DRFx architecture [75] found that its runtime overheads were quite modest – less than 25% for a simulated 16-core machine – showing that supporting SC throughout the execution stack can be done affordably.

Other work on memory consistency has focused on extracting maximum performance from weak consistency models. The Conditional Memory Ordering (CMO) system [76], which observes that the fence semantics offered by modern processors are frequently stronger than is needed to correctly implement language memory models like the JMM. CMO elides memory fences at runtime when they are dynamically unnecessary, and a pure-software implementation of CMO improved the performance of many Java applications by 5-10%. In a similar vein, [39] exploits the full flexibility of the C++ memory model [10] to expand the scope for compiler optimizations across lock acquires and releases (see Section 2.1.1). The extra flexibility allows the compiler to elide many memory instructions that a conventional compiler cannot, and though the resulting performance impact is negligible many optimization opportunities remain unexplored. The RCDC system (Chapter 4) takes similar advantage of language memory models' relaxations to reduce the cost of deterministic memory ordering, which is even more expensive than its nondeterministic counterpart. RCDC's consistency optimizations in particular are modeled heavily on CMO's design.

2.2 Programming Models for Parallelism

Parallel programming models have a long history both in theory and in practice, dating back nearly as long as computing itself. This section examines select recent developments in parallel programming models that are most relevant for understanding the deterministic systems described in Chapters 3-5.

2.2.1 Task Parallel Frameworks

Tasks are an increasingly popular programming abstraction for parallel architectures. Many major hardware and software vendors have produced task parallelism frameworks, from Intel's Cilk [21] and Threading Building Blocks [77] to Apple's Grand Central Dispatch [78], Microsoft's Task

Parallel Library [79] and Java's ForkJoin framework [80]. These task parallel frameworks abstract away notions of processors and threads, requiring programmers to think only in terms of tasks and the dependences between tasks. Task dependences are often specified via control flow, instead of via locks and shared memory, resulting in a simplified programming model. The runtime system manages the scheduling of tasks onto actual processing resources, honoring dependences and preventing the hardware from being over- or underutilized.

Task systems answer many of parallel computing's performance challenges, but their correctness benefits are less clear. Data races are still possible if a programmer specifies task dependences incorrectly. The Cell Superscalar (CellSs) task system [81] offers the potential for increased safety by requiring that inter-task communication occur only via function arguments, instead of shared memory. Such a limited interface is likely amenable to automatic dependence verification. Many task systems additionally have very strong repeatability guarantees, providing sequential semantics for race-free programs [21,81]. That is, if a program uses data-race-free task parallelism, it can be debugged and replayed in exactly the same manner as a single-threaded program. However, data races can introduce nondeterminism that voids this guarantee [82]. Overall, current task systems focus on providing performance portability and provide no significant safety benefits over parallel programming with general-purpose languages.

Tasks are also the primary parallelism construct in GPU programming models such as CUDA [83], OpenCL [84] and DirectCompute [85]. Historically, GPUs were programmed for computer graphics in a fork-join style where communication between concurrently-executing tasks was impossible to express in the programming model. More recently, GPUs have begun supporting atomic operations that allow communication between concurrent tasks. These atomic operations can be used to build mutexes, allowing arbitrary parallelism constructs to be expressed [86]. As GPUs adopt more of CPUs' generality, they also inherit the latter's programmability challenges. GPU architecture researchers have adopted many CPU techniques to cope with these challenges, such as race detectors [87,88] and transactional memory [89] (see Section 2.2.2 below).

2.2.2 Transactional Memory

Transactional memory (TM) is similar in spirit to, and borrows its name from, database transactions [90], though TM was first proposed as a hardware accelerator for lock-free data structures [30]. In the original TM proposal, a processor speculatively executes a *transaction* – a series of updates to shared memory that execute atomically and in isolation. The updates are buffered in the processor's L1 cache and so are invisible to remote processors. Cache coherence is used to detect conflicting operations by remote processors. If the local transaction completes without any conflicts, the operation is committed to memory by allowing the buffered cache lines to become globally visible. If a conflict is encountered, the local transaction can be rolled back and the original data fetched again from elsewhere in the memory hierarchy. Transactions, as first proposed, provide an elegant and performant mechanism for implementing multiword CAS operations which simplify many lock-free algorithms.

After this initial hardware TM (or HTM) proposal, the idea of using transactions more broadly as a synchronization mechanism gained tremendous momentum [91]. Transactions offer a compelling alternative to lock-based synchronization, as transactions are 1) a simple, declarative synchronization primitive, 2) amenable to a highly-concurrent implementation whose complexity can be hidden from the programmer, and 3) inherently deadlock-free.[1] Researchers have investigated many aspects of TM, such as support for transactions that overflow the resources of the L1 cache [92], the semantics of transactions and non-transactional code [93], different buffering and conflict detection strategies [94], partial transaction commit to avoid wasted work [95], partial re-execution to avoid the need to perform rollback on conflicts [96], and hybrid hardware-software support for transactions [97]. HTM techniques have also been repurposed to accelerate lock-based critical sections by speculatively eliding lock acquisition [70], to sandbox speculative compiler optimizations [98], and to protect against atomicity violations [99].

[1] Application-level progress is not guaranteed, however, as conflicting transactions may continually trigger rollbacks.

Many software TM (or STM) systems have been built as well, in academia [100,101] and also by Microsoft [102], Intel [103], and the GCC team [104]. STM systems have brought to the surface many real-world implementation issues such as interoperating with legacy, lock-based code and handling I/O operations that cannot be rolled back [105]. HTM support was also integrated into some production microprocessors, starting with Azul's formerly-available Vega 3 chip [106] that offered a form of speculative lock elision [70]. Today, IBM's BlueGene/Q processor [32] and zEnterprise EC12 mainframe [31] offer HTM support, and Intel will integrate TM support into the x86 ISA starting with the 2013 Haswell architecture [33].

Transactional memory plays an important role in the design of the Deterministic MultiProcessor architecture (Chapter 3). DMP divides the execution of an arbitrary parallel program into a series of transactions. Building on TM's serializable execution semantics, DMP then ensures that the transactions logically execute in a fixed serial order. The underlying TM facility ensures that transactions execute in parallel to recover performance.

2.3 Thread-level Speculation

Thread-level speculation (TLS) [28] is a set of techniques to extract parallelism from legacy sequential programs automatically. TLS was originally proposed as a hardware design analogous to out-of-order execution, but at coarser granularity: entire loop iterations or function calls are speculatively executed out-of-order on multiple CPU cores. Sophisticated hardware tracks memory dependences to ensure equivalence to the sequential semantics of the original program.

Knight [107] first proposed hardware support for speculative parallelization of LISP programs. The mostly-functional nature of LISP dramatically simplifies dependence tracking between tasks. Later, Multiscalar [28] proposed the first recognizable TLS system, which extracted parallelism from sequential, imperative code running on a conventional microprocessor. Multiscalar inspired a number of subsequent proposals [108–112]. While the initial TLS proposals required hardware support, they inspired many software-only approaches as well. Speculation and dependence tracking are implemented in software, typically via compiler instrumentation. However, despite a

large amount of research, practical TLS techniques to extract parallelism from sequential programs remain elusive. Recently, impressive results have been obtained from using sophisticated profile-driven compilation and programmer-provided annotations to parallelize sequential integer codes across multiple cores [113] and even nodes in a cluster [114]. This semi-automated parallelization approach seems a promising path going forward.

TLS techniques have inspired many of the techniques used in the DMP system (Chapter 3), though to an opposite end. While TLS seeks to parallelize a sequential program, DMP imposes determinism by serializing a (nondeterministic) parallel program to obtain determinism. DMP exploits the TLS observation that the execution of a program can be parallel even if serial semantics are required. Starting from a parallel program, as DMP does, allows speculative parallelization to be more effective than it has been in the context of sequential programs.

2.4 Multithreaded Record and Replay

Record and replay systems can be used to replay an execution for debugging or forensic purposes. Record and replay systems have strong repeatability properties but have not as yet been extended to improve safety. The central design trade-off in record and replay systems is between the amount of execution information logged (which impacts runtime overhead and log size) and the extent to which an execution can be replayed. In particular, we divide record and replay schemes into those that can replay executions with data races and those that cannot. There has also been steady progress in lossless recording techniques, mostly in hardware but recently in software-based systems as well, that has reduced time and space overheads without sacrificing repeatability.

2.4.1 Hardware Record and Replay

Logging the values returned by each memory read is sufficient to replay any multithreaded execution [115,116], but incurs high time and space overhead. Recording just the order of accesses to a global shared cache [117], with the help of hardware support, also suffices. A better approach, however, is to record just the points of communication between threads; Netzer [118] showed how to transitively reduce this communication to produce an optimally small log.

A number of architecture proposals have built on and extended Netzer's transitive-reduction idea. The Flight Data Recorder (FDR) system [119] tags cache lines with the instruction count of their last accessor processor. By transferring these counters on coherence events the inter-processor communication graph can be built. FDR uses transitivity to avoid logging redundant edges in this graph, though since accessor information in restricted to the cache, cache misses must be conservatively logged. BugNet [120] proposes record and replay for user-space events only, using processor-local logs for the values returned by all loads (as in the very first record and replay systems) and an FDR-like mechanism for recording synchronization events between processors. The Regulated Transitive Reduction (RTR) system [121] improves the communication graph recording of FDR by hallucinating inter-processor edges that increase the effectiveness of Netzer's transitive reduction beyond what is possible with the program's true edges alone. RTR is also the first record and replay scheme to support non-sequentially-consistent hardware, by recording the values of loads that potentially violate SC. Overall, RTR reduces log sizes by about 20x compared to FDR. Concurrently proposed with RTR, Strata [122] divides an execution into periods free of inter-processor communication, each of which is called a "stratum".[2] Strata simply logs the number of memory references performed by each processor during a stratum, requiring much simpler hardware than FDR or RTR. Since multithreaded programs do not communicate constantly, an individual stratum can cover many thousands of instructions, and log sizes are about 5x smaller than FDR.

ReRun [123] and DeLorean [124] both exploit Strata's insight about communication-free periods of execution. ReRun [123] achieves log sizes on par with RTR and hardware requirements on par with Strata by tracking communication-free periods (called "episodes") on a per-processor basis instead of on a global basis. Tracking per-processor episodes results in longer episodes and yields more scalable hardware. Logging 1) the size of each episode (in number of memory references performed within it) and 2) a Lamport timestamp [37] ordering each episode with those of other processors are sufficient to subsequently reproduce an execution. Rerun uses small hardware signatures [72]

[2] NB: We use the term *Strata* (capital S) for the research proposal, and the term *stratum* (lower-case s) for the region of execution logged in the Strata proposal.

to discover inter-processor communication, instead of FDR/RTR's per-cache line accessor information. While ReRun ends an episode whenever communication occurs between concurrent episodes, Timetraveler [125] observes that only *cyclic* communication between episodes requires an episode boundary. If the communication is entirely from episode A (on P0) to episode B (on P1), even if there are multiple communicating loads and stores in each episode, the execution can be reproduced by ordering episode A before B during replay. If communication from episode B to episode A occurs, an episode boundary must be inserted to maintain an acyclic episode graph. Timetraveler allows for longer episodes which result in log sizes 90% smaller than ReRun's. Karma [126] also extends ReRun to support parallel replay. Karma replaces ReRun's Lamport timestamps with a DAG encoding that identifies which processors' episodes immediately precede and succeed the current processor's episode. During replay, a processor may replay an episode as soon as all of its predecessor episodes have replayed; parallelism in the DAG accelerates the replay process to within 30% of recording speed, unlike ReRun's completely sequential replay.

DeLorean [124] divides execution into "chunks" of instructions that are executed atomically and in isolation. A chunk from each processor updates shared state via a commit process, potentially triggering re-execution if two chunks' updates conflict. By constraining execution to commit chunks in a fixed order and to mostly produce chunks of constant size (though overflowing cache capacity may nondeterministically cause smaller chunks to form), DeLorean reduces logging requirements by two orders of magnitude over previous proposals.

More recently, researchers have proposed record-and-replay designs that leverage a global clock for smaller log sizes and simpler hardware designs. LReplay [127] uses the global clock to divide execution into fixed periods of time, e.g., 1024 cycles, without needing to assign a logical timestamp to each period. Communicating memory operations that execute in non-concurrent periods are naturally ordered by the global clock, while those that execute in concurrent periods are recorded in a separate log. LReplay handles consistency models as relaxed as TSO by detecting potential SC violations and logging them. Relying on a global clock allows the LReplay design to avoid modifying the caches or cache coherence protocol, which is a common feature of previous record-and-replay

approaches. CoreRacer [128] adopts a similar approach to LReplay, but in the context of an Intel multicore design. CoreRacer has a similar minimally-invasive design based around Intel's TimeStamp Counter (TSC) register, a synchronous global clock visible to all cores that was introduced in the Nehalem architecture.

The idea of modifying execution to reduce logging requirements, as used by RTR and DeLorean, is a direct influence on the design of DMP. DMP takes this idea to its logical conclusion: using complete control over the execution to replace logging with a pre-determined scheduling policy. Deterministic execution shows that logging is not intrinsically necessary to provide reproducibility. However, recent record-and-replay schemes [127,128] maintain an advantage over deterministic execution in that they require less invasive hardware support.

2.4.2 Software Record and Replay Techniques

Recent advances in pure-software record and replay systems provide acceptable runtime overheads without sacrificing replayability in the presence of data races. The LEAP [129] system provides full replayability by using the log-all-reads idea [116] in conjunction with static analysis to prune the amount of logging required. LEAP's overheads range from 10% to 7x on a range of Java benchmarks. DoublePlay [130] is a pure-software scheme inspired by the idea of constrained execution first espoused in hardware proposals like RTR and DeLorean. DoublePlay uses multiple executions of the same program to parallelize recording. One version of the program "runs ahead" at nearly full speed, logging only synchronization operations, and separating the dynamic execution into windows called "epochs." Each epoch is then re-executed on a single processor (so that only scheduling decisions need be logged) guided by the synchronization log. If epoch execution diverges from the run-ahead process' state, execution is rolled back and restarted from the end of the problematic epoch. Otherwise, sufficient information has been recorded to replay that epoch precisely. DoublePlay achieves low recording overhead – typically less than 50% – by speculating that epochs will not diverge under re-execution and parallelizing re-execution across extra cores.

Many software-only record and replay systems sacrifice replayability for performance. RecPlay [131] was among the first to do this, logging only synchronization operations, which is sufficient to replay execution up until the first data race. PRES [132] and ODR [133] extend this idea by recording a configurable subset of the information required for full replay, e.g., system calls, synchronization, or function calls. During replay, these schemes use the recorded log and a data-race detector to search for a desired execution that, e.g., triggers a bug. PRES and ODR incur runtime overheads similar to those of DoublePlay, but without using extra compute resources during recording. However, these schemes sometimes require thousands of replay attempts to reproduce an execution.

2.5 Deterministic Parallelism

Deterministic parallelism systems deliver the benefits of repeatability without any logging requirements by eliminating nondeterministic interactions among threads. Determinism's constrained execution is the key to bringing the benefits of repeatability to the entire software development cycle (Figure 2, page 3).

2.5.1 Determinism Verification

There have been several proposals to verify the determinism of a computation at runtime. Sadowski et al. [134] showed with SingleTrack that determinism can be thought of as a stronger safety property than race detection, defining nondeterminism as a "race" on a lock object. Due to its expanded notion of a data race, SingleTrack's evaluation was limited to programs that use deterministic synchronization primitives, though "race-free" locking is permitted in principle. SingleTrack has the same overhead as vector clock race detection, but provides repeatability by raising an exception at the first sign of nondeterminism.

Burnim et al. [135] advocate using programmer-specified determinism specifications to verify executions as "semantically deterministic." The DETERMIN system [136] can automatically infer these specifications, though such high-level notions of correctness will likely require human verification. Semantic determinism allows insignificant low-level details (e.g., the order of elements

in a set implemented via a linked-list) to be glossed over in favor of verifying the determinism of a higher-level property (e.g., the contents of the set). However, semantic determinism offers no real safety guarantees, as it permits data races and non-sequentially-consistent behavior, and offers no repeatability when determinism specifications are violated.

2.5.2 Deterministic Parallel Languages

Deterministic parallel languages trade reduced expressiveness in exchange for strong safety and programmability guarantees. Earlier languages built on functional languages [19] or required runtime checking to support imperative language features [137], but more recent efforts [20] have employed rich type systems capable of statically verifying determinism even in the presence of mutability.

Jade [137] is an implicitly parallel language based on C. A Jade programmer decomposes a serial C program into a nested task hierarchy, each task having a programmer-specified access specification that states what data structures the task will read or write. Using the access specification, the Jade runtime will opportunistically execute non-interfering tasks in parallel, preserving the sequential semantics of the program while attaining greater performance. Tasks that violate their access specification trigger runtime errors.

Similar to Jade, Prometheus [138] is an extension to C++ that dynamically tracks object ownership and uses ownership to collect operations on each object into *serialization sets*. The operations on a given serialization set are performed serially, ensuring determinism while allowing parallel operations on distinct objects. Programmers are responsible for partitioning objects into serialization sets and incorrect specifications, such as putting the same object into two distinct sets, trigger a runtime error.

StreamIt [18] is a deterministic stream processing language that restricts programmers to a loop-free language to define the nodes in a stream processing graph. Determinism is enforced by allowing communication only along the edges of the stream graph. The language restrictions allow the StreamIt compiler to statically size the buffers needed between nodes in the stream graph.

NESL [19] is a purely functional language with support for nested data parallelism. NESL's functional features ensure determinism in the face of parallelism without the need for runtime checks. NESL has also heavily influenced the design of Data Parallel Haskell [139], which implements many of NESL's language constructs in the lazy, purely functional language Haskell.

Deterministic Parallel Java (DPJ) [20] is a set of extensions to Java to provide statically-checked deterministic fork-join parallelism in the context of a modern object-oriented language. DPJ adds a sophisticated type-and-effect system to Java to statically prove non-interference among parallel tasks. DPJ programs incur zero runtime overhead, running as efficiently as nondeterministic code. A subsequent extension to DPJ has added support for interfering parallel tasks, using a STM system to ensure serializable, but potentially nondeterministic, execution [140]. Other extensions have added support for effect inference [141] and more expressive parallelism such as pipelines [142] and non-nested parallel tasks [143]. DPJ, a core component of the MELD system, is described more fully in Section 5.2.

Besides performance, another major benefit of many deterministic languages is that they provide strong safety guarantees in the form of sequential semantics: every execution of a program written in Jade, Prometheus, NESL, DPH or DPJ is exactly equivalent to a sequential execution of that program. This allows executions to be replayed and replicated without any further machinery. This is the same guarantee provided by many task parallel systems (e.g., Cilk [21]) but crucially – and unlike currently-proposed task systems – deterministic languages actually enforce the non-interference properties required for sequential semantics to hold.

2.5.3 Execution-level Determinism Systems

In contrast with language-level approaches to determinism, execution-level determinism is characterized by dynamic instrumentation to enforce determinism on a more general class of programs. Different systems have explored the costs of supporting different programming models and different memory consistency models.

2.5.3.1 Execution-Level Determinism for Arbitrary Programs

DMP's ([24] and Chapter 3) execution-level determinism contrasted sharply with previous language-based approaches to determinism with its support for arbitrary programs, including programs with data races. CoreDet ([34] and Section 4.1.1) is a compiler-based implementation of execution-level determinism that supports the TSO memory consistency model, instead of the stronger SC model as with DMP. This relaxation of consistency allows CoreDet to achieve good scalability without resorting to the complexities of speculation that DMP requires. RCDC ([26] and Chapter 4) continues CoreDet's use of store buffering but relaxes consistency to the furthest extent permissible by modern language memory models [9,10]. Combining relaxed consistency with a synchronization scheduling algorithm adapted from Kendo [23], RCDC delivers performance comparable to the original DMP proposal but with much simpler non-speculative hardware.

Calvin [144] is a hardware version of CoreDet that offers two modes of execution. In bounded mode, Calvin behaves like DMP and RCDC, generating quantum boundaries due to 1) hitting a fixed instruction count, 2) executing an atomic instruction, or 3) deterministically overflowing a hardware resource like an L1-backed store buffer. In unbounded mode, Calvin virtualizes the size of hardware resources with software. In exchange for reduced performance, Calvin's unbounded mode ends quanta only due to ISA-visible events, providing portability across different hardware implementations.

dOS [145] is an extension to Linux that provides DMP-style deterministic execution as an OS service. The dOS service eliminates "internal" nondeterminism due to timer interrupts and data races, and uses a shim layer to allow user-level control of "external" nondeterminism such as I/O. dOS uses page protection to enforce isolation between threads, which incurs much higher overheads than previous hardware and compiler proposals but works for arbitrary binaries, sans recompilation or hardware support.

Dthreads [146] is a pure-software deterministic runtime provides works for arbitrary programs using copy-on-write paging for isolation coupled with the TSO-based consistency model from

CoreDet [34]. Instead of counting instructions to define quanta, Dthreads counts synchronization operations. Using a source-level event as the marker of logical time means that deterministic program execution cannot be perturbed by small code changes, e.g., adding a single instruction, as is the case with DMP. While Dthreads guarantees determinism in the presence of data races, forward progress does not always hold for programs that use racy flag-based synchronization. Dthreads' virtual memory approach to store buffering is generally faster than CoreDet's hash-table-based implementation.

The Tern [147] and Peregrine [148] systems propose a broader notion of determinism by steering program execution onto the same schedules across a range of inputs. A set of schedules are accumulated during testing and, during production, these memoized schedules are reused whenever possible. In contrast with the traditional notion of determinism that makes execution repeatable only for a single input but can allow distinct inputs to diverge widely, Tern and Peregrine force different inputs to execute in a similar fashion. Reusing schedules helps avoid bugs by adhering to well-tested schedules, and can reduce the cost of multithreaded record-and-replay and replication as well.

2.5.3.2 Execution-Level Determinism with Restrictions

Kendo [23] provides deterministic execution for race-free programs via a library of deterministic synchronization primitives. Because Kendo assumes a race-free program, it has no need for expensive store buffering mechanisms found in other execution-level determinism systems. Kendo makes synchronization events deterministic by using threads' instruction counts to govern when lock acquires can occur (see Section 4.3.4.2); instruction counts are tracked efficiently via hardware performance counters. With its lightweight approach, Kendo's overheads on real hardware are quite reasonable, at roughly 20% with 4 threads.

Grace [22] (the predecessor to Dthreads [146]) and Determinator [149] explore support for a nested fork-join programming model with isolation enforced via virtual memory protection. Isolation, plus a limitation to deterministic synchronization idioms only, guarantees the

determinism of Grace and Determinator programs. Grace supports C/C++ programs written in a fork/join style, while Determinator provides new OS primitives for fork/join programming. Both Grace and Determinator observe that data races are eliminated via a combination of their programming model restrictions and dynamic checks. Legacy programs that use threads and locks can be mapped, somewhat inefficiently, onto Determinator's primitives using a CoreDet-like scheme.

The Concurrent Revisions [150] system offers determinism for task-parallel programs, where tasks execute in isolation from one another, modifying private copies of memory. Each task operates on a revision of memory and, when tasks join, their revisions are merged together via programmer-specified deterministic merge functions. The shared state that is modified and merged across revisions must be annotated by the programmer, and missing annotations can result in nondeterministic results. However, the judicious use of buffering only where necessary allows revisions to have low performance overhead: just a 5% increase in runtime over sequential execution for a 3D gaming workload. Subsequent extensions [151] showed that the Concurrent Revisions programming model is a natural fit for incremental computation, as isolation between tasks simplifies determining which tasks to reexecute when inputs change.

Chapter 3 A Deterministic MultiProcessor Architecture

This chapter describes the design and evaluation of a fully Deterministic MultiProcessing (DMP) shared memory computer architecture. We show that, with hardware support, arbitrary shared memory parallel programs can be executed deterministically with modest performance cost.

We define a deterministic shared memory multiprocessor system as a computer system that: (1) executes multiple threads that communicate via shared memory, and (2) will produce the same program output if given the same program input. This definition implies that a parallel program running on a DMP system is as deterministic as a single-threaded program.

The most direct way to guarantee deterministic behavior is to preserve the same global interleaving of instructions in every execution of a parallel program. However, several aspects of this interleaving are irrelevant for ensuring deterministic behavior. It is not important which global interleaving is chosen, as long as it is always the same. Also, if two instructions do not communicate, their order can be swapped with no observable effect on program behavior. The key to deterministic execution is that all communication between threads must be precisely the same for every execution. This guarantees that the program always behaves the same way if given the same input.

Guaranteeing deterministic inter-thread communication requires that each dynamic instance of an instruction (consumer) read data produced from the same dynamic instance of another instruction (producer). Producer and consumer need not be in the same thread, so this communication happens via shared memory. Interestingly, there are multiple global

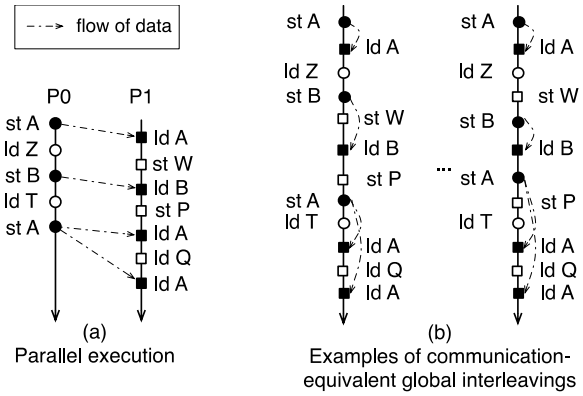

Figure 7: A parallel execution (a) and two of its multiple communication-equivalent interleavings (b). Solid markers represent communicating instructions, hollow markers represent instructions that do not communicate.

interleavings that lead to the same communication between instructions, they are called *communication-equivalent interleavings* (Figure 7). In summary, any communication-equivalent interleaving will yield the same program behavior. To guarantee deterministic behavior, then, we need to carefully control only the behavior of load and store operations that cause communication between threads. This insight is key to efficient deterministic execution.

3.1 Enforcing Deterministic Shared Memory Multiprocessing

This section describes how to build a deterministic multiprocessor system. We focus on the key mechanisms and defer discussion of specific implementations to Section 3.2. For explanatory purposes we begin with a naïve approach that serializes all threads, and then refine this simple technique into progressively more efficient organizations.

3.1.1 Basic Idea – DET-SERIAL

As seen earlier, making multiprocessors deterministic depends upon ensuring that the communication between threads is deterministic. The easiest way to accomplish this is to allow only one processor at a time to access memory in a deterministic order. This process can be thought of as a "memory access token" being deterministically passed among the processors. We call this deterministic serialization of a parallel execution (Figure 8b). Deterministic serialization guarantees that inter-thread communication is deterministic by preserving all pairs of communicating memory instructions.

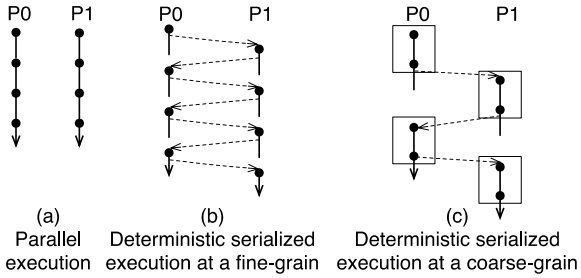

Figure 8: Deterministic serialization of memory operations. Dots are memory operations and dashed arrows are happens-before synchronization.

The simplest way to implement such serialization is to have each processor obtain the memory access token (henceforth called deterministic token) and, when the memory operation is completed, pass it to the next processor in the deterministic order. A processor blocks whenever it needs to access memory but does not have the deterministic token.

Waiting for the token at every memory operation is likely to be expensive and will cause significant performance degradation compared to the original parallel execution (Figure 8a). Performance degradation stems from overhead introduced by waiting and passing the deterministic token and from the serialization itself, which removes the benefits of parallel execution. Synchronization overhead can be mitigated by synchronizing at a coarser granularity (Figure 8c), allowing each processor to execute a finite, deterministic number of instructions, a *quantum*, before passing the token to the next processor. We refer to a system with serialization at the granularity of quanta as DET-SERIAL. The process of dividing the execution into quanta is called quantum building: the simplest way to build a quantum is to break execution up into fixed instruction counts, on the order of tens of thousands of instructions. We call this simple quantum building policy QB-COUNT.

3.1.2 Handling Application-Level Synchronization

Note that this deterministic serialization does not interfere (e.g., introduce deadlocks or violate memory ordering requirements) with application-level synchronization. This serialized execution is a valid execution schedule in any traditional nondeterministic system. Moreover, the deterministic execution systems we propose provide sequential consistency [65] if full memory fences are inserted at quantum boundaries (Section 3.2). Hence, the extra synchronization imposed by DET-SERIAL (and the other deterministic execution variants) resides below application-level synchronization and the two do not impact one another's correctness. Awareness of application-level synchronization can, however, improve performance (Section 3.1.4).

3.1.3 Recovering Parallelism

Reducing the impact of serialization requires enabling parallel execution while preserving the same execution behavior as deterministic serialization. We propose two techniques to recover parallelism. The first technique exploits the fact that threads do not communicate all the time, allowing concurrent execution of communication-free periods. The second technique uses speculation to allow parallel execution of quanta from different processors, re-executing quanta when determinism might have been violated.

3.1.3.1 Leveraging Communication-Free Execution – DET-SHTAB

The performance of deterministic parallel execution can be improved by leveraging the observation that threads do not communicate all the time. Periods of the execution that do not communicate can execute in parallel with other threads. Thread communication, however, must happen deterministically. With DET-SHTAB, we achieve this by falling back to deterministic serialization only while threads communicate. Each quantum is broken into two parts: a communication-free prefix that executes in parallel with other quanta, and a suffix, from the first point of communication onwards, that executes serially. The execution of the serial suffix is deterministic because each thread runs serially in an order determined by the deterministic token, just as in DET-SERIAL. The transition from parallel execution to serial execution is deterministic because it occurs only when all threads are blocked – each thread will block either at its first point of inter-thread communication or, if it does not communicate with other threads, at the end of its current quantum. Thus, each thread blocks during each of its quanta (though possibly not until the end), and each thread blocks at a deterministic point within each quantum because communication is detected deterministically (described later).

Inter-thread communication occurs when a thread writes to shared (i.e., non-private) pieces of data. In this case, the system must guarantee that all threads observe such writes at a deterministic point in their execution. Figure 9 illustrates how this is enforced in DET-SHTAB. There are two important cases: (1) reading data held private by a remote processor, and (2) writing to shared data (privatizing it). Case (1) is shown in Figure 9a: when quantum 2 attempts to read data that is held private by a remote processor P0, it must first wait for the deterministic token and for all other threads to be blocked waiting for the deterministic token. In this example, the read cannot execute until quantum 1 finishes executing. This is necessary to guarantee that quantum 2

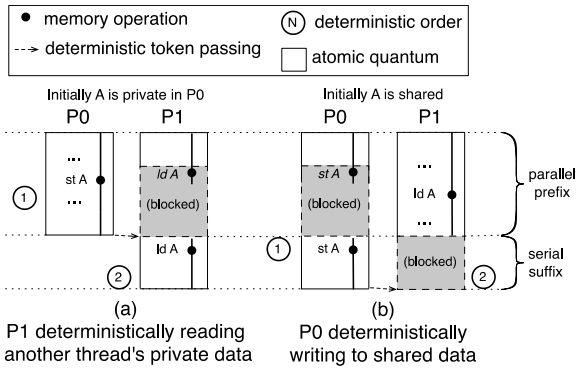

Figure 9: Recovering parallelism by overlapping communication-free execution.

always gets the same data, since quantum 1 might still write to A before it completes executing. Case (2) is shown in Figure 9b: when quantum 1, which already holds the deterministic token, attempts to write to a piece of shared data, it must also wait for all other threads to be blocked waiting for the deterministic token. In this example, the store cannot execute until quantum 2 finishes executing. This is necessary to guarantee that all processors observe the change of the state of A (from shared to privately held by a remote processor) at a deterministic point in their execution. Note that each thread waits to receive the token when it reaches the end of a quantum before starting its next quantum. This periodically (and deterministically) allows a thread waiting for all other threads to be blocked to make progress.

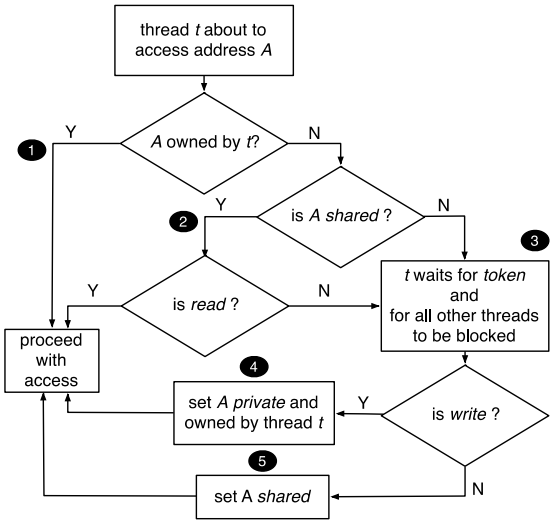

Figure 10: Deterministic serialization of shared memory communication only.

To detect writes that cause communication, DET-SHTAB needs a global data-structure to keep track of the sharing state of memory positions. A *sharing table* is a conceptual data structure that contains sharing information for each memory position; it can be kept at different granularities, e.g., line or page. Figure 10 shows a flowchart of how the sharing table is used. Some accesses can freely proceed in parallel: a thread can access its own private data without holding the deterministic token (1) and it can also read shared data without holding the token (2). However, in order to write to shared data or read data regarded as private by another thread, a thread needs to wait for its turn in the deterministic total order, when it holds the token and all other threads are blocked also waiting for the token (3). This guarantees that the sharing information is kept consistent and its state transitions are deterministic. When a thread writes to a piece of data, it becomes the owner of the data (4). Similarly, when a thread reads data not yet read by any thread, it becomes the owner of the data. Finally, when a thread reads data owned by another thread, the data becomes shared

(5). Other deterministic policies to govern sharing are possible, but this policy, modeled on MESI cache coherence protocols, works well for applications with locality.

In summary, DET-SHTAB lets threads run concurrently as long as they are not communicating. As soon as they attempt to communicate, DET-SHTAB deterministically serializes communication and updates the sharing table to exploit locality.

3.1.3.2 Leveraging Support for Transactional Memory – DET-TM and DET-TMFWD

Executing quanta atomically and in isolation in a deterministic total order is equivalent to deterministic serialization. To see why, consider a quantum executed atomically and in isolation as a single instruction in the deterministic total order, which is the same as DET-SERIAL. Transactional Memory [30] can be leveraged to make quanta *appear* to execute atomically and in isolation. This, coupled with a deterministic commit order, makes execution equivalent to deterministic serialization while recovering parallelism.

We use TM support by encapsulating each quantum inside a transaction, making it appear to execute atomically and in isolation. In addition, we need a mechanism to form quanta deterministically and another to enforce a deterministic commit order. As Figure 11a illustrates, speculation allows a quantum to run concurrently with other quanta in the system as long as there are no overlapping memory accesses that would violate the original deterministic serialization of memory operations. In case of conflict, the quantum later in the deterministic total order gets squashed and re-executed (2). We enforce a deterministic commit order by requiring a processor to hold the deterministic token in order to commit. Once a processor is done committing, it passes the token to the next processor in the deterministic order. Note that

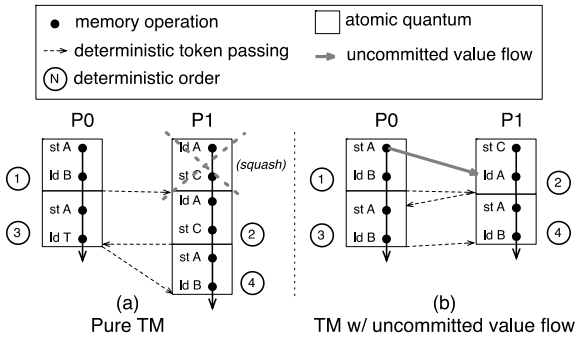

Figure 11: Recovering parallelism by executing quanta as memory transactions (a). Avoiding unnecessary squashes with un-committed data forwarding (b).

the deterministic total order of quantum commits is a key component in guaranteeing deterministic serialization of memory operations. We call this system DET-TM.

Having a deterministic commit order also allows isolation to be selectively relaxed, further improving performance by allowing uncommitted (or speculative) data to be forwarded between quanta. This can potentially save a large number of squashes in applications that have more inter-thread communication. To do so, we allow a quantum to fetch speculative data from another uncommitted quantum earlier in the deterministic order. This is illustrated in Figure 11b, where quantum 2 fetches an uncommitted version of A from quantum 1. Note that without support for forwarding, quantum 2 would have been squashed. To guarantee correctness, if a quantum that provided data to other quanta is squashed, all subsequent quanta must also be squashed since they might have consumed incorrect data. We call a DMP system that leverages support for TM with forwarding DET-TMFWD.

Another interesting effect of pre-defined commit ordering is that *memory renaming*, analogous to register renaming in out-of-order processors, can be employed to avoid squashes on write-after-write and write-after-read conflicts. For example, in Figure 11a, if quanta 3 and 4 execute concurrently, the store to A in (3) need not squash quantum 4 despite their write-after-write conflict.

3.1.4 Exploiting the Critical Path – QB-SYNCFOLLOW, QB-SHARING and QB-SYNCSHARING

The most basic quantum building policy, QB-COUNT, produces quanta based on counting instructions and breaking a quantum when a deterministic, target number of instructions is reached. However, instruction-count based quantum building does not capture the fact that threads execute instructions at different rates, and this can lead to idle time at quantum boundaries. It also does not capture the fact that multi-threaded programs have a critical path. Intuitively, the critical thread changes as threads communicate with each other via synchronization operations and data sharing.

We now describe how to exploit typical program behavior to adapt the size of quanta and lead to more efficient progress on the critical path of execution. We devised three heuristics to do so. The first heuristic, called QB-SYNCFOLLOW, simply ends a quantum when an unlock operation is performed. As Figure 12 shows, the rationale is that when a thread releases a lock (P0), other threads might be spinning waiting for that lock (P1), so the deterministic token should be sent forward as early as possible to allow the waiting thread to make progress. In addition, QB-SYNCFOLLOW passes the token forward immediately if a thread starts spinning on a lock.

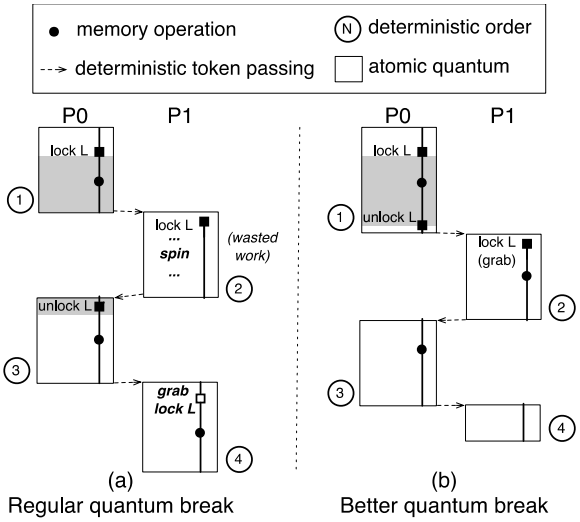

Figure 12: Example of a situation when better quantum breaking policies lead to better performance.

The second heuristic relies on information about data sharing to identify when a thread has potentially completed work on shared data, and consequently ends a quantum at that time. It does so by determining when a thread hasn't issued memory operations to shared locations in some time, e.g., in the last 30 memory operations. The rationale is that when a thread is working on shared data, it is expected that other threads will access that data soon. By ending a quantum early and passing the deterministic token, the consumer thread potentially consumes the data earlier than if the quantum in the producer thread ran longer. This not only has an effect on performance in all DMP techniques, but also reduces the amount of work wasted by squashes in DET-TM and DET-TMFWD. We call this quantum building heuristic QB-SHARING.

In addition, we explore a combination of QB-SYNCFOLLOW and QB-SHARING, which we refer to as QB-SYNCSHARING. This quantum building strategy monitors synchronization events and sharing behavior. QB-SYNCSHARING determines the end of a quantum whenever either of the other two techniques would have decided to do so.

3.2 Implementation Issues

As seen in the previous section, implementing an execution-level determinism system requires a mechanism to deterministically break the execution into quanta and mechanisms to guarantee the properties of deterministic serialization. A system could have all these mechanisms completely in hardware, completely in software or even as a mix of hardware and software components. The trade-off is one of complexity versus performance. A hardware-only implementation offers better performance but requires changes to the multiprocessor hardware. Conversely, a software-only implementation performs worse but does not require special hardware. This section discusses the relevant points in each implementation.

3.2.1 Hardware-Only Implementation

Quantum Building: The simplest quantum building policy, QB-COUNT, is implemented by counting dynamic instructions as they retire and placing a quantum boundary when the desired quantum size is reached. QB-SYNCFOLLOW requires access to information about synchronization, which can be obtained by a compiler or annotations in the synchronization libraries. On a weakly ordered consistency model, synchronization can be inferred from the fence instructions present in a binary, but with stronger models like TSO lock releases are not readily distinguishable from regular stores. QB-SHARING requires monitoring memory accesses and determining whether they are to shared data or not, which is done using the sharing table (Section 3.1.3.1), discussed later in this section. Finally, QB-SYNCSHARING is exactly a logical *OR* of the decision made by QB-SYNCFOLLOW and QB-SHARING. Regardless of the quantum building policy used, depending upon the consistency model of the underlying hardware, threads must perform a memory fence at the edge of a quantum, which is where inter-thread communication occurs.

We now describe the hardware implementations of DET-SERIAL, DET-SHTAB, DET-TM and DET-TMFWD, which we label with a "DMP-" prefix to distinguish them from software implementations discussed in Section 3.3.2.

DMP-SERIAL: DET-SERIAL is implemented in hardware with a token that is passed between processors in the deterministic order. The hardware supports multiple tokens, allowing multiple deterministic processes at the same time – each process has its own token.

DMP-SHTAB: The sharing table data-structure used by DET-SHTAB keeps track of the sharing state of data in memory. Our hardware implementation of the sharing table leverages the cache line state maintained by a MESI cache coherence protocol. A line in exclusive or modified state is considered private by the local processor so, as the flowchart in Figure 10 shows, it can be freely read or written by its owner thread without holding the deterministic token. The same applies for a read operation on a line in shared state. Conversely, a thread needs to acquire the deterministic token before writing to a line in shared state, and moreover, all other threads must be at a deterministic point in their execution, e.g., blocked. The state of the entries in the sharing table corresponding to lines that are not cached by any processor is kept in memory and managed by the memory controller, much like a directory in directory-based cache coherence. Note, however, that we do not require directory-based coherence per se. This state is transferred when cache misses are serviced. Nevertheless, directory-based systems can simplify the implementation of DMP-SHTAB even further.

We now address how the state changes of the sharing table happen deterministically. There are three requirements: (1) speculative instructions cannot change the state of the sharing table; (2) a coherence request that changes the state of a cache line can only be performed during the serial suffix of a quantum when the issuer holds the deterministic token; and (3) all nodes need to know when the other nodes are blocked waiting for the deterministic token – this is necessary to implement step 3 in Figure 10. To guarantee (1), speculative instructions that need to change the sharing table can only do so when they are not speculative anymore. To guarantee (2), for all coherence requests performed during the parallel prefix of a quantum, the servicing node nacks any request that implies a change in a line's coherence state, e.g., a downgrade, otherwise the node processes the request as usual. Finally, (3) is guaranteed by having all processors broadcast when they block or when they unblock.

Alternatively, sharing table implementations can use memory tagging, where the tags represent the sharing information. Moreover, our evaluation (Section 3.4) shows that tracking sharing information at page granularity does not suffer from excessive false sharing. This suggests a page-level implementation, which is simpler than a line-level implementation.

DMP-TM and DMP-TMFWD: On top of standard TM support, a hardware implementation of DET-TM needs a mechanism to enforce a specific transaction commit order – the deterministic commit order of quanta encapsulated inside transactions. DMP-TM does that by allowing a transaction to commit only when the processor receives the deterministic token. After a single commit, the processor passes the token to the next processor in the deterministic order. DET-TMFWD requires more elaborate TM support to allow speculative data to flow from uncommitted quanta earlier in the deterministic order. This is implemented by making the coherence protocol aware of the data version of quanta, very similarly to versioning protocols used in Thread-Level Speculation (TLS) systems [152]. One interesting aspect of DMP-TM is that, if a transaction overflow event is made deterministic, it can be used as a quantum boundary, making a bounded TM implementation perfectly suitable for a DMP-TM system. Making transaction overflow deterministic requires making sure that updates to the speculative state of cache lines happen strictly as a function of memory instruction retirement, i.e., updates from speculative instructions cannot be permitted. In addition, it also requires all non-speculative lines to be displaced before an overflow is triggered, i.e., the state of non-speculative lines cannot affect the overflow decision.

The implementation choices in a hardware-only DMP system also have performance versus complexity trade-offs. DMP-TMFWD offers better performance but requires mechanisms for speculative execution, conflict detection and memory versioning, whereas DMP-SHTAB performs a little worse but does not require speculation.

3.2.2 Software-Only Implementation

A deterministic system can also be implemented using a compiler or a binary rewriting infrastructure. The implementation details are largely similar to the hardware implementations.

The compiler builds quanta by sparsely inserting code to track dynamic instruction count in the control-flow-graph – quanta need not be of uniform size as long as the size is deterministic. This is done at the beginning and end of function calls, and at the tail end of CFG back edges. The inserted code tracks quantum size and, when the target size has been reached or exceeded, it calls back to a runtime system, which implements the various determinism techniques. DET-SERIAL is supported in software by implementing the deterministic token as a queuing lock. For DET-SHTAB, the compiler instruments every load and store to call back to the runtime system, the runtime system implements the logic shown in Figure 12, and the sharing table itself is kept in memory.

It is also possible to implement a deterministic system using software transactional memory (STM), but, as discussed in [34], the assumptions made by STM systems do not mesh well with DET-TM's requirements. First, STM systems assume that most code executes non-transactionally, contrary to DET-TM's transactions-all-the-time approach. With DET-TM, the overhead of running software transactions cannot be amortized by executing non-transactional code. Second, transactions in DET-TM are not lexically scoped at the source language level. Supporting un-scoped transactions requires an STM system that can roll back the call stack arbitrarily, which in turn requires instrumenting all stack reads and writes – a significant runtime cost.

3.2.3 Leveraging Commercial Hardware Transactional Memory

Implementing DMP-TM on top of commercially available hardware TM systems [31–33] is also possible. These systems do not offer the ability to modify cache eviction policies, or control the speculative forwarding of values between concurrent transactions. The biggest obstacle, however, is ensuring transactions commit in a deterministic order. On Intel's upcoming Haswell architecture [33], for example, there is no way to perform non-transactional accesses from within a transaction. Such accesses could be used to implement a spin-loop at the end of a transaction to ensure it did not commit out of order. In the absence of such an out-of-band communication channel, deterministic transaction commit cannot be enforced directly because any bidirectional communication between transactions will cause one of the transactions to roll back.

However, a mechanism can be built that will *detect* out-of-order transaction commits by injecting synthetic data dependences between quanta. Say we have three threads T0, T1 and T2 executing three quanta A, B and C, respectively, and the deterministic commit order is A, then B, then C. At the end of quantum A, T0 writes a sentinel value to location A'. At the end of quantum B, T1 reads A' and, if the sentinel value is not present, rolls back its transaction for quantum B. If the sentinel is present T1 writes a sentinel value to location B'. At the end of quantum C, T2 then reads B', checking for the sentinel, etc. The data dependences verify that a quantum is serialized in the appropriate order, otherwise it (and all quanta ordered after it) will be rolled back.

This order-violation detection mechanism is likely to trigger many needless rollbacks in practice. Optimizing quantum formation to suit the limitations of HTM systems is another interesting open question. While these and other issues will prove challenging, we are optimistic that upcoming HTM support will accelerate the performance of determinism on real systems.

3.3 Experimental Setup

We evaluate both hardware and software implementations of a DMP system. We use the SPLASH2 [153] and PARSEC [154] benchmark suites and run the benchmarks to completion. Some benchmarks were not included due to infrastructure problems such as out of memory errors and other system issues such as lack of 64-bit compatibility. Note that the input sizes for the software implementation experiments are typically larger than the ones used in the simulation runs due to simulation time constraints. We ran our native experiments on a machine with dual Intel Xeon quad-core 64-bit processors (8 cores total) clocked at 2.8 GHz, with 8GB of memory running Linux 2.6.24. In the sections below we describe the evaluation environment for each implementation category.

3.3.1 Hardware Implementation

We assess the performance trade-offs of the different hardware implementations of DMP systems with a simulator written using PIN [155]. The model includes the effects of serialized execution,[3] quantum building, memory conflicts, speculative execution squashes and buffering for a single outstanding transaction per thread in DMP-TM. Note that even if execution behavior is deterministic, simulated performance is not deterministic because it relies on traces generated (nondeterministically) from Pin. Therefore we run our simulator multiple times, average the results and provide error bars showing the 90% confidence interval for the mean. While the model simplifies some microarchitectural details, the specifics of the various DMP system implementations are modeled in detail. To reduce simulation time, our model assumes that the IPCs (including squashed instructions) of all the different DMP modes are the same. This reasonable assumption allows us to compare performance between different DMP schemes using our infrastructure. Note that the comparison baseline (nondeterministic parallel execution) also runs on our simulator.

3.3.2 Software Implementation

We evaluate the performance impact of a software-based determinism system (CoreDet[4]) by using a compiler pass written for LLVM v2.2 [156]. Its main transformations are described in Section 3.2.2. The pass is executed after all other compiler optimizations. Once the object files are linked to the runtime environment, LLVM does another complete link-time optimization pass to inline the runtime library with the main object code.

The runtime system provides a custom pthreads-compatible thread management and synchronization API. Finally, the runtime system allows the user to control the maximum quantum size and the granularity of entries in the sharing table. We configured these parameters on a per-application basis, with quantum sizes varying from 10,000 to 200,000 instructions, and sharing table entries from 64B to 4KB. Our CoreDet experiments run on real hardware, so we took multiple

[3] Note that the simulation actually serializes quanta execution functionally, which affects how the system executes the program. This accurately models the effects of quanta serialization on application behavior.
[4] This use of the term CoreDet is anachronistic. See Section 1.5 for details.

runs, averaged their running time and provided error bars in the performance plot showing the 90% confidence interval for the mean. The focus of this chapter is on the deterministic algorithms and their hardware implementations, so we omit a detailed description and evaluation of our software-only implementation.

3.4 Evaluation

We first show the scalability of our hardware proposals: in the best case, DMP-SHTAB has negligible overhead compared to nondeterministic parallel execution with 16 threads, while the more aggressive DMP-TMFWD reduces DMP-SHTAB's overheads by 20% on average. We then examine the sensitivity of our hardware proposals to changes in quantum size, conflict detection granularity, and quantum building strategy. Finally, we show the scalability of our software-only COREDET-SHTAB proposal and demonstrate that it does not unduly limit performance scalability. We believe that DMP-SHTAB represents a good trade-off between performance and complexity, and that COREDET-SHTAB is fast enough to be useful for debugging and, depending on the application, deployment purposes.

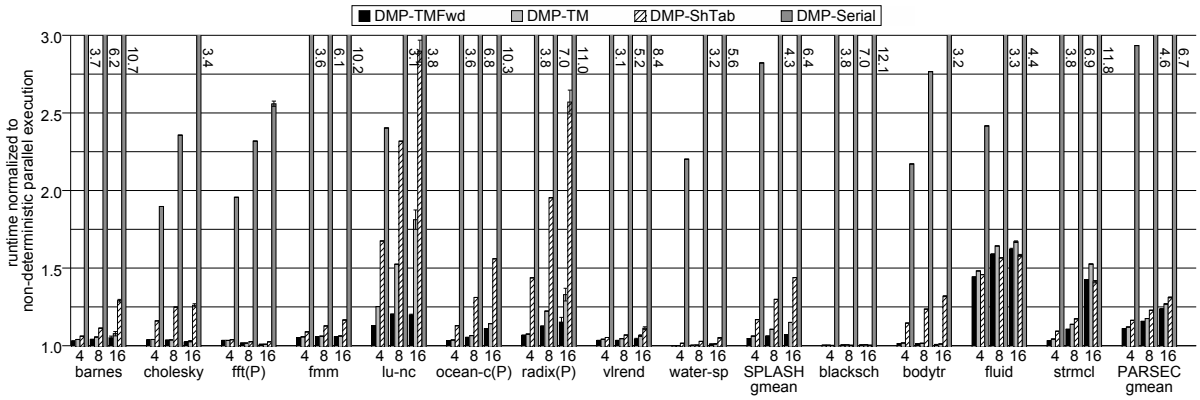

Figure 13: Runtime overheads with 4, 8 and 16 threads. (P) indicates page-level conflict detection; otherwise line-level detection is used.

3.4.1 Performance and Scalability

Figure 13 shows the scalability of our techniques compared to the nondeterministic, parallel baseline. We ran each benchmark with 4, 8 and 16 threads, and QB-SYNCSHARING producing 1,000-instruction quanta. As one would expect, DMP-SERIAL exhibits slowdown nearly linear with the number of threads. The degradation can be sub-linear because DMP affects only the parallel

behavior of an application's execution. DMP-ShTab has 38% overhead on average with 16 threads, and in the few cases where DMP-ShTab has larger overheads (e.g., lu-nc), DMP-TM provides much better performance. For an additional cost in hardware complexity, DMP-TMFwd, with an average overhead of only 21%, provides a consistent performance improvement over DMP-TM. DMP-ShTab and the TM-based schemes all scale sub-linearly with the number of processors. The overhead for TM-based schemes is flat for most benchmarks, suggesting that a TM-based system would be ideal for larger DMP systems. Thus, with the right hardware support, the performance of deterministic execution can be very competitive with nondeterministic parallel execution.

3.4.2 Sensitivity Analysis

Figure 14 shows the effects of changing the maximum number of instructions included in a quantum. Again, we use the QB-SyncSharing scheme, with line-granularity conflict detection. Increasing the size of quanta consistently degrades performance for the TM-based schemes, as larger quanta increase the likelihood and cost of aborts since more work is lost. The ability of DMP-TMFwd to avoid conflicts helps increasingly as the quanta size gets larger. With DMP-ShTab, the effect is more application dependent: most applications (e.g., vlrend) do worse with larger quanta, since each quantum holds the deterministic token for longer, potentially excluding other threads from making progress. For lu-nc, however, the effect is reversed: lu-nc has relatively large communication-free regions per quantum, allowing the token-passing overhead to be amortized better. On average, however, DMP-Serial is less affected by quantum size.

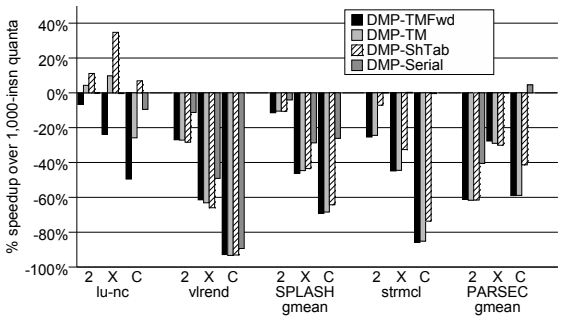

Figure 14: Performance of 2,000 (2), 10,000 (X) and 100,000 (C) instruction quanta, relative to 1,000 instruction quanta.

Figure 15 compares conflict detection at cache line (32-byte) and page (4096-byte) granularity. Increasing the conflict detection granularity decreases the performance of the TM-based schemes, as they suffer more (likely false) conflicts. The gap between DMP-TMFWD and DMP-TM grows as the former can avoid some of the conflicts by forwarding values. DMP-SERIAL is unaffected, because it does no conflict detection. With DMP-SHTAB, a coarser granularity can lead to more blocking (e.g., lu-nc and streamcluster) but can also, surprisingly, improve performance by taking advantage of spatial locality (e.g., radix, ocean-c). This suggests a pro-active privatization/sharing mechanism to improve the performance of DMP-SHTAB. On average, our results show that exploiting existing virtual memory support to implement DMP-SHTAB could be quite effective.

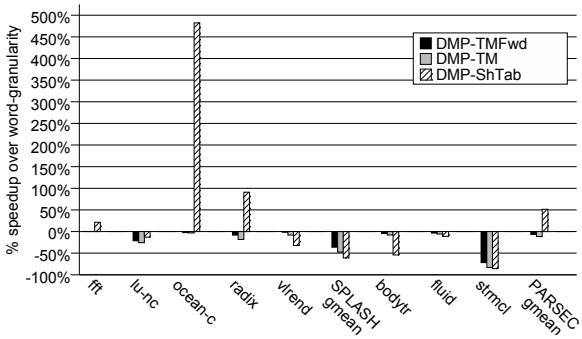

Figure 15: Performance of page-granularity conflict detection, relative to line-granularity.

Figure 16 shows the performance effect of different quantum building strategies. Smarter quantum builders generally do not improve performance much over the QB-COUNT 1,000-instruction baseline, as QB-COUNT produces such small quanta that heuristic breaking cannot substantially accelerate progress along the application's critical path. With 10,000-instruction quanta (Figure 17), the effects of the different quantum builders are more pronounced. In general, the quantum builders that take program synchronization into account (QB-SYNCFOLLOW and QB-SYNCSHARING) outperform those that do not. DMP-SERIAL and DMP-SHTAB perform better with smarter quantum building,

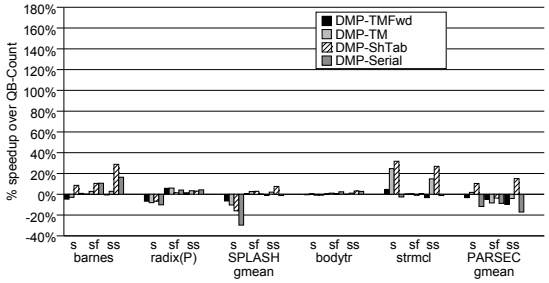

Figure 16: Performance of QB-SHARING (s), QB-SYNCFOLLOW (sf) and QB-SYNCSHARING (ss) quantum builders, relative to QB-COUNT, with 1,000-insn quanta.

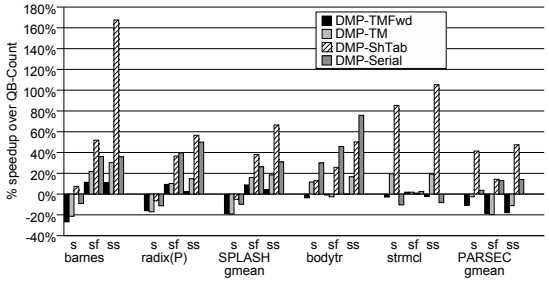

Figure 17: Performance of quantum building schemes, relative to QB-COUNT, with 10,000-insn quanta.

while the TM-based schemes are less affected, as TM-based schemes recover more parallelism. QB-SHARING works well with DMP-SHTAB and PARSEC, and works synergistically with synchronization-aware quantum building: QB-SYNCSHARING often outperforms QB-SYNCFOLLOW (as with barnes).

Table 2: Characterization of hardware DMP results. †Same granularity as used in Figure 13.

	DMP with 1,000-insn quanta						QB Strategy, with 10,000-insn quanta†					
	TM				ShTab							
	Line		Page		Line	Page	SyncFollow		Sharing		SyncSharing	
Benchmark	R/W set size	% conflicts	R/W set size	% conflicts	% Q overlap	% Q overlap	avg Q size	% sync breaks	avg Q size	% sharing breaks	avg Q size	% sync breaks
barnes	27/9	37	9/2	64	47	46	5929	42	4658	67	5288	54
cholesky	14/6	23	3/1	39	31	38	6972	30	3189	94	6788	35
fft	22/16	25	3/4	26	19	39	9822	1	3640	62	4677	49
fmm	30/6	51	7/1	69	33	29	8677	15	4465	65	5615	50
lu-nc	47/33	71	6/4	77	14	16	7616	24	6822	37	6060	42
ocean-c	46/15	28	5/2	34	5	46	5396	49	3398	73	3255	73
radix	16/20	7	3/7	13	31	42	8808	15	3346	71	4837	57
vlrend	27/8	38	7/1	50	41	39	7506	28	7005	45	6934	38
water-sp	32/19	19	5/1	45	40	37	7198	5	5617	30	6336	20
SPLASH amean	30/16	31	5/2	44	29	35	7209	27	4987	57	5363	48
blacksch	28/9	8	14/1	10	48	48	10006	<1	9163	10	9488	7
bodytr	11/4	16	3/2	28	39	19	7979	25	7235	31	6519	37
fluid	41/8	76	8/2	75	43	40	871	98	2481	95	832	99
strmcl	36/5	28	10/2	91	60	12	9893	1	1747	79	2998	77
PARSEC amean	29/6	36	9/1	51	45	30	7228	19	5156	54	3880	64

3.4.3 Characterization

Table 2 provides more insight into our sensitivity results. For DMP-TM, with both line- and page-level conflict detection, we give the average read- and write-set sizes (which show that our TM buffering requirements are modest), and the percentage of quanta that suffer conflicts. The percentage of conflicts is only roughly correlated with performance, as not all conflicts are equally expensive. For the DMP-SHTAB scheme, we show the amount of execution overlap of a quantum with other quanta (parallel prefix), as a percentage of the average quantum size. This metric is highly correlated with performance: the more communication-free work exists at the beginning of each quantum, the more progress a thread can make before needing to acquire the deterministic token. Finally, we give the average quantum size and the percentage of quanta breaks caused by the heuristic of each of the quantum builders, with a 10,000-instruction maximum quanta size. The average quantum size for QB-COUNT is uniformly very close to 10,000 instructions, so we omit those results. Since the average quantum size for QB-SYNCFOLLOW is generally larger than that for QB-

SHARING, and the former outperforms the latter, we see that smaller quanta are not always better: it is important to choose quantum boundaries well, as QB-SYNCFOLLOW does.

3.4.4 CoreDet[5]: Performance and Scalability

Figure 18 shows the performance and scalability of CORESHTAB compared to the parallel baseline. We see two classes of trends, slowdowns that increase with the number of threads (e.g., barnes) and slowdowns that don't increase much with the number of threads (e.g., fft). For benchmarks in the latter class, adding more threads substantially improves raw performance. Even for benchmarks in the former class, while adding threads does decrease raw performance compared to the corresponding parallel baseline, the slowdown is sublinear in the number of threads. Thus, adding threads still results in an improvement in raw performance. In summary, this data shows that CORESHTAB does not unduly limit performance scalability for multithreaded applications.

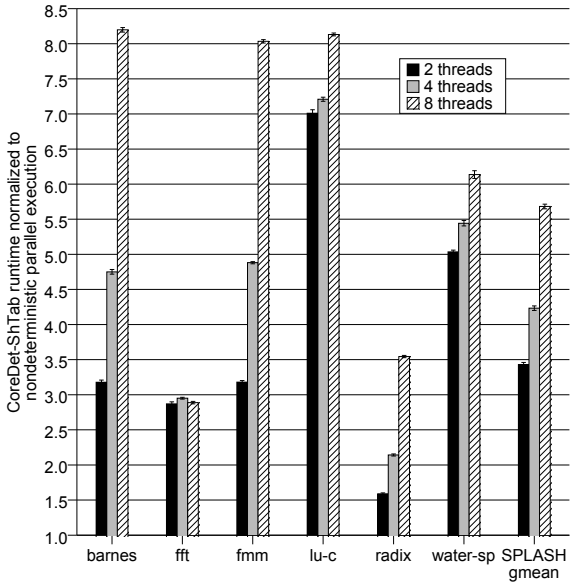

Figure 18: Runtime of CORESHTAB relative to nondeterministic execution.

3.5 Discussion

Our evaluation of the various DMP schemes leads to several conclusions. At the highest level, the conclusion is that deterministic execution in a multiprocessor environment is achievable on future systems with little, if any, performance degradation. The simplistic DMP-SERIAL has a geometric mean slowdown of 6.5x on 16 threads. By orchestrating communication with DMP-SHTAB, this slowdown reduces to a geometric mean of 37% and often less than 15%. By using speculation with DMP-TM, we were able to reduce the overhead to a geometric mean of 21% and often less than 10%. Through the addition of forwarding with DMP-TMFWD, the overhead of deterministic execution is less than 15% and often less than 8%. Finally, software solutions can provide

[5] This use of the term CoreDet is anachronistic. See Section 1.5 for details.

deterministic execution with a performance cost suitable for debugging on current generation hardware, and depending upon the application, deployment.

Now that we have defined what deterministic shared memory multiprocessing is, shown how to build efficient hardware support for it, and demonstrated that software solutions can be built for current generation hardware, we discuss several additional points. These are: (1) performance, complexity and energy trade-offs; (2) support for debugging; (3) interaction with operating system and I/O nondeterminism; and (4) making deterministic execution portable for deployment.

Implementation Trade-offs. Our evaluation showed that using speculation in the hardware-based implementation pays off in terms of performance. However, speculation potentially wastes energy, requires complex hardware and has implications in system design, since some code, such as I/O and parts of an operating system, cannot execute speculatively. Fortunately, DET-TM, DET-SHTAB and DET-SERIAL can coexist in the same system. One easy way to co-exist is to switch modes at a deterministic boundary in the program (e.g., the edge of a quanta). More interestingly, a DMP system can be designed to support multiple modes co-existing simultaneously. This allows a DMP system to use the most convenient approach depending on what code is running, e.g., using speculation (DET-TM) in user code and avoiding it (DET-SHTAB) in kernel code.

DMP systems could also be built in a hybrid hardware-software fashion, instead of a purely hardware or software implementation. A hybrid DET-TM system, for example, could leverage modest hardware TM support while doing quantum building and deterministic ordering more flexibly in software with low performance cost. A hybrid DET-SHTAB system could efficiently implement the sharing table by leveraging modern cache coherence protocols; exposing coherence state transitions could enable a simple high performance hybrid DET-SHTAB implementation in the near future. Chapter 4 explores the idea of a hybrid hardware-software system in greater depth.

Supporting Debugging Instrumentation. In order to enable a debugging environment in a DMP system, we need a way of allowing the user to instrument code for debugging while preserving the interleaving of the original execution. To accomplish this, implementations must support a

mechanism that allows a compiler to mark code as being inserted for instrumentation purposes only; such code will not affect quantum building, and thus preserves the original behavior.

Dealing with nondeterminism from the OS and I/O. There are many sources of nondeterminism in today's systems, from concurrently running processes to the state of hardware predictors and arbiters. A DMP system hides most of them, allowing many multithreaded programs to run deterministically. Besides hiding the nondeterminism of the microarchitecture, a DMP system also hides the nondeterminism of OS thread scheduling by using the deterministic token to provide low-level deterministic thread scheduling, causing threads to run in the same order on every execution. Nevertheless, challenging sources of nondeterminism remain.

One challenge is that parallel programs can use the operating system to communicate between threads. A DMP system needs to make that communication deterministic. One way to address the problem is to execute OS code deterministically, which was discussed earlier in this section. Alternatively, a layer between the operating system and the application can be utilized to detect communication and synchronization via the kernel and provide it within the application itself. This is the solution employed by our software implementation.

Another challenge is that many operating system API calls allow nondeterministic outcomes. System calls such as `read` may lead to variations in program execution from run to run, as their API specification itself permits such variation. There are two ways to handle nondeterministic API specifications: ignore them – any variation in outcome could also have occurred to sequential code; or fix them, by providing alternative, deterministic APIs. With `read`, a solution is to always return the maximum amount of data requested until EOF. The dOS [145] deterministic OS explores issues of OS-nondeterminism even further, providing a deterministic process abstraction that provides deterministic shared memory communication and identifies other sources of nondeterminism in the POSIX API. dOS allows programmers to write handlers for these nondeterministic sources, simplifying multithreaded record-and-replay and the replication of multithreaded programs.

The final, and perhaps most difficult challenge is that the real world is simply nondeterministic. Ultimately, programs interact with remote systems and users, which are nondeterministic and can affect thread interleavings. With DMP, the inputs on which a program's output depends are fewer, and easier to identify and control than with nondeterministic multiprocessors. Though the sources of these inputs may be nondeterministic, these inputs are much lower-volume than the totality of shared memory communication. Thus, DMP allows many multithreaded programs to execute completely deterministically, and allows the behavior of the remaining programs to be debugged, recorded and replicated more easily.

Support for deployment. We contend that deterministic systems should not just be used for development, but for deployment as well. We believe systems in the field should behave like systems used for testing. The reason is twofold. First, developers can have higher confidence their programs will work correctly once deployed. Second, if the program does crash in the field, then deterministic execution provides a meaningful way to collect and replay crash history data. Supporting deterministic execution across different physical machines places additional constraints on the implementation. Quanta must be built the same across all systems. This means machine-specific effects cannot be used to end quanta (e.g., micro-op count, or a full cache-set for bounded TM-based implementations). Furthermore, passing the deterministic token across processors must be the same for all systems. This suggests that DMP hardware should provide the core mechanisms, and leave the quanta building and scheduling control up to software. The Calvin [144] system explores similar issues (see Section 2.5.3.1).

3.6 Conclusions

This chapter made the case for fully deterministic shared memory multiprocessing. We have shown that the key requirement to support deterministic execution is deterministic communication via shared memory. Fortunately, this requirement still leaves room for efficient implementations. We described a range of implementation alternatives, in both hardware and software, with varying degrees of complexity and performance cost. Our simulations show that a hardware implementation of a DMP system can have negligible performance degradation over

nondeterministic systems. We also briefly described our compiler-based software-only deterministic system and show that while the performance impact is significant, it is quite tolerable for debugging.

While the benefits for debugging are obvious, we suggest that parallel programs should always run deterministically. Deterministic execution in the field has the potential to increase reliability of parallel code, as the system in the field would behave similarly to in-house testing environments, and to allow a more meaningful collection of crash information.

Perhaps contrary to popular belief, a shared memory multiprocessor system can execute programs deterministically with little performance cost. We believe that deterministic multiprocessor systems are a valuable goal, as they abstract away several difficulties in writing, debugging and deploying parallel code.

Chapter 4 Trading Strong Memory Consistency for Simpler Determinism

Relaxing memory ordering has proven instrumental in improving performance and scalability in conventional nondeterministic shared-memory multiprocessor architectures [157,158]. While speculation alleviates some of the costs of strong ordering [55,67,71,159] in complex architectures, we still relax memory ordering to allow compiler optimizations and to simplify hardware. Interestingly, strong memory ordering has a much higher cost in deterministic multiprocessing than in nondeterministic multiprocessing. Therefore we argue that, in deterministic multiprocessors, it is even more important to give up strong memory ordering in favor of higher performance and lower complexity.

This chapter discusses RCDC, a Relaxed Consistency Deterministic Computer system. RCDC improves upon the DMP design presented in Chapter 3 in two ways. First, RCDC implements a new deterministic execution algorithm, called DET-HB (for "happens-before"), which relaxes memory consistency while still supporting data-race-free-based memory models (e.g., those of Java and C++). This improves performance and scalability by requiring fewer costly fences, which leads to less serialization. DET-HB does not employ speculation and does not sacrifice determinism in the presence of races.

Second, RCDC uses a lower complexity hybrid hardware/software implementation in which the hardware provides only two simple mechanisms, software-controlled store buffering and instruction counting, leaving the rest of the implementation to software. Implementing store buffering in hardware has the pleasant side effect of reducing the effects of false sharing. RCDC can be implemented on a commodity multiprocessor architecture and does not interfere with software (e.g., the OS) that does not choose to use it.

4.1 Relaxed-Consistency Deterministic Execution

This section presents our new deterministic execution algorithm, DET-HB. For expository purposes, we first describe the DET-TSO deterministic execution algorithm [34], which DET-HB builds upon.

4.1.1 DET-TSO: Store Buffering

Isolating threads is a core component of providing deterministic execution. Store buffers are a common and useful mechanism for achieving this isolation. In this approach, each quantum round is divided into three modes, a *parallel mode*, a *commit mode*, and a *serial mode*. During parallel mode, all stores are buffered in a thread-local store buffer, giving each thread a private view of shared memory. After parallel mode, all threads enter a commit mode in which the local store buffers are published to the global memory space. This commit happens deterministically. The effect is a serial commit order, but the implementation uses parallelism to avoid a sequential bottleneck. After commit mode is a short serial mode in which threads execute in a deterministic serial order and operate on shared memory directly. Serial mode is used to execute atomic synchronization operations, as described below.

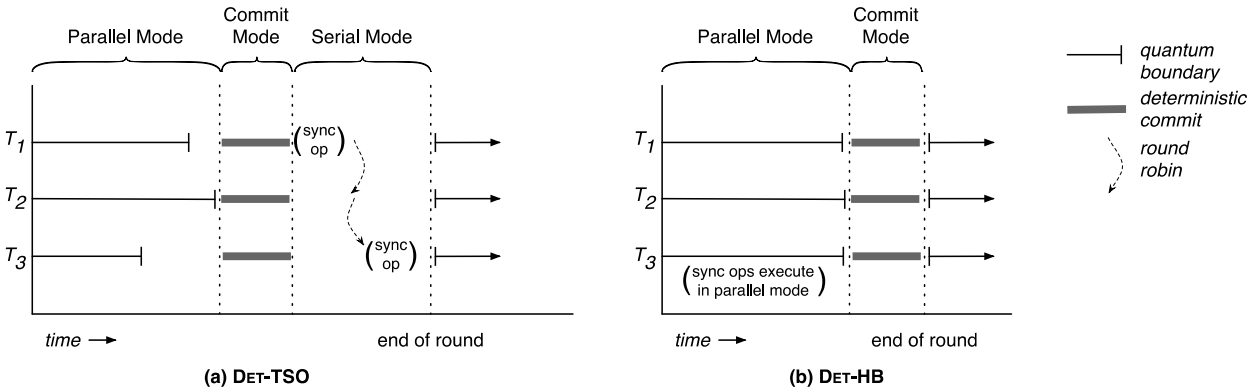

Figure 19: Timeline of a quantum round in DET-TSO and DET-HB, showing the division of each round into parallel, serial, and commit modes. DET-HB improves upon DET-TSO by allowing synchronization to happen in parallel mode, eliminating the need for serial mode.

Figure 19a illustrates one round of execution in DET-TSO. Each thread executes one quantum per round, where, as in DET-SERIAL, a quantum is some deterministic number of instructions. DET-TSO is deterministic due to four properties: (1) quantum lengths are deterministic; (2) threads are isolated in parallel mode, preventing nondeterministic interference from other threads; (3) commit mode ensures that writes to shared memory happen in a deterministic order; and (4) serial mode

ensures that atomic synchronization happens in a deterministic order. Note that the deterministic guarantee offered by DET-TSO does not depend on a race-free assumption – data races are resolved deterministically as a result of the isolation provided by parallel mode, combined with the deterministic order on writes provided by commit mode.

Notice that execution under DET-TSO is not sequentially consistent. Stores are not globally visible until commit mode, effectively reordering them after loads in the same quantum. This reordering breaks atomic operations like compare-and-swap (CAS), as a CAS is an atomic read-write pair but DET-TSO will reorder the write to commit mode, breaking atomicity. This issue reveals the need for serial mode – during serial mode, atomic operations execute atomically and deterministically. Further, it reveals the need to define the semantics of a memory fence. In DET-TSO, thread T ends its parallel mode when it reaches a memory fence. This flushes T's local store buffer, implementing the semantics of a full memory fence. Because DET-TSO does not distinguish between different types of memory fences, it implements the total-store-order (TSO) memory model.

DET-TSO achieves high performance when serial mode is empty and parallel mode is *balanced*, meaning that all quanta in a round execute in the same amount of real time. Serial mode is empty when synchronization is rare; prior work [34] has shown how to use instruction counting to achieve balanced parallel modes when serial mode is empty. When synchronization does happen, it forces DET-TSO into serial mode, whereby every synchronization operation causes global coordination. Synchronization not only causes serialization but also *imbalance* in parallel mode, which results in additional lost parallelism due to excess waiting. When synchronization is frequent, the effects of serialization and imbalance dominate and performance suffers.

4.1.2 DET-HB: Leveraging Data-Race-Free Memory Models

DET-HB addresses the major weakness of DET-TSO: synchronization. Like DET-TSO, DET-HB uses deterministic store buffers and divides execution into quantum rounds with parallel modes and commit modes. However, DET-HB introduces a new approach to deterministic synchronization that improves on DET-TSO in two respects. First, DET-HB implements a data-race-free [157] (DRF)

relaxed memory model based on the *happens-before* relation between threads (hence "DET-HB"). This model requires fewer memory fences than TSO, which makes parallel mode less likely to end early, thus increasing parallelism. However, weakening consistency alone does not remove all impediments to scalability, as DET-TSO requires that synchronization execute in a globally serialized fashion. To remove this further bottleneck, DET-HB eliminates the need for an explicit serial mode by using the Kendo algorithm [23] to execute synchronization directly in parallel mode, while still providing determinism even for programs with data races. Overall, these optimizations let DET-HB execute with less serialization and less imbalance than DET-TSO, leading to improved parallelism as illustrated in Figure 19b.

The key observation of DET-HB is that language-level memory models have weaker consistency guarantees than TSO. Specifically, Java [9] and C++ [10] define consistency models based on the data-race-free model [157]. From the programmer's perspective, it does not matter that the execution layer (e.g., the hardware) provides TSO when other layers of the system (e.g., the compiler) guarantee only DRF. Further, the need to precisely control memory visibility causes nondeterministic processor-local fences to become global operations in deterministic systems like DET-TSO, which suggests that strong memory ordering has a much higher cost in deterministic systems than in nondeterministic systems. Both these observations imply that deterministic systems should relax consistency as much as possible. As DRF-based models are specified by high-level languages, they represent the limit to which memory ordering can be relaxed.

4.1.2.1 Synchronization in DET-HB

As DET-TSO is a deterministic version of a TSO consistency model, DET-HB is a deterministic version of a DRF consistency model. DET-HB differs from DET-TSO in its approach to synchronization. The rest of this section presents the basic ideas of DET-HB. We describe details of the DET-HB synchronization algorithm along with our synchronization library in Section 4.3.4.

Consider mutex locks in a language with a DRF-like model, such as Java or C++. In these languages, the visibility of stores is guaranteed only along happens-before edges, which can arise from

program order between consecutive operations in a thread or from synchronization operations across threads. When thread T acquires lock L, this creates a happens-before edge E from the previous releaser of L to T. The DRF model guarantees that from this point forward, T will see stores that transitively happen-before its acquire of L. Other stores need not be visible. Therefore, in DRF models, T needs a memory fence after acquiring lock L only when happens-before edge E is not redundant. When E is redundant, the fence can be elided.

DET-HB exploits two happens-before redundancies: (1) thread-local edges, and (2) cross-quantum edges. First consider thread-local redundancies: if T was the previous releaser of L, then lock L has not been handed off to another thread, and we say that happens-before edge E is *local* to thread T. A fence is not needed in this case because edge E is redundant with program order. Prior work has had this same insight but in the context of nondeterministic systems, and furthermore has shown that lock locality is very common in Java programs [30, 33].

Cross-quantum redundancies are more interesting. They follow from the observation that all quanta in round N are connected to all quanta in round N + 1 by implicit happens-before edges. These implicit edges arise from the bulk-synchronous style of execution used by DET-HB, illustrated in Figure 19b. The important result is that an explicit fence is not necessary when synchronization is separated by a quantum boundary. Thus, by matching quantum length with the frequency of synchronization, DET-HB can eliminate many unnecessary fences, increasing performance and scalability.

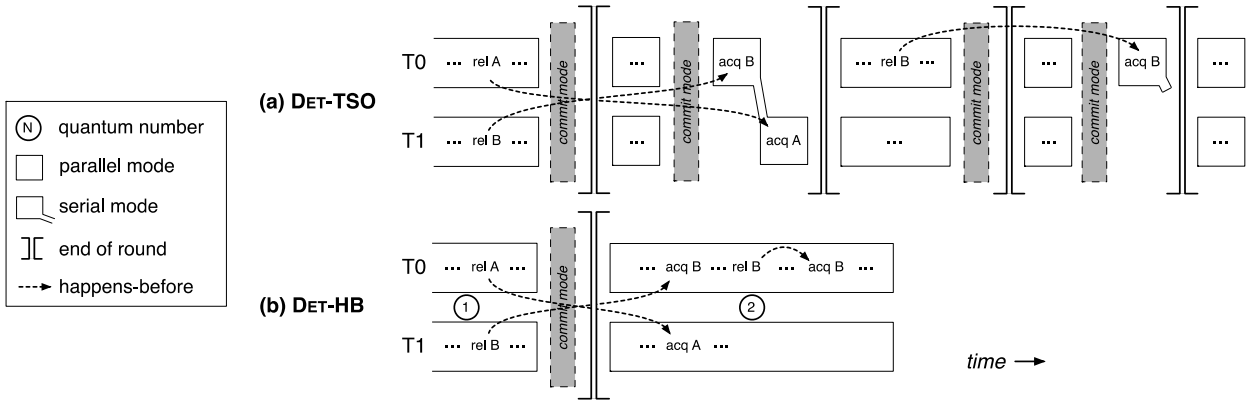

Figure 20: A comparison of execution under DET-HB with execution under DET-TSO, showing how DET-HB extracts more parallelism from programs with frequent synchronization.

Figure 20b demonstrates both the above redundancies. An example of a redundant cross-quantum edge is shown when thread T0 acquires lock B in quantum 2: this creates a happens-before edge with the release of lock B by thread T1 in quantum 1. Because this edge crosses a quantum boundary (i.e., it crosses a *commit mode*), T0 does not need to execute an explicit fence when acquiring lock B. In contrast, note that under DET-TSO, T0 must execute a fence, i.e., end its quantum, before acquiring lock B.

An example of a thread-local redundancy is also shown in quantum 2, where T0 reacquires lock B. As T0's updates are automatically visible to itself, there is no need for a fence. The extra serialization necessary to enforce the stronger TSO memory model is illustrated in Figure 20a.

Further, Figure 20b shows that DET-HB does not require a serial mode, in contrast to DET-TSO, which executes all lock acquires in serial mode. Even with the weaker DRF memory model, serializing all synchronization eliminates the ability to exploit thread-local redundant fences. Recall that DET-TSO uses a serial mode to guarantee both atomicity and a deterministic order of synchronization. For correctness and determinism, DET-HB must make these same two guarantees. Our solution is to use the Kendo algorithm [26] to impose a deterministic total order on all synchronization within a single quantum round. This algorithm allows synchronization to operate directly on the global memory space, bypassing the store buffer so the operation happens atomically. We describe this algorithm along with our synchronization library in Section 4.3.4.

4.1.2.2 Language Memory Models

Even though DRF does not specify the semantics of races, DET-HB's deterministic guarantees hold even for programs with data races. DET-HB's DRF memory model naturally matches the C++ memory model; however, the Java memory model specifies some behavior for data races, e.g., to prevent "out-of-thin-air" values. DET-HB does not itself introduce any potential "out-of-thin-air" values because it does not employ any form of speculation. A Java compiler must (still) ensure that its optimizations do not violate the Java memory model and also must ensure that proper synchronization and fences are inserted. Therefore, compiling Java code for DET-HB's memory model is no more complex than compiling for other weakly ordered architectures.

4.2 RCDC System Overview

RCDC provides an efficient implementation of DET-HB through a combination of hardware and software mechanisms, as summarized in Figure 21. The four main components of RCDC are (1) a precise instruction-count mechanism to divide each thread's instruction stream into balanced quanta efficiently, (2) a store-buffer mechanism that allows threads to execute in isolation from other threads, (3) a deterministic commit mechanism that concludes each quantum round, and (4) a custom synchronization library that implements a pthreads interface while enforcing DET-HB's memory-consistency model. These components are implemented as a combination of hardware and software designed for maximal flexibility and minimal hardware complexity.

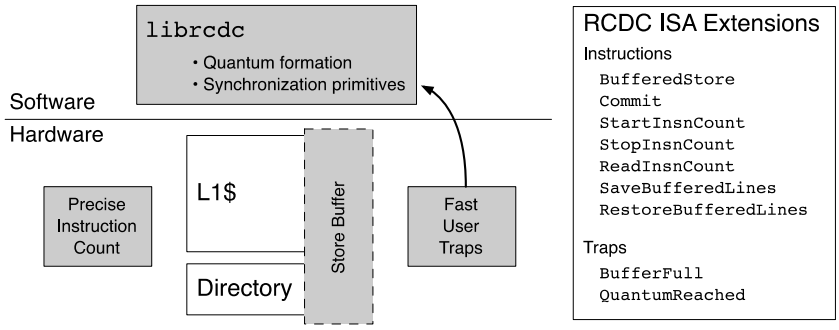

Figure 21: RCDC system overview, showing the division of responsibility between hardware and software. The shaded boxes show RCDC's additions.

Quantum formation is an ideal use case for hardware, as counting instructions involves nearly zero overhead in hardware but causes substantial slowdown in software. The hardware instruction-counting mechanism simply counts instructions as they retire, triggering a user-level

`QuantumReached` trap when a pre-defined total is reached. This trap is responsible for actually starting the commit process that makes buffered data visible.

Our synchronization library requires the ability to disable and enable instruction counting for the local processor, and also to read the current instruction counts of remote processors. For this purpose the instruction-count mechanism can be controlled and queried via the `StopInsnCount`, `StartInsnCount`, and `ReadInsnCount` instructions.

Deterministic quantum formation beyond simply counting instructions is also possible. Section 4.3.1 describes a more advanced strategy that uses opcodes and store buffer hit/miss information to construct quanta with better balance.

The **store buffer** mechanism is also a natural fit for hardware, where processor-private caches can isolate an executing thread from other threads in the system (like in hardware transactional memory, but without an abort mechanism) with additional bits of cache line state. We rely on simple compiler modifications or binary rewriting to replace existing store instructions with our new `BufferedStore` instruction. With more sophisticated analysis, ordinary non-buffered store instructions can, without any loss of determinism, replace buffered stores to locations that are provably thread-private. This increases the effective capacity of the store buffer without additional hardware resources.

The **deterministic commit** mechanism is triggered by software, via a new `Commit` instruction. The actual commit process is implemented in hardware, as described in Section 4.3.3. The commit process is invoked by software in response to `QuantumReached` and `BufferFull` traps, as well as to enforce the memory consistency requirements of DET-HB. Hardware triggers a `BufferFull` trap immediately when a store buffer overflows. Note that our cache replacement policy, described in Section 4.3.2.1, is designed to ensure that store buffer overflows happen deterministically.

Finally, our custom **synchronization library** acts as a drop-in replacement for pthreads. It uses the instructions described above to enforce DET-HB's consistency model. Because the decision of *when*

to commit is left to software, our synchronization library can easily be modified to implement other consistency models, e.g., DET-TSO.

4.3 Implementation

We now discuss implementation details for the major hardware and software components described in Section 4.2.

4.3.1 Quantum Formation

For quantum formation, the hardware does instruction counting and sets a trap after a software-defined quantum budget is exhausted. This mechanism is initialized by system software when a process is created. This code registers a user-level trap that is invoked whenever the quantum size is reached, establishes the size of quanta, and executes the `StartInsnCount` instruction.

Thus far, we have discussed quantum formation in terms of counting instructions. However, it is possible to provide better quantum balance by giving instructions non-uniform weights, e.g., based on opcode. One useful optimization leverages the deterministic contents of the store buffer. Memory accesses deterministically either hit in the store buffer (i.e., hit to a written cache line) or miss the store buffer (which means either hitting to an unwritten cache line or missing to the next-level cache or beyond). Knowing whether the memory operation is a load or a store lets us, in many cases, accurately assess the latency of that operation deterministically. Loads that miss the store buffer are often hits to a clean line, so we assign them a low weight. But stores that miss in the store buffer tend to be cache misses, and thus have high latency. Assigning higher weights to such stores results in better quantum balance since the weight assigned to each instruction better approximates its actual latency. Finally, we add to the sum of instruction weights as instructions retire, allowing access to hit/miss information, and also avoiding any issues with wrong-path instructions.

4.3.2 Buffering

RCDC's hardware provides support for buffering data, while software (i.e., the compiler) controls what data is buffered via the `BufferedStore` instruction. This section details how buffering support is implemented as an extension to the cache hardware.

Cache-based data buffering imposes a few system requirements: (1) buffered lines cannot be provided to remote requests; (2) the commit protocol, which makes buffered data available to all processors, needs to be deterministic; and (3) the system needs to support context switches. RCDC provides this functionality on top of a conventional directory-based MOESI cache coherence protocol and implements buffering in private caches, while still naturally supporting shared caches. For simplicity of explanation, we consider only a single private L1 cache per processor in the discussion below.

4.3.2.1 Cache Extensions for Store Buffering

Each L1 cache line is extended with a write-mask, which has as many bits as bytes in the cache line. When a `BufferedStore` instruction is executed, the corresponding write-mask bits are set. Consequently, lines with non-null write-masks contain buffered data.

To ensure that store buffer capacity is exhausted deterministically, we modify the cache eviction policy to always preferentially evict unwritten cache lines from a set. This ensures determinism while maximizing the amount of progress a processor can make before running out of store buffer capacity. When all lines of a cache set are buffered and an eviction needs to happen, RCDC triggers a `BufferFull` trap and the runtime system ends the quantum.

Non-buffered stores to cache lines in the non-buffered state proceed normally, following the conventional MOESI protocol. Non-buffered stores to cache lines in the buffered state are treated like `BufferedStores`. Note that buffering data from a non-buffered store is valid with respect to the instruction semantics as it is always correct to buffer private data – it just will not bring any benefit. If a non-buffered store necessarily cannot be buffered because of program semantics, then software needs to guarantee that this does not happen (e.g., using careful memory layout).

4.3.2.2 Coherence Operations

We augment the transitions in a conventional MOESI protocol to handle our new `BufferedStore` instruction. This requires three changes to a conventional MOESI protocol. First, if an L1 cache receives a request for a line whose write-mask is non-null, the request is nacked. The requester then goes to a shared cache (or memory) to fulfill its request. This is necessary to guarantee that buffered data is never provided to remote requests. Second, a line must be in the Shared state before it can be written by a `BufferedStore` instruction. Finally, our commit protocol (Section 4.3.3) moves lines to the Owned state after they have been published. As a consequence of these last two changes, moving a line to the Shared state to satisfy a `BufferedStore` may require a write-back operation (e.g., because that line may have been buffered in the previous round).

Interestingly, isolating each thread's updates into separate store buffers also yields a solution to false sharing, by allowing threads to perform updates to the same cache line within a quantum round without any serialization via the coherence protocol. The line becomes temporarily incoherent, but the updates are merged deterministically at the end of the round. If threads' updates are in conflict (i.e., two threads update the same bytes), there is a data race in the user program – data-race free programs can never observe this relaxation of coherence. This approach is similar to delayed consistency [160] for nondeterministic multiprocessors.

4.3.2.3 Context Switches

The kernel can context switch away from and back to a thread at any time, even during parallel mode, as long as it invokes the `SaveBufferedLines` and `RestoreBufferedLines` instructions to save and restore a thread's current store buffer.

These instructions make use of a per-thread, in-memory data structure called the Buffered Data Table (BDT), which contains the saved store buffer contents for a given thread. A BDT has a row for each cache line in a processor's store buffer, with one column for the line's data, another for the line's write-mask, and a third for a "next" pointer whose use is described below. A row in the BDT is considered valid if its write-mask is non-null. The `SaveBufferedLines` instruction simply

flushes all buffered lines from the cache to the BDT. As it does so, it clears the write-masks of all buffered lines in the L1 cache and transitions them to the Invalid state, making them available for the next thread to be switched in. The `RestoreBufferedLines` instruction iterates over all buffered lines in a given BDT, restoring them into the L1 cache. After a line is restored from the BDT to the L1 cache, its write-mask is cleared in the BDT to signify that the line is no longer saved in-memory.

These two instructions additionally maintain a separate, per-process table called the Buffered Address Map (BAM). The BAM is a table of pointers mapping each line address to a linked list of BDT entries storing the in-memory versions of that line. The pointer in each BDT entry points to the next thread's BDT entry in the list. Each BDT entry represents the saved state of one buffered version of the given cache line. By walking the list, the BAM table can be used to enumerate all in-memory versions of a buffered line. BAMs are used during the commit process as described in the following section.

We highlight that these instructions can be expensive, not only on their own, but also because of the extra work they impose on the commit process. In Section 4.4 we describe a few kernel scheduling optimizations that make these instructions infrequent.

4.3.3 Committing Buffered Data

In DET-HB, the transition to commit mode is controlled by software, which uses the `Commit` instruction to initiate the actual commit process in hardware. The RCDC software runtime executes the following pseudocode for each thread when it reaches its quantum boundary, e.g., when its quantum budget has been exhausted:

```
1   end_quantum() {
2     global_barrier()
3     Commit
4     global_barrier()
5   }
```

The first barrier represents the transition from parallel mode to commit mode; the `Commit` instruction represents commit mode; and the second barrier represents the transition back to parallel mode to start the next quantum (see Figure 19b). The first barrier ensures that all threads are ready to commit, while the second barrier ensures that all threads have finished publishing the contents of their store buffers.

The goal of the commit process is to merge buffered data deterministically and publish it globally. If a line has no buffered data in any cache, commit has no effect on that line. When a line has buffered data in one or more caches, the commit process deterministically merges all buffered data and then publishes this data to the rest of the system by moving the merged line to the Owned state.

A processor executes the commit instruction by iterating over all lines in its cache that have buffered data, i.e., those lines with non-null write-masks. For each of those lines, the processor executes the commit protocol. The commit protocol coordinates with the directory and with other processors, collects all buffered versions of a line, and then deterministically merges them. Once the commit protocol has been executed for all of the processor's buffered lines, the `Commit` instruction retires. At this point, the processor's entire store buffer has been globally published.

The commit protocol needs to handle two cases: committing when all buffered versions of a line are in-cache; and committing when at least one buffered version of a line is out-of-cache (i.e., because the thread was context-switched out). We describe both cases in detail below.

4.3.3.1 In-cache Commit (all threads running)

The processor issues a commit message for the given line to the directory; the directory replies with an acknowledgment for commit and a list of sharers for that line. The processor sends a commit message to each sharer. Upon receipt of a commit message, each sharer returns a reply indicating if it has the line, and if so, it includes the write-mask and the data for the requested line, as well as its deterministic *order id*; it then clears its write-mask and moves the line to Invalid state. When the committing processor receives a reply, it merges the other processor's data into its own

line. Once the processor has collected replies from all sharers, it clears the line's write-mask and moves the line to the Owned state, making the line visible to all processors.

The merge algorithm takes lines from two processors, P0 and P1, and computes the result of P0's writes happening before P1, where P0 has the smaller order id. This algorithm is straightforward and has been described by prior work [34].

Note that if the directory nacks the request for commit of a given line, this implies that some other processor has already started the commit for that line; the requester then waits until it receives a commit request for the line from another processor. Also note that while the commit process can actually happen in any order, the final state is guaranteed to be deterministic because the merge process is deterministic. Moreover, when the sharers list includes only the committing processor, no merge is necessary; the processor simply clears the line's write-mask and moves the line to the Owned state.

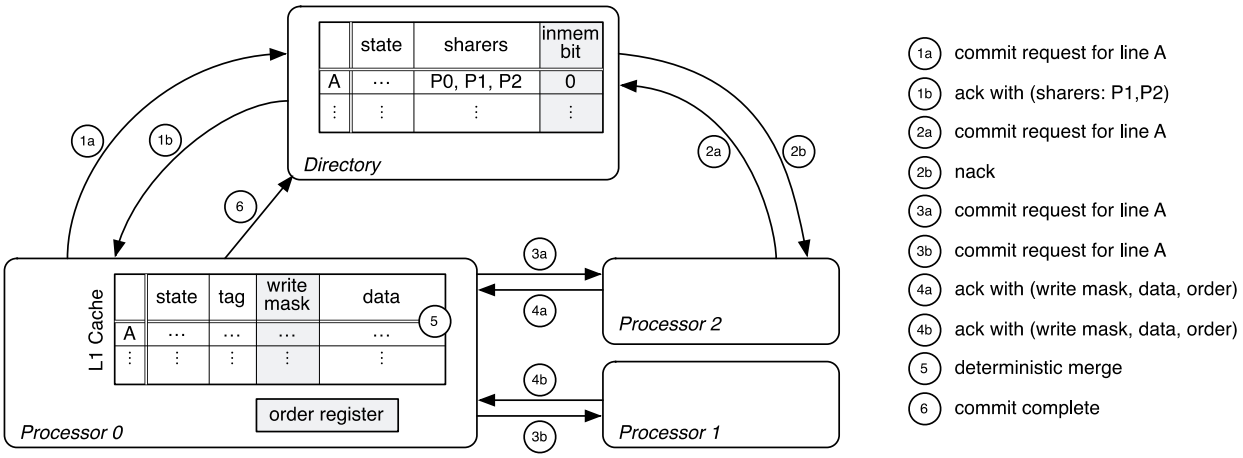

Figure 22: RCDC commit process when all application threads are scheduled. Shaded areas are RCDC additions.

Figure 22 illustrates how RCDC deals with multiple caches trying to commit to the same line A. Processors P0, P1, and P2 all have buffered copies of line A. First, P0 and P2 send concurrent commit requests for line A to the directory (1a, 2a). P0's message arrives first, and the directory responds to P0 with an acknowledgment message (1b), including the list of sharers, allowing P0 to proceed with committing line A. Since commit has started for line A, the directory responds to P2 with a negative acknowledgment (2b) and P2 waits for a commit request (which it is bound to get

since it is guaranteed that another processor is committing A). P0 continues by sending commit messages to all sharers of A (3a, 3b). P1 and P2 respond with a message containing their data, write mask, and deterministic order (4a, 4b) and then invalidate their copy of the line. Upon receiving the acks, P0 deterministically merges the data with its own (5) and notifies the directory that line A has been committed (6).

4.3.3.2 Out-of-cache Commit (at least one thread is switched out)

We now describe the more general case where at least one sharer has been switched out. RCDC supports this case with a simple extension to the directory: each directory entry, in addition to the sharers, also includes a single bit called the *in-memory bit*, which indicates if the line has data in an in-memory Buffered Data Table. This in-memory bit is set by the `SaveBufferedLines` instruction.

When a committing processor issues a commit message for a line to the directory, the directory replies with an acknowledgment and a list of sharers as before, and the processor communicates with the sharers as before. However, the directory also replies with the state of the in-memory bit. If the in-memory bit is set, the committing processor walks rows in Buffered Data Tables via the Buffered Address Map table to enumerate all in-memory versions of the line being committed. The processor merges these versions into its own line using the same algorithm as before, and then sends a commit-complete message to the directory. At this point the directory can clear the in-memory bit.

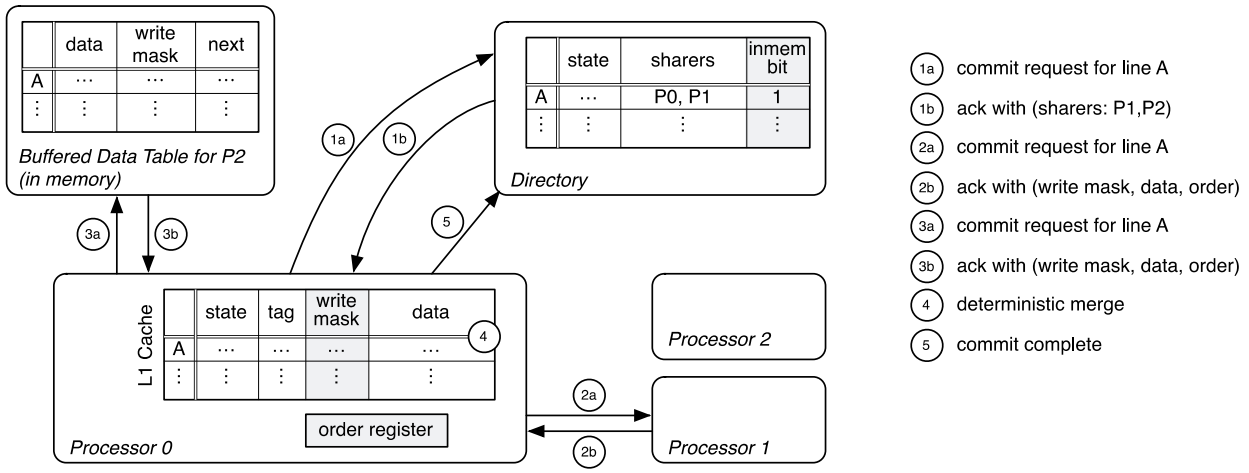

Figure 23: RCDC commit process when an application thread is switched out.

Note that commit still proceeds correctly even if the only thread that has a given line buffered is switched out, since that thread will invoke the `Commit` instruction when it is eventually switched in. (The barrier on line 4 of `end_quantum` ensures this.) Also, note that the directory serves as a serialization point for the cache line operations performed by the `Commit`, `SaveBufferedLines`, and `RestoreBufferedLines` instructions; this prevents races between the commit process and context switches, making it safe for the kernel to switch out a thread at any time without sacrificing determinism (note especially that a thread can be safely switched out between lines 2 and 3 of `end_quantum`).

Figure 23 illustrates how RCDC commits line A when a thread that has buffered line A has been switched out. P0, P1, and P2 all had buffered copies of line A. The thread on P2 is ready for commit but was switched out just before the commit process starts. The buffered data is saved in P2's Buffered Data Table in memory, and the in-memory bit set in the directory. P0 sends a commit message (1a) to the directory for line A. The directory replies with an acknowledgment (1b), including the list of sharers and the in-memory bit. P0 then sends a commit message for line A to P1 (2a), which replies with an acknowledgment (2b) before invalidating the copy of the line. At the same time, P0 accesses the Buffered Address Map to enumerate the list of in-memory lines, and notices that P2's BDT contains a copy of line A (3a). P2's saved line is found and returned to P0 (3b), and then removed from the Buffered Data Table. Next, P0 merges both received versions of line A

with its own version (4) and clears the write mask. Finally, P0 notifies the directory that the commit for line A is complete (5), and the directory resets line A's in-memory bit.

4.3.4 Synchronization Library

Our synchronization library is implemented using two basic building blocks: conditional memory fences and deterministic serialization. Conditional memory fences enforce DET-HB's memory model. A mechanism for deterministic serialization based on the Kendo algorithm [26] is used to execute synchronization during parallel mode of DET-HB.

Figure 24 shows our implementation of deterministic mutex locks. The `sync_acquire` and `sync_release` functions represent conditional memory fences, while `wait_for_turn` represents deterministic serialization. Other synchronization objects such as barriers, condition variables, and even lock-free data structures can be built from these same building blocks.

```
1   sync_acquire(o: SyncObject) {
2     if (o.quantum < curr_quantum ||
3         o.releaser == self) {
4       return // fence not necessary
5     }
6     end_quantum()
7   }
8
9   deterministic_lock(l: Lock) {
10    StopInsnCount
11    while (true) {
12      wait_for_turn()
13      if (CAS(l.locked, 0, 1))) {
14        sync_acquire(l)
15        StartInsnCount
16        return
17      }
18      end_quantum()
19    }
20  }
21
22  sync_release(o: SyncObject) {
23    o.quantum  = curr_quantum
24    o.releaser = self
25  }
26
27  deterministic_unlock(l: Lock) {
28    StopInsnCount
29    wait_for_turn()
30    sync_release(l)
31    l.locked = 0
32    StartInsnCount
33  }
```

Figure 24: Deterministic locking for DET-HB

4.3.4.1 Conditional Memory Fences

We use the functions `sync_acquire` and `sync_release` to implement deterministic lock and unlock just as traditional nondeterministic implementations of lock and unlock use acquire and release fences [158]. The key difference is the conditional on lines 2-3 of `sync_acquire`: when this conditional is true, the release-to-acquire happens-before edge is redundant and a fence can be elided. When this conditional is false, a fence is necessary: `end_quantum` is invoked, which

executes the `Commit` instruction. This conditional implements the observation noted earlier in Section 4.1.2.1: a fence is not necessary when the happens-before edge is local to a thread (line 3) or crosses a quantum boundary (line 2). Note also that when lines 2-5 are removed, `sync_acquire` is a full fence, and so the remaining algorithm implements a consistency model equivalent to TSO.

4.3.4.2 Deterministic Synchronization

We use the Kendo algorithm [26] to serialize synchronization deterministically. The basic idea is as follows: before performing synchronization, a thread T must wait for its *turn*, meaning it must wait until it has the global minimum instruction count (ties are broken by thread ID). While waiting for its turn, T must disable instruction counting by invoking `StopInsnCount`; this ensures deterministic instruction counting since T may have to wait a nondeterministic amount of time before its turn arrives. After synchronization is complete, T invokes `StartInsnCount`. The `wait_for_turn` function can be implemented by polling other threads' instruction counts via the `ReadInsnCount` instruction.[6]

Note that lines 13-14 and lines 30-31 execute atomically: `wait_for_turn` designates the beginning of an atomic region that is ended by `StartInsnCount`.[7] It is within these regions that the lock object is updated. For these updates to appear atomically, they must apply directly to the global memory space; i.e., all reads and updates of lock objects must bypass the store buffer. To ensure that lock objects are never buffered, lock objects can never exist on the same cache line as ordinary data; this introduces a partition of shared memory into lock objects and ordinary data.

4.4 System Issues

This section describes some of the issues that would be encountered when integrating RCDC with operating systems and legacy software.

---

[6] Alternatively, we could implement `wait_for_turn` via interprocessor interrupts rather than polling.

[7] Atomicity for the lock release is necessary to guarantee that concurrent releases of the same lock (e.g., due to programmer error) still result in a deterministic outcome.

4.4.1 Support for nondeterministic execution

The use of store buffers is a software choice. Therefore, programs can choose to execute nondeterministically. Kernel code, for example, would not need to be executed deterministically. One caveat is that our eviction policy risks monopolizing the cache. Recall that buffered lines are pinned in the cache; if a cache set fills with buffered lines, it cannot be reused until the store buffer has committed. This can accidentally prevent important systems code (e.g., context switch code) from running. We have two solutions. The first is to reserve a small victim buffer for non-buffered cache lines; and the second is to reserve just N-1 lines of a set for buffered data, where the cache uses N-way sets.

4.4.2 Processes

In RCDC, each process is by default its own determinism domain; in other words, threads within a process behave deterministically with respect to each other. Deterministic processes can run alongside nondeterministic processes. Moreover, if multiple processes share memory pages, the processes can be aggregated into a single determinism domain, much like the deterministic process group abstraction in dOS [145]. As long as different determinism domains do not share memory pages, the boundary of determinism domains can be defined completely by software without any extra hardware support.

4.4.3 Context Switches

To maintain determinism, RCDC requires that a thread's current instruction count and the contents of its store buffer be saved and restored across context switches. To reduce the amount of state that must be saved and restored, the OS kernel can be modified in two ways, described below.

First, the kernel can be modified to context switch away from a deterministic thread only at a quantum boundary, i.e., just after line 4 of `end_quantum` (Section 4.3.3). This eliminates the need to save and restore the contents of store buffers, since store buffers are always empty at a quantum boundary.

Additionally, if there are N CPUs but more than N threads in a given determinism domain, the kernel can schedule threads in groups of N per quanta, much like gang scheduling [161]. This considerably reduces the need to save and restore the contents of store buffers. It also can improve quantum balance, by eliminating the underutilization that occurs when N+1 threads must be scheduled per round, yet there are only N processors available.

4.4.4 Paging

It is important to make sure that none of the pages that have buffered data are paged out. The simplest way to provide this guarantee is to restrict paging so it happens only at the end of commit mode. In addition, the runtime system can provide the kernel with a list of pages that are provably unshared; these can be paged out at any time.

4.4.5 Memory Errors

As discussed in Section 4.3.4, lock objects must be partitioned from ordinary data. If this partition is broken by some memory operation, e.g., due to a memory error in a type unsafe language like C++, then that memory operation is a potential source of nondeterminism. For example, an errant read that happens to address a lock object will return a nondeterministic value, since that read can race with some other thread performing a lock acquire.

4.4.6 Store Buffer Parameters and Determinism

The parameters of the store buffer (i.e., the cache geometry) can affect quantum boundaries because buffer overflows cause a quantum to end. Thus, RCDC cannot guarantee the same deterministic execution will arise on two machines with different cache/store buffer configurations. One can address this potential issue by restricting store buffer usage such that its effective size is the same across different machines. The number of threads a program uses, and the parameters used to build quanta (e.g., size) are also implicit inputs that must be replicated to ensure repeatability.

4.5 Evaluation

The goals of our evaluation are to understand the effects of memory ordering relaxation on deterministic execution and to understand how RCDC's mechanisms behave dynamically. To these ends, we evaluate RCDC in two basic ways: (1) a hardware simulator of the DET-HB mechanisms called RCDC-HB, and (2) a software-only implementation of DET-HB using a compiler and runtime system called COREDET-HB. To measure the effectiveness of our consistency model optimizations we also implemented a TSO version of RCDC (termed RCDC-TSO) and used the TSO version of CoreDet from [34] (termed COREDET-TSO).

We built a hardware simulation infrastructure using the Intel Pin [155] binary instrumentation tool. The model focuses on the first order effects and includes RCDC's major components, including store buffering in private caches, quantum formation, committing, and consistency models for both DET-TSO and DET-HB. For the memory system, private 8-way 32KB L1 and private 8-way 256KB L2 caches for each core, with a 16-way 8MB shared L3. All caches have 64B lines. Instructions take 1 cycle to execute, and it takes an additional 1, 10, 35 and 120 cycles to access the L1, L2, L3 and main memory, respectively. We modeled 2, 4, 8 and 16 processor systems. With the exception of Figure 28, all workloads are run with a target quantum size of 50,000 instructions, except for ferret (25k), fluidanimate (1k) and streamcluster (1k). We determined these parameters by finding the best performance of our workloads, at 16 processors, for each quantum size in the range shown in Figure 28. Quantum commit costs 100 cycles. Error bars indicate the 95% confidence interval for the mean of 10 runs.

Our hardware simulations use version 2.1 of the PARSEC [154] benchmark suite. We used the simsmall input set for each workload. Due to excessive memory usage, we were not able to run the freqmine, raytrace and facesim workloads. Due to a lack of support for reader-writer locks and lock-free synchronization in our runtime system, we were not able to run the bodytrack and canneal workloads, respectively.

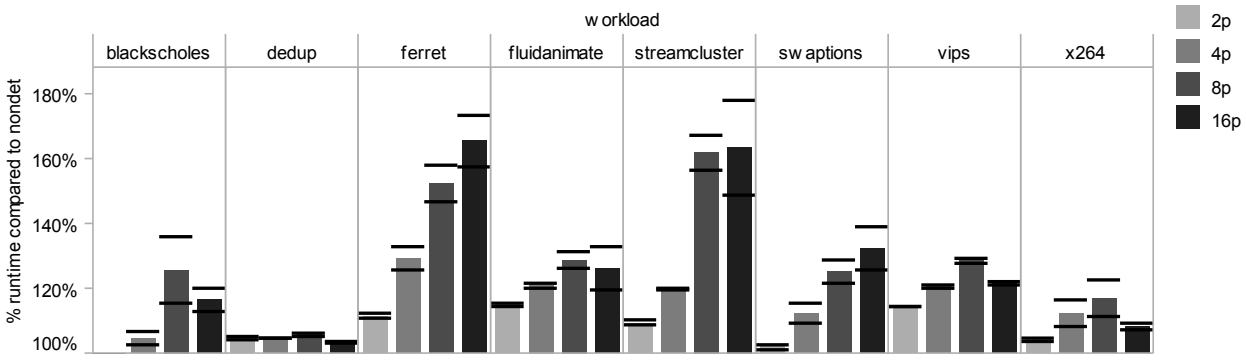

Figure 25: Performance of RCDC normalized to NONDET for 2, 4, 8 and 16 processors.

Our software-only implementation was built on top of the CoreDet [34] compiler and runtime system infrastructure. The source code for our simulator, modifications to CoreDet, and experimental data are available from http://sampa.cs.washington.edu.

4.5.1 Performance and Scalability

We start with a performance comparison of RCDC and the nondeterministic baseline (NONDET), as measured using our hardware simulator. Figure 25 plots performance of RCDC-HB for 2, 4, 8, and 16 processors normalized to NONDET with the same number of processors. Most applications suffer little performance degradation, but the overheads are still just over 60% in the worst case with 16 threads. Broadly, the performance costs in RCDC come from imbalance (periodic barriers at the end of parallel mode), extra stalls due to costly fences in synchronization operations, and the cost of committing buffered data. We characterize these costs more precisely below. Overall, RCDC provides fully deterministic execution for a modest runtime cost for many of our workloads.

From Figure 25 we can also see how RCDC's performance scales with additional cores. In a minority of cases (e.g., ferret), RCDC does not scale as well as NONDET. Most of the time, however, RCDC scales just as well as NONDET does, as evidenced by a consistent slowdown despite increasing core counts. Sometimes (e.g., vips) RCDC even closes the performance gap at higher core counts because the underlying benchmark does not scale well even with NONDET. Some of RCDC's overheads, like reduced cache capacity due to store buffering, can take advantage of additional parallel resources even when the underlying application cannot.

We also implemented a version of DET-TSO on top of RCDC (called RCDC-TSO) to assess the benefit of the extra memory reordering relaxation offered by DET-HB. Figure 26 compares the performance of RCDC-HB and RCDC-TSO, normalized to NONDET with the same number of processors. We include only the benchmarks ferret, fluidanimate, and vips; other benchmarks have less frequent synchronization, so the performance of RCDC-HB and RCDC-TSO is essentially identical. For these three benchmarks, RCDC-HB yields markedly better performance compared to RCDC-TSO, which comes from the fact that RCDC-HB is able to elide many costly fences (i.e., quantum boundaries) that RCDC-TSO cannot elide.

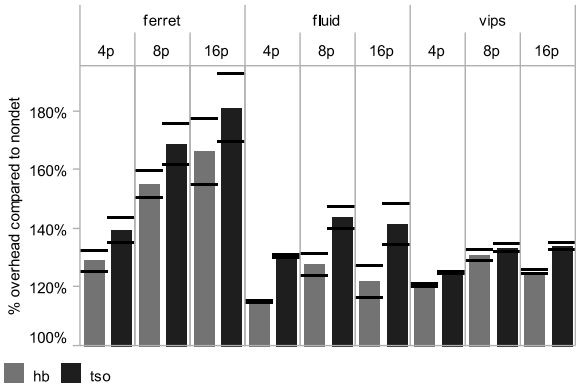

Figure 26: Performance of RCDC-HB and RCDC-TSO normalized to NONDET for 4, 8 and 16 processors.

4.5.2 Characterization

To better understand RCDC's behavior, Figure 27 breaks down the reasons for quantum boundaries. The three reasons a quantum can end are: *instruction count*, which is simply when a quantum has reached its maximum size; *store buffer overflows*, when the store buffer overflows and the thread cannot continue until its buffered data is committed; and *fences*, when a synchronization operation needs a memory fence to ensure the consistency model is upheld.

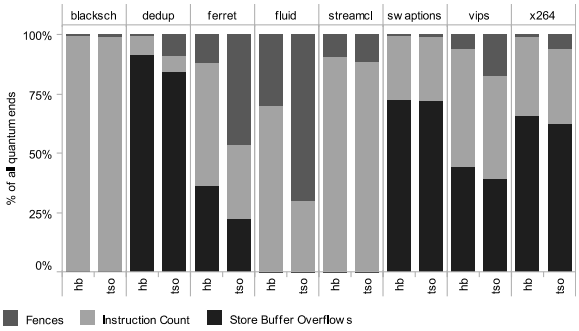

Figure 27: Reasons why quanta end for RCDC-HB and RCDC-TSO, for 16 processors.

Note that RCDC-HB has many fewer commits due to fences (the top segment of each bar) than RCDC-TSO. This quantifies the effect discussed in Section 4.1.2 (Figure 20), which is the essence of why RCDC-HB offers significantly better performance than RCDC-TSO.

Store buffer overflows are a frequent source of quantum imbalance for several workloads. While RCDC-HB is effective at reducing the number of fences, some of the premature quantum ends that

would have been a fence with RCDC-TSO are then replaced with store buffer overflows, which still result in quantum imbalance.

4.5.3 Sensitivity to Quantum Size

We end our RCDC evaluation with a characterization of how maximum quantum size affects performance. Figure 28 shows performance of ferret on a 16-processor RCDC system. The relationship between performance and quantum size can be highly non-linear: for ferret, larger quanta help smooth the effects of frequent quantum rounds, but beyond 25k instructions the extra imbalance of large quanta hurts performance. This effect was noticeable with both RCDC-HB and RCDC-TSO.

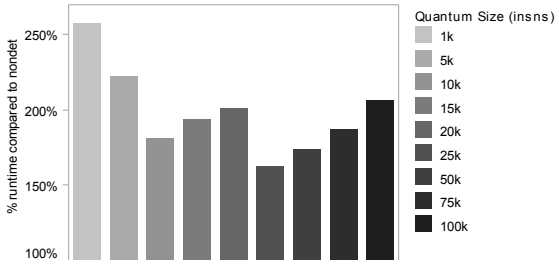

Figure 28: Performance of ferret with 16 processors using different quantum sizes.

4.5.4 Compiler-Runtime Implementation

In addition to the above hardware simulation, we implemented DET-HB in the CoreDet deterministic compiler and runtime system [34]. This implementation required changes only to CoreDet's synchronization library; the compiler and other parts of the runtime system were unmodified. We evaluated our CoreDet implementation using the PARSEC and SPLASH2 benchmark suites, and include a comparison of the performance of COREDET-HB with the performance of COREDET-TSO. For this evaluation, we enabled all of CoreDet's compiler optimizations.

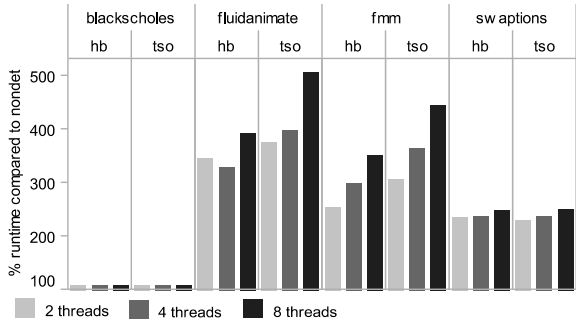

Figure 29: Performance of COREDET-HB and COREDET-TSO normalized to NONDET for 2, 4, and 8 threads.

Figure 29 summarizes this evaluation. The performance of COREDET-HB is largely the same as that of COREDET-TSO, with two exceptions: fluidanimate and fmm. Both these benchmarks have a relatively high frequency of synchronization. COREDET-HB's improved handling of synchronization

allows it to increase performance by about 20%: from 5x to 4x overhead for fluidanimate and from 4.5x to 3.5x overhead for fmm. This shows that the benefits of relaxed consistency determinism are not limited to hardware.

4.6 Conclusions

We have presented RCDC, a new deterministic multiprocessing architecture that leverages memory ordering relaxation to improve performance. We propose a new deterministic execution algorithm that combines deterministic synchronization with weak memory ordering to improve performance by reducing unnecessary stalls when enforcing determinism for arbitrary multithreaded programs. We also propose a hybrid hardware/software design that requires the hardware to provide only software-controlled store buffering and precise instruction counting, thereby reducing hardware complexity. Our results show that RCDC is competitive with nondeterministic multiprocessors, in terms of both absolute performance and scalability, without employing speculation. Moreover, our HW/SW approach allows precise control of when determinism should be enforced, providing flexibility to system software.

We believe this work is an important step toward realistic systems for the deterministic execution of arbitrary programs. Relaxed memory ordering aids performance by avoiding global barriers for synchronization operations while our HW/SW approach provides simplicity and flexibility.

Chapter 5 Merging Execution-level and Language-level Determinism

As discussed in Section 2.5, the proposals for ensuring determinism for parallel programs largely fall into two camps: execution-level determinism and language-level determinism. Execution-level techniques [22–24,26,34,145,146,149] enforce a deterministic, but still parallel, interleaving of memory operations at runtime. This necessitates some runtime overhead in exchange for determinism – providing determinism for arbitrary, potentially-racy programs can exact up to a 10x slowdown [34]. Language-level techniques [18–20,137] eschew runtime overheads by adopting a more restrictive programming model, such as pipeline [18] or fork-join [20] parallelism. For code that fits into such paradigms, determinism can be enforced by construction or via a static type system, which results in no runtime overhead.

Programmers wishing for the benefits of determinism are thus faced with a difficult trade-off: accepting the runtime overheads of execution-level determinism or rewriting their programs (if possible) to fit into a deterministic language's model. This chapter proposes a system called MELD that *m*erges *e*xecution-level and *l*anguage-level *d*eterminism. MELD employs execution-level determinism by default – to support arbitrary existing code – with a targeted application of deterministic language mechanisms to make the performance-critical part of an application fast.

5.1 Pitfalls of Integrating Execution-Level and Language-Level Determinism

MELD ensures the deterministic integration of deterministic languages within an execution-level determinism system. MELD uses runtime checks and a lightweight data-centric qualifier system for Java that allows a program's data to be partitioned at fine-grain between static and dynamic determinism enforcement schemes. Initially, one might think that to integrate a deterministic language into an execution-level determinism system it would suffice to call functions whose code was written in a deterministic language. The code of these functions would be verified by the deterministic language's machinery, which would take the place of the runtime instrumentation

used by the execution-level system. There are unfortunately several fundamental reasons why this simple approach would not work:

• *It violates the deterministic language's assumptions*: Deterministic languages make assumptions about aliasing and concurrency that do not hold if threads can make arbitrary simultaneous calls with arbitrary data into code written in a deterministic language.

• *It violates the deterministic execution system's assumptions*: To preserve determinism, returning from the deterministic-language call must happen at a deterministic point with respect to other threads. Doing so in a scalable way requires that we still instrument the code executing under language-level determinism (see Section 5.3.2).

• *It does not support real programs*: If the deterministic-language code can only access completely disjoint data from the deterministic execution code, then we cannot support the access patterns of many real programs. But it is crucial to distinguish data that is only accessed by a deterministic language as this is the key way that we improve performance.

Identifying and overcoming the problems with this naive approach is the motivation for the language design we describe. We present the key invariants necessary for using a deterministic language within a deterministic execution system while preserving determinism and improving performance. To the best of our knowledge, our work is the first to enable incorporating a deterministic language within a deterministic execution system.

5.2 Background

To evaluate the MELD system, we are building a prototype compiler that augments programs with a deterministic execution runtime system, and allows the integration of code written in a deterministic language. There are many possible choices for these two components, and we believe our approach generalizes to other combinations of execution-level and language-level determinism systems. We have based our Java compiler on the CoreDet deterministic compiler for C/C++ ([34] and Section 4.1.1) because of our familiarity with its code and because it provides execution-level

determinism for arbitrary parallel programs. We chose Deterministic Parallel Java [20] as the deterministic language because it is also open-source and provides statically-enforced determinism for Java.

MELD's **execution-level determinism system** is built on the CoreDet compiler and runtime system [34]. We described CoreDet more fully in Section 4.1.1 and include a brief recapitulation here for convenience. We build upon the RCDC version of CoreDet ([26] and Chapter 4) that extends the original work with additional memory consistency optimizations. CoreDet enforces determinism by isolating each program thread with its own private copy of memory, effectively turning a multithreaded program into a series of single-threaded programs (the "parallel mode" of Figure 19). The chunk of instructions executed by each thread is called a *quantum;* determinism follows from having quanta of a deterministic size (typically constant). Building quanta in this way requires instrumenting a program to count instructions; CoreDet counts instructions in a deterministic but approximate manner to keep overheads low.

Because threads need to communicate with one another, isolation among threads is broken periodically when all threads have finished their quanta and the accumulated updates of each thread are made visible to all other threads during "commit mode." Values produced by a thread are visible only to the thread itself until commit mode makes them globally visible. Thread isolation is implemented with per-thread store buffers: hash tables mapping addresses to fixed-size chunks of memory. Finally, a *quantum round* consists of the execution of one quantum by each thread during parallel mode and the subsequent commit mode. A series of quantum rounds allows an arbitrary computation to execute deterministically. Synchronization (memory ordering) may force quanta to end early; for this and other implementation details like thread creation and I/O please refer to Chapter 4.

MELD's **deterministic language** is DPJ, a region-based type and effect system for Java. We use the version of DPJ described in [20], and do not leverage the nondeterministic extensions described subsequently [140]. DPJ enforces determinism by allowing a programmer to carve up the heap into

named regions. Regions can be very fine-grained, at the level of a single object field or array index. Regions can also be hierarchical, which allows tree-based data structures and recursive array decompositions to be soundly expressed. Code that reads or writes regions is summarized in the type system via read and write effects, respectively, added to method signatures. These effects can be written by the programmer (though they are statically checked) and in many cases inferred [141]. Parallelism is supported via parallel for-loops (`foreach` statements) and parallel statement execution (`cobegin` blocks). A compile-time analysis checks the effects of concurrent operations for non-interference: if there are two concurrent operations on the same region, and at least one is a write effect, the program will not type check. DPJ programs that do type check are guaranteed to have the semantics of a deterministic serial execution of the program where parallel for-loops and coroutines are executed sequentially.

Figure 30 shows a simple DPJ program with two regions and effects. The variables `x` and `y` are placed in regions `RX` and `RY`, respectively. The method `setX` writes to variable `x`, which is in region `RX`, and therefore `setX` has the effect `writes RX`. A similar situation arises for `getY`.

```
class C {
  region RX, RY;
  int x in RX;
  int y in RY;

  void setX(int i) writes RX { x = i; }
  int getY() reads RY { return y; }

  void do() {
    cobegin { setX(); getY(); }
  }
}
```

Figure 30: A simple DPJ program with regions and effects. Underlined code indicates new syntax added for DPJ.

Since regions `RX` and `RY` are distinct, DPJ can prove that the statements of `do`'s `cobegin` block can be parallelized safely and retain the semantics of running sequentially. Effects must be specified for every method that is called, even via other methods, from inside a parallel task, as `setX` and `getY` are. The `do` method, however, is not called inside a parallel task and thus needs no effect annotation (see Section 5.4.2). If `do` were to be called inside a parallel task, it would need the effect `writes RX, reads RY` to reflect its own effects (none) as well as the effects of its callees.

CoreDet and DPJ have complementary strengths. CoreDet performs an alias analysis that is able to remove runtime instrumentation from simple uses of thread-private data, but DPJ's programmer-driven effect system is much more powerful. DPJ's parallelism constructs are limited to fork and

join, while CoreDet supports all pthread synchronization. MELD combines these systems to form a deterministic system that is both fast and general.

Not all weaknesses can be complemented away, however. To boost program throughput, CoreDet's quantum formation (but not store buffering) is enabled for *all* code, even code written in a deterministic language. Moreover, due to determinism's noncomposable nature, data managed by CoreDet and then passed to a DPJ function cannot recover DPJ's sequential semantics, though the data can soundly be computed upon without store buffer instrumentation while preserving determinism. The practical ramification for MELD is that while the entire program is guaranteed to be deterministic, it carries the guarantees provided by an execution-level determinism system, which is weaker than those provided by a language-level approach. A parallel program written entirely in a deterministic language has sequential semantics, ensuring that differing numbers of threads at runtime cannot affect the outcome of the program – the parallelism is implicit and invisible. For execution-level techniques, however, thread count is part of program input that must be explicitly tested – running a program with a different number of threads may expose new interleavings and new program behaviors, albeit in a deterministic manner.

5.3 Combining Execution-level and Language-Level Determinism

To integrate execution-level and language-level determinism in a single program without compromising determinism guarantees, some amount of isolation must be enforced between them. This section discusses the isolation that code written in deterministic languages implicitly requires in order to be soundly incorporated into a larger program. Section 5.4 discusses the MELD type system that statically enforces this isolation.

5.3.1 Starting Simple: Pure Language-Level Determinism

A program written entirely in DPJ is both deterministic and data-race-free [20]. All concurrent accesses to shared memory (the relative timing of which is a primary cause of nondeterminism) are proven non-interfering at compile-time. While this approach offers determinism with no runtime

overhead, DPJ programs cannot express many useful synchronization idioms and data sharing patterns.

5.3.2 Supporting Concurrent Conflicting Tasks

A natural generalization from a pure-DPJ approach is to allow conflicting concurrent tasks, while maintaining determinism (unlike in [140] which supports conflicting concurrent tasks at the cost of nondeterminism). We will use the CoreDet execution-level determinism system to handle conflicts deterministically. Consider, without loss of generality, a DPJ program that has a conflict on only one memory location (Figure 31).

```
class C {
  region R;
  int M in R = 0;
  float F in R = 0.0;
  main() writes R {
    cobegin { M++; M++ }
  }
}
```

Figure 31: A simple DPJ program with concurrent conflicting tasks.

To ensure the determinism of this program, we want CoreDet to manage reads and writes to `M`. Other locations (e.g., `F`) are ignored by CoreDet. We partition memory locations into two classes: those that are possibly subject to concurrent conflicts are labeled `exdet`, as they are managed by *ex*ecution-level *det*erminism, and all other memory locations are labeled as `langdet`, as they are managed by *lang*uage-level *det*erminism. DPJ ignores `exdet` memory locations for the purposes of verifying interference-freedom, and CoreDet does not generate code to enforce determinism for accesses to `langdet` memory locations. As with regular DPJ, all `langdet` locations must be placed in some region and a method `M` called, directly or indirectly, from a parallel task that touches `langdet` data must have effect annotations (see Figure 30 and Section 5.4.2). Methods that touch only `exdet` data, however, require no effect annotations.

Removing `exdet` accesses from DPJ's analysis is conceptually straightforward. However, CoreDet, along with all other execution-level determinism systems, requires a notion of deterministic logical time to coordinate updates to shared memory. CoreDet uses instruction counting for this purpose. CoreDet's main source of overhead is its thread-isolation mechanism – a per-thread *store buffer* implemented as a hash table – though quantum formation has noticeable overhead as well.

With MELD, we must instrument the program so that each thread maintains its dynamic instruction count and uses that count to form quanta. We must instrument all parts of the program to keep quanta *balanced* – as uniformly sized as possible to eliminate excessive waiting at quantum boundaries. However, we only need CoreDet's heavyweight store buffers for accesses to `exdet` memory locations. `langdet` locations can safely skip the store buffer and directly access global memory, reducing runtime overhead compared to CoreDet's pure-`exdet` solution.

5.3.3 Supporting Arbitrary Parallelism Constructs

We can incorporate the use of arbitrary parallelism constructs, like locks and atomic operations,[8] using the same mechanism we used to handle concurrent conflicting tasks. Each DPJ task can synchronize in complicated ways with other tasks, though DPJ cannot statically reason about this synchronization and so will find a large number of concurrent, conflicting accesses. In general, DPJ may deem every memory location to be subject to conflicting accesses. If we label all memory locations as `exdet`, then DPJ will ignore these conflicts and no parallel program will fail its type checking. This is precisely the CoreDet approach, providing generality at a significant runtime cost.

By allowing programs to contain a mix of `langdet` and `exdet` data, MELD is a generalization of the pure-DPJ and pure-CoreDet approaches. Allowing such flexibility is not without its costs; as we saw earlier we instrument the entire program for quantum formation, even if `exdet` accesses are rare. To ensure that we compile accesses to a memory location M correctly we must assign a consistent label for M. Failing to do so would allow two threads to see M as both `exdet` and `langdet`, respectively (Figure 32). If `ld` and `ed` run concurrently, `ld`'s update to M will skip the store buffer and directly update memory. This update races with `ed`'s read of M, causing the print statement to generate nondeterministic output. Thus, we must ensure that M is either always treated as `langdet`, where DPJ will statically catch the data race, or `exdet`, where

```
int M = 0;

void ed() { // sees M as exdet
  print M;
}

void ld() { // sees M as langdet
  M = 2;
}

// nondeterministic
cobegin { ed(); ld(); }
```

Figure 32: Aliasing between `exdet` and `langdet` locations results in nondeterminism.

[8] Mapping thread fork/join is slightly more involved, and we defer the discussion to Section 5.5.4.

both accesses will use the store buffer and `ed`'s read will be safely isolated. Section 5.4 discusses implementing this simple labeling of memory locations to work with the Java language.

5.3.4 Supporting Casts and Modularity

While a fixed partition of memory locations into `langdet` and `exdet` is sound, it is inflexible. Moving data from one partition to the other entails copying each memory location involved, inducing extra programmer burden and runtime cost. We would like the ability to soundly cast memory locations between labels. We would also like the flexibility of using DPJ on small components within a larger program, e.g., a pure function that sorts an array in parallel (Figure 33). Our current approach requires the array A to be labeled `langdet` and that DPJ be able to prove non-interference for all accesses to A throughout the *entire* program.

```
<region R> void psort(int[]<R> A) writes R {
  // sort A in parallel, in-place
}
```

Figure 33: An internally-parallel, in-place sort function that we can easily write in DPJ.

These two goals of supporting casts and modular use of a deterministic language are highly related. We call each modular use of DPJ, e.g., to write `psort` in DPJ without taking the rest of the program into consideration, a *deterministic language component* or DLC. In our implementation a DLC is a function F and the transitive closure of functions F calls, but a DLC can be generalized to any lexically-scoped region of code and the code reachable from it. Previously we've implicitly treated the entire program as one DLC (with potentially many `exdet` locations). However, a program can consist of multiple DLCs as long as each satisfies DPJ's preconditions: that DPJ has full visibility into the threading and aliasing of the data it operates upon. These conditions are naturally fulfilled when an entire program is written in DPJ, but in our modular use we violate this "closed world" assumption. While DPJ statically guarantees non-interference among the threads it creates and memory it allocates, it cannot, of course, provide guarantees about other threads or memory that it is unaware of. To implement `psort` correctly as a DLC, we must ensure no other code operates on `psort`'s array concurrently. Two concurrent calls operating on the same array may interfere with one another, violating determinism and other correctness properties. To maintain determinism, we need to ensure non-interference between:

1. the threads internal to the DLC
2. different threads running outside the DLC
3. concurrent calls to a given DLC by different threads
4. different DLCs running in different threads
5. a DLC and a thread running outside the DLC

Condition 1 is handled by DPJ and condition 2 by CoreDet. Conditions 3-5 are handled by a sound cast mechanism that allows transferring locations from `exdet` to `langdet`. The cast mechanism dynamically checks that, if we cast a memory location M from `exdet` to `langdet`, M is not accessed by anyone other than the casting thread – the cast effectively checks for correct privatization. Failures to privatize correctly are, thanks to determinism, repeatable across executions. Once M is correctly privatized, M satisfies DPJ's preconditions and can be safely passed to a DLC. As a performance optimization, we introduce a new label `xldet` to track memory locations subject to these casts to reduce the runtime overhead of the cast mechanism.

5.3.4.1 Casting from `exdet` to `langdet`

MELD's cast mechanism relies on dynamic checks. Casting a location M from `exdet` to `langdet` adds M to a "poison set." Once M is in the poison set, concurrent `exdet` accesses to M trigger a runtime error (Condition 5). Casts also trigger runtime errors for a concurrent cast of the same memory location, which allows us to support multiple DLCs executing simultaneously (Conditions 3 and 4). To detect concurrent writes, in commit mode threads check if they are updating a poisoned location. To detect concurrent reads, all reads of `exdet` locations also check if they access a poisoned location. These dynamic checks are not necessary in the baseline CoreDet scheme, but in the common case – an empty poison set – they add little overhead. Casts must execute at quantum boundaries for two reasons. First, casting at quantum boundaries ensures casts occur at a deterministic time, avoiding races with `exdet` accesses to the about-to-be-cast location that can result in nondeterministic runtime errors. Second, the quantum boundary also acts as a memory fence, ensuring that buffered `exdet` writes from the current thread are made visible to subsequent `langdet` reads by the same thread, as the `langdet` reads will skip the store buffer.

5.3.4.2 Casting back to `exdet`

Supporting only casts from `exdet` to `langdet` makes subsequent computation via an `exdet` alias impossible. Casting from `langdet` to `exdet` must be done carefully, however. Supporting arbitrary casting from `langdet` to `exdet` would require instrumenting all `langdet` accesses, to ensure that, after the cast, all accesses occurred with `exdet` instrumentation. If `langdet` aliases to E persist, nondeterminism can result because, as Figure 34 shows, writes through a `langdet` reference can race with writes performed through an `exdet` reference. Keeping `langdet` accesses streamlined is also crucial for performance, so MELD disallows general `langdet`-to-`exdet` casts. However, to be able to prune the poison set we would like to support `langdet`-to-`exdet` casts for a memory location E that was `exdet` to begin with.

```
region R;
exdet int[] E = new int[4];

// cast to langdet
langdet int[] L in R = (langdet int[]) E;

psort(L); // call DLC

// cast to exdet, langdet alias remains
E = (exdet int[]) L;

// nondeterministic result
cobegin { L[0] = 1; E[0] = 2; }
```

Figure 34: After casting a location from `langdet` to `exdet`, existing `langdet` aliases must not be used.

As a simple approach to control the aliasing of locations cast from `exdet` to `langdet`, we introduce a new `xldet` label and impose scope restrictions on its use. Instead of allowing casts from `exdet` to `langdet`, we only allow casts between `exdet` and `xldet`. `langdet` memory locations have no scope restrictions – they can be globally-visible static fields live for the entire execution of the program. `xldet` locations must be locally-scoped so that they do not outlive the DLC which computes on them. Assignment restrictions (Section 5.4.3) prevent `xldet` memory locations from escaping to the larger scope of `langdet` locations. Viewing the entire program as one large DLC reveals that `langdet` locations are in fact subject to the same scoping restrictions as `xldet`, but the scope is the life of the program. Note also that casts from `xldet` to `exdet` require no special consideration and are compiled as NOPs. During the scope of an `xldet` alias, dynamic checks prevent conflicting accesses via `exdet` aliases. After the `xldet` alias has gone out of scope, it is removed from the poison set and `exdet` aliases can be used again.

The local scope of `xldet` locations allows our compiler to automatically handle removing `xldet` locations from the poison set. These removals must occur at quantum boundaries to avoid nondeterministic runtime errors from conflicting `exdet` accesses, just as with `exdet`-to-`xldet` casting (Section 5.3.4.1).

5.3.4.3 Nested `xldet` locations

Supporting the nesting of `xldet` references requires one final refinement to our type system: uniquely identifying each `xldet` scope. Figure 35 shows how nondeterminism can result if `xldet` scopes are not uniquely identified. We begin with `inner`'s cast on line 5: this adds `ef` to the poison set, and since all `xldet` scopes are treated alike the assignment on line 6 type checks even though `arr` has `outer`'s larger scope and `ef` has `inner`'s smaller scope. At the end of `inner`, `ef` is removed from the poison set. However, because the `ef` reference has escaped `inner`'s scope, `outer` is able to write to one of `ef`'s fields via the array `arr` on line 12. As `outer`'s write is via an `xldet` reference, no store buffer instrumentation is performed. Concurrently, `bad` accesses `ef` via an `exdet` reference on line 17, and since `ef` is no longer in the poison set, this access succeeds and may nondeterministically see or fail to see `outer`'s write. The fact that `ef`'s removal from the poison set must occur at a quantum boundary does not solve this issue. Nothing prevents `bad` from performing enough computation to cause its conflicting access (line 17) to be in the same quantum round as `outer`'s access (line 12).

```
1   exdet Foo ef;
2   region R;
3
4   void inner(xldet Foo[] arr) {
5     xldet Foo xf in R = (xldet) ef;
6     arr[0] = xf; // assignment ok
7   }
8
9   void outer() {
10    xldet Foo[] arr in R = new Foo[1];
11    inner(arr);
12    arr[0].field = 1;
13  }
14
15  void bad() {
16    // ...
17    ef.field++;
18  }
19
20  void main() {
21    // nondeterministic!
22    cobegin {
23      outer();
24      bad();
25    }
26  }
```

Figure 35: If all `xldet` scopes are treated alike, assignment between different scopes can introduce nondeterminism.

Our solution to this problem in MELD is to ensure that `xldet` qualifiers are annotated with the scope in which they appear. Our solution is modeled on the region annotations used in the Cyclone language [162], but is simplified because 1) all `xldet` qualifiers are in local scope (Section 5.3.4.2) and 2) MELD does not support existential types.

For simplicity, programmers must explicitly write scope annotations on all `xldet` variables, though [162] has shown how scope annotations can often be inferred to reduce programmer burden. Class definitions can be parameterized by different scope variables, to allow each field of an instance to belong to a different scope. Similarly, methods are scope-polymorphic to allow them to take arguments of different scopes and to support recursion. Scopes are checked for equality based on their names, and only assignments between variables of equal scope are allowed. The scope annotations can be checked with an intraprocedural analysis that statically guarantees the absence of any assignment of a variable into a location of greater scope.

```
1   exdet Foo ef;
2   region R;
3
4   void inner<s>(xldet(s) Foo[] arr) {
5     xldet(inner) Foo xf in R = (xldet(inner)) ef;
6     arr[0] = xf; // assignment fails
7   }
8
9   void outer() {
10    xldet(outer) Foo[] arr in R = new Foo[1];
11    inner<outer>(arr);
12    arr[0].field = 1;
13  }
14
15  void bad() {
16    // ...
17    ef.field++;
18  }
19
20  void main() {
21    cobegin {
22      outer();
23      bad();
24    }
25  }
```

Figure 36: An updated version of the program from Figure 35, with each `xldet` location annotated with its scope. New code is underlined, with locals' scope annotations in parentheses and methods' polymorphic scope variables in brackets.

Returning to our code example, Figure 36 shows an updated version of the program from Figure 35 with each `xldet` qualifier annotated with its scope (in parentheses), and methods annotated with polymorphic scope variables (in brackets). These scope annotations allow our type checker to notice the mismatched scope assignment at line 6, as the two scopes have different names, and trigger a compile-time error.

Even though our scope annotation rule for checking assignments disallows them, assignments from locations of outer scope to locations of inner scope are sound. Because scopes are properly nested, identifying an outer scope is straightforward – when a new scope N is created, any scope S that already exists must be larger, and thus assignments from S to N are sound. Prior work has supported such assignments through a subtyping mechanism [162]. For simplicity, MELD conservatively disallows all inter-scope assignments.

5.4 The MELD Type System

The MELD type system uses type qualifiers to enforce a partition of a program's memory locations among the labels `exdet`, `xldet`, and `langdet`. MELD's qualifier system has been implemented for Java, but a previous prototype was implemented for C [27] and these ideas translate naturally into other languages as well. This section describes the semantics of our type qualifiers and the typing rules that enforce isolation at compile-time.

5.4.1 Type Qualifiers

In MELD, type qualifiers can be attached to any reference type appearing in a field definition, local variable, function parameter or generic type instantiations. Qualifiers are also permitted on static fields of primitive type (see below). A qualifier identifies the label for the storage of the object the reference points to. So `langdet Foo f` describes a reference `f` to an instance of type `Foo` where the memory locations comprising the instance's fields are all labeled with `langdet`. The instance may contain fields that are themselves references; the labels on these fields similarly apply to the storage for the objects they point to. The label for the reference `f` itself is specified by the object containing `f`, unless f is a local variable or static field (see below). To reduce the annotation burden on the programmer, `exdet` is the default qualifier and needn't be written explicitly.

The memory location holding a **local variable** doesn't need a label, because this location is always thread-private in Java. Both DPJ and CoreDet can safely ignore the memory locations used to hold local variables. Of course, local variables can still have qualifiers, labeling the memory locations pointed to by locals of reference type.

A **static field** `s` has no enclosing instance, references to which would label the memory location holding `s`. We handle static fields like `s` specially by allowing `s` to have two qualifiers – the first labeling the memory location of `s` itself, and the second labeling the memory locations of the instance `s` points to if `s` is of reference type.

References to the current object (`this`) are also handled specially. The qualifier on `this` identifies the label for the memory locations of the fields the `this` reference points to. As there is no natural way to add a qualifier to `this` we require a qualifier on methods stating which label for `this` they accept. To call the same method on both, say, `langdet` and `exdet` instances requires duplicating the method with a different name. Our approach avoids adding more complexity to Java's overloading rules.

A qualifier on a single-dimensional **array**, e.g., `langdet int[] L`, identifies the label for the memory locations storing the array elements. For multidimensional arrays a single qualifier identifies the label for the memory locations of all the array elements and sub-elements. E.g., `langdet int[][] L` means that `L[0]` points to an array of type `langdet int[]`, which in turn points to a `langdet` memory location holding an `int`.

5.4.2 Defaults

In MELD the default qualifier is exdet. This choice ensures that legacy programs will run deterministically without programmer intervention. The DPJ system that MELD builds on has its own set of defaults, however, that must be modified for compatibility with MELD. In DPJ, methods have a default effect annotation of "reads and writes the entire heap" – a safe approximation of what a programmer might write. DPJ's default makes all single-threaded code valid DPJ code: effects are only required for methods that can be called (directly or indirectly) from a concurrent context like `cobegin` or `foreach`.

In the context of MELD, DPJ's default effect is problematic, however. When we consider a call to a method M in a concurrent context, and if M's signature has no effects, we cannot distinguish whether 1) M is a method that touches only `exdet` data and thus needs no effects or 2) M touches

`langdet` data but the programmer has neglected to specify effects for M. In case 1), concurrent calls to M *are* deterministic because the execution-level deterministic runtime system ensures determinism even if M contains data races. In case 2), concurrent calls to M *are not necessarily* deterministic and DPJ should trigger a compile-time error, prompting the programmer to specify more precise effects for M. However, we cannot distinguish these cases without examining the code of M. Thus, in MELD we eliminate DPJ's default effect and instead require all methods that access `langdet` data to have an effect annotation, even if those methods are never called in a concurrent context. This requirement is easily enforced as part of MELD's type checking.

5.4.3 Type Rules

Assignments between memory locations of different labels are not allowed in MELD, except for explicit casts between `exdet` and `xldet`. All `xldet` qualifiers carry a scope annotation (see Section 5.3.4.3) and assignments between `xldet` qualifiers of different scopes are not allowed. Assignments from newly-allocated objects (the result of the `new` operator) are implicitly cast to the label of the receiving location. Implicit assignments, such as passing parameters in function calls and return values, are also subject to these restrictions. These assignment restrictions are straightforward to enforce using intra-procedural analysis.

To handle Java's generics, MELD permits qualifiers to be attached to type parameters. Type parameters with different MELD qualifiers are incompatible: e.g., a `langdet Foo` cannot be added to a `List<exdet Foo>`.

5.5 Implementation

The MELD compiler consists of three main components: 1) a type qualifier system, 2) the DPJ compiler, and 3) a Java-based compiler and runtime system based on CoreDet. While work has begun on these components, the MELD type qualifier system and modifications to the DPJ compiler are not yet complete.

5.5.1 Type Qualifier System

MELD's type qualifier system will be implemented in the Checker Framework [163], a framework for pluggable type checking in Java. MELD's qualifiers are naturally expressed as Java annotations (e.g., `@Langdet`) which allows them to be visible to many other Java-based tools. MELD's typing rules will be implemented as a checker for the Checker Framework. We will also use Java annotations to identify functions that are DLC entry points.

5.5.2 DPJ Compiler

To ensure that our `langdet` and `xldet` qualifiers are sound we must ensure that every `langdet`/`xldet` location is in fact statically checked by the DPJ compiler, and that all `exdet` locations are ignored. We will extend the DPJ compiler to take MELD qualifiers into account while performing its analysis. To support multiple DLCs, we will run the DPJ compiler iteratively over "slices" of the program. The first iteration will check `langdet` qualifiers, taking the whole program into account. Then, each DLC will be considered separately, checking the `xldet` locations within that DLC for non-interference. To simplify the DLC analysis, we parameterize the `xldet` qualifier with an argument identifying which DLC it belongs to.

Because DPJ extends the Java language with new syntax for type and effect annotations, standard Java tools will not work on DPJ code. Thus, our compiler flow will run the DPJ analysis first. The output of the DPJ compiler is standard Java source code; crucially, Java annotations are preserved through the DPJ compiler so that MELD type checking can follow.

5.5.3 MELD Compiler and Runtime System

While the CoreDet compiler and runtime system [34] worked for C/C++ programs, we have implemented a similar infrastructure in Java-based technologies to allow MELD to take advantage of the DPJ language. Our compiler is built on the Soot Java optimization framework [164]. The MELD compiler instruments a program with callbacks to the MELD runtime system. Memory accesses are instrumented for store buffering, and control-flow edges to perform instruction counting and quantum formation. The MELD runtime system itself is implemented as a Java library.

Store buffers are implemented as thread-private hash tables. To avoid unnecessary boxing each thread has a collection of hash tables: one for each Java primitive type and one for reference types. Instruction counting is performed by a callback on every basic block that counts the number of bytecodes executed.

Once the MELD type qualifier system is in place, the MELD compiler will use Java annotations on each memory location to elide store buffer instrumentation for `langdet` and `xldet` locations. `exdet`-to-`xldet` casts will also be implemented by the compiler, along with the automatic cleanup of `xldet` locations once their DLC is finished. This automatic cleanup mechanism requires a stack (LIFO) for each DLC. New `xldet` scope entries are pushed and popped whenever an `xldet` scope is entered and exited, respectively. On an `exdet`-to-`xldet` cast, a new element is added to the `xldet` scope entry at the top of the stack and the `exdet` location is added to the poison set. When exiting an `xldet` scope, all elements in the `xldet` scope entry at the top of the stack are removed from the poison set.

5.5.4 Handling thread fork/join

To allow the DPJ compiler to understand the parallel structure of a program, our prototype requires that a MELD program's parallelism be expressed in terms of DPJ's parallelism constructs: `cobegin` and `foreach`. Many multithreaded applications use threads in a straightforward way – creating a number at program launch, computing with them, and then joining with them before exiting. Such nested use of threads maps well to DPJ's nested parallelism constructs, e.g., each "thread" can be created via a function call in an iteration of a `foreach` loop. To support the full expressivity of threads, which can be forked and joined in a non-nested fashion, we propose the use of a static analysis, such as the Soot framework's [164] May-Happen-in-Parallel analysis, to conservatively identify what code may run in parallel with other code, and to feed this information to the DPJ compiler to ensure that methods that concurrently access `langdet` data are conflict-free. Recently-proposed extensions to DPJ incorporate support for non-nested parallelism [143], allowing all thread fork/join operations to be handled directly with DPJ primitives.

5.6 Extensions

This section describes some extensions to MELD that we view as promising future work: soundly handling nondeterminism and supporting qualifier polymorphism.

5.6.1 Incorporating Nondeterminism

We may wish to allow a certain amount of nondeterminism within our program, e.g., logging, network output or profiling code, for two reasons: 1) its nondeterminism will not have a large bearing on the determinism of the rest of the program and 2) such nondeterministic code can run without the overheads of the execution-level determinism system. To employ nondeterminism in a sound way, we need to formally guarantee that its effects are not allowed to "contaminate" the determinism of the rest of our program.

To soundly incorporate determinism and nondeterminism within a single program, we modify the type system described in Section 5.4 to become more like a standard static information-flow tracking type system [165]. The main extensions are 1) additional restrictions placed on scalar assignments, and 2) protection against implicit flows via control flow. These additions, combined with MELD's existing restrictions on assignments for reference types, suffice to prevent the nondeterministic part of the program from affecting the deterministic parts.

The nondeterministic part of the program (unlike integrating with deterministic languages) has no special requirements. To return to our list of correctness conditions from Section 5.3.4, we must additionally ensure non-interference between:

6. nondeterministic code and threads running inside a DLC
7. nondeterministic code and threads running outside a DLC

There is, of course, no requirement to isolate nondeterministic code from itself. For the purposes of integrating nondeterminism, it is sufficient to consider `exdet`, `xldet` and `langdet` as being equivalent: they all guarantee that a memory location is deterministic. If we consider these as a single "det" label, then we can use the type lattice det $\sqsubseteq$ nondet and run standard information-flow typing rules to support nondeterministic memory locations.

Casts that modify the qualifiers of nondet data (endorsements in the information-flow tracking literature) have a special semantic meaning: they represent a kind of "internal input" to the deterministic part of a program, analogous to external input read from, e.g., files or sockets. A record-and-replay system building upon MELD would perform logging at nondet endorsements to precisely capture this internal input and allow for repeatability of the deterministic portion of the program.

MELD's data-centric annotation approach is especially significant when nondeterminism is allowed into a program, because determinism guarantees are meaningful only when referring to data. A memory location M will have deterministic contents at the end of a program if and only if M is only ever updated with deterministic values.[9] A code-centric approach to determinism like [140] allows deterministic and nondeterministic values to be assigned to M, albeit in a data-race-free manner. Such an approach cannot, however, make any guarantees about M's value being deterministic. More simply, as we observed earlier (Section 5.2), levels of determinism are not composable: passing a variable with a nondeterministic value to a function written in a deterministic language will not "recover" determinism for that variable.

5.6.2 Qualifier Polymorphism

Others have shown how to extend a type qualifier system like MELD's to account for qualifier polymorphism [165,167]. Polymorphic qualifiers would admit extra flexibility for, e.g., the `this` reference or function parameters, avoiding the need to clone methods to perform the same computation on different labels.

5.7 Limit Study

To evaluate MELD within the limits of our current infrastructure (Section 5.5) we have undertaken a limit study of MELD's potential.

[9] Modulo the special case of resetting the variable to a known deterministic value. This is useful in a security context for regaining trust from untrusted values [166] but is not useful in our context since the nondeterministic value cannot be read.

5.7.1 Experimental Setup

We ran our experiments on an 8-core 2.4GHz Intel Xeon E5462 ("Nehalem") with 10GB of RAM, using 64-bit Ubuntu Linux 8.10. Our experiments run with 8 program threads on the Sun Java HotSpot VM 1.7.0_05, running in 64-bit server mode. We use the kernel benchmarks from the Java Grande Forum [168]. We have scaled the input sizes of the kernels to have meaningful runtimes, except for the Series kernel which already runs for several minutes on our machine. For Crypt, we increased the input array size by a factor of 20 and perform the main encrypt-decrypt operation 30 times in a loop. For LUFact, we increased the input size by a factor of 5; for SOR by a factor of 10; and for SparseMatmult by a factor of 20.

5.7.2 Results

We have evaluated the performance of MELDed programs with various portions of MELD's runtime instrumentation disabled. These experiments show where the performance bottlenecks are in the current system, and give an upper bound on the speedup possible through the use of `langdet` and `xldet` qualifiers.

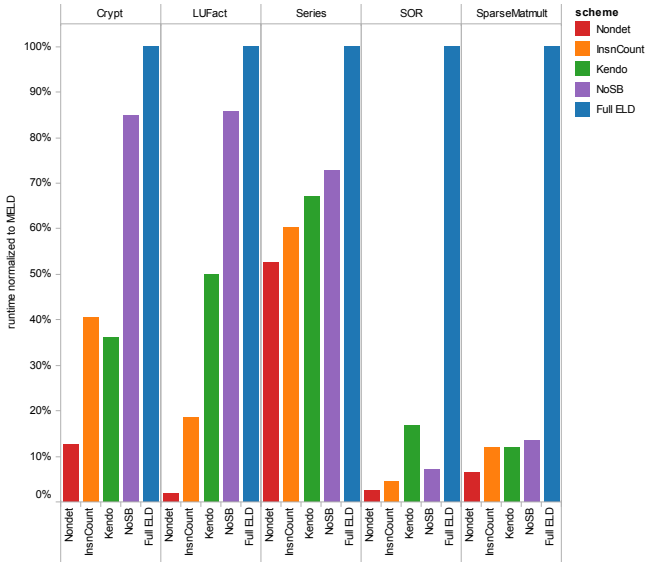

Figure 37: Runtime of Java Grande kernels with various components of MELD's runtime instrumentation disabled.

The bars in Figure 37 are normalized to the blue bars – runtime of execution under full execution-level determinism, i.e., with all memory locations labeled as `exdet`. The blue bars are the slowest configuration, and lower bars are better. The purple bars show the performance when store buffering is completely disabled, modeling the case when all memory locations are labeled as `xldet` or `langdet`. The green bars show the performance when executing in Kendo [23] mode – dynamic instruction count is used to force synchronization operations into a deterministic order, but no quantum formation or store buffering takes place. The orange bars show the

performance when only dynamic instruction counting is performed. Finally, the red bars show the performance of nondeterministic execution.

The differences between different bars in Figure 37 reveal where the time goes in the MELD runtime system. First, the sizeable gap between red and blue, for all benchmarks except Series, indicates that full execution-level determinism imposes a substantial slowdown over nondeterministic execution: up to 58x for LUFact. The gap between purple and blue shows that the extensive use of `xldet/langdet` accesses has the potential to dramatically reduce runtime overheads for SOR and SparseMatmult. Crypt and LUFact gain more from eliminating quantum formation (green bars) and deterministic synchronization (yellow bars) than from eliminating the store buffer. Smarter quantum formation strategies, such as adjusting quantum size dynamically [144,145], will likely reduce quantum formation overhead. The gap between the red and orange bars shows that our current implementation of instruction counting imposes high overheads in some cases (10x for LUFact). The MELD compiler currently naïvely instrument all control-flow back edges, which is far from optimal for tight loops and short branches. A smarter policy (e.g., from [34]) would reduce overheads noticeably.

Overall, these experiments show that MELD's primary optimization, using a deterministic language to remove store buffer instrumentation, is a promising approach. They also point to areas in the MELD runtime system that need improvement.

5.8 Conclusions

We have presented MELD, a new system for integrating deterministic languages within a deterministic execution system. We leverage deterministic execution's support for arbitrary parallel programs, and deterministic languages' support for fast, statically-checked determinism to accelerate the performance of deterministic execution on programs that cannot be readily expressed in a deterministic language. We show that naïvely integrating a deterministic language into a deterministic execution system does not preserve determinism, and we describe the requirements for soundly doing so. We use a simple qualifier-based type system to enforce isolation

between the execution-level and language-level determinism parts of a program. Our results suggest that integrating Deterministic Parallel Java into a Java port of the CoreDet deterministic execution system can improve determinism's performance substantially.

We believe that the static and dynamic approaches to determinism, heretofore separate, are much better together. An integrated approach is an important step towards making deterministic systems fast enough and general enough for widespread use.

Chapter 6 Conclusions

This dissertation has demonstrated and evaluated a series of hardware and software techniques for providing execution-level determinism in an efficient manner. We established that deterministic execution of arbitrary parallel programs is possible (Chapter 3), evaluated the trade-offs among pure-hardware (Chapter 3), hybrid hardware-software (Chapter 4), and pure-software (Chapters 4, 5) deterministic platforms, and showed that relaxing memory consistency (Chapter 4) and incorporating deterministic languages (Chapter 5) are two important optimizations for increasing the performance of execution-level determinism. We have also placed these techniques in the context of parallel programmability research (Chapter 2), showing how our work on determinism has been inspired by previous research and has inspired others.

6.1 Summary of Techniques

This dissertation describes several techniques that allow arbitrary parallel programs to execute deterministically and investigates the trade-offs among these techniques. The basic pattern that all of these techniques follow is to divide a program's execution into deterministically-sized chunks of instructions called quanta. During each quantum, threads initially execute in isolation and subsequently are allowed to communicate in a precisely controlled way.

Chapter 3 describes the use of a sharing table that tracks memory ownership (Section 3.1.3.1) and hardware transactional memory (Section 3.1.3.2) to enforce isolation among threads. For the sharing table approach, an initially-immutable sharing table prevents threads from communicating with one another. Subsequently, threads that need to modify the sharing table to make progress are executed in a deterministic serial order. The TM approach uses hardware transactions [30] to keep threads isolated and a deterministic transaction commit order to ensure that communication occurs in a precise order.

Chapter 4 describes the use of store buffers to isolate threads without the need for conflict detection. Store buffers offer a simple, speculation-free isolation mechanism without the overhead

of tracking and updating memory ownership information. The price is a more complicated memory consistency model – the sharing table and TM-based determinism systems both provide sequential consistency [65] while the store buffer approach provides weaker ordering guarantees.

Finally, Chapter 5 discusses how to integrate deterministic languages with our previously developed techniques. Using the type-and-effect system of Deterministic Parallel Java [20] to statically prove the isolation of concurrent operations, we can eliminate some of the overhead of enforcing determinism at runtime. As with providing other program properties like memory safety and type safety, a combination of static and dynamic approaches suits determinism well.

6.2 Limitations

While we have detailed how to provide determinism for arbitrary programs, there are nevertheless several limitations to our approaches. We discuss elsewhere the performance implications of our techniques which are certainly an important factor limiting wider adoption. In this section we describe other factors that inhibit real-world usage of determinism.

The first is that our deterministic mechanisms are built on a notion of deterministic logical time that is quite brittle: dynamic instruction count. Virtually any code change will alter a program's dynamic instruction count, even for an identical input, and this change can occur in opaque ways due to sophisticated compiler optimizations. A change in dynamic instruction count can alter quantum boundaries, affecting in turn a program's communication patterns and therefore its execution. In principle, a code change results in an entirely new program for which testing and validation must begin from scratch. Limited forms of robustness to such code changes are possible, e.g., ignoring extra instructions inserted for debugging. However, a more principled way of handling code changes would dramatically simplify the testing of deterministic programs.

Another limitation of our approaches is that a deterministic execution is a function of both program input and a small set of explicit configuration parameters. Quantum size, for example, has a large impact on performance and must be set explicitly, though others have proposed [144,145] deterministically adjusting quantum size at runtime. For our hardware-based techniques, the

geometry of private caches affects execution by altering quantum boundaries, though Calvin [144] shows how to virtualize these hardware resources at some performance cost. The deterministic ordering imposed on threads, needed to resolve memory update conflicts and coordinate synchronization operations, can also impose slowdown, though it too can perhaps be modified online in a deterministic way to improve performance. Allowing these configuration parameters to be derived automatically from program input will improve determinism's performance portability and eliminate the need to test different configurations for each input.

Finally, we believe that determinism is an enabler of program correctness, though it does not directly provide it. A program run on one of our deterministic systems may well crash more frequently than when run on a nondeterministic platform. Over time, determinism's debugging benefits should result in more correct programs even though end users may prefer not to wait. Nondeterminism also provides a weak kind of fault-tolerance, in that there is a chance that a particular crash may vanish simply on re-running the program. Our determinism techniques can perhaps be extended to allow a more principled notion of execution diversity where execution is a function of program input as well as, e.g., a random seed. Research in exposing [56] and masking [54] software bugs is also likely to prove deeply synergistic with deterministic execution.

6.3 Looking Forward

It would be fitting if future work on deterministic execution were entirely a function of previous work. This is unlikely to be the case, however, as researchers continue to improve determinism's performance, overcome its limitations, uncover new uses for it, and incorporate it with other established mechanisms for improving parallel programmability. It is our particular hope that determinism's debugging and testing benefits can be more rigorously established going forwards. This can come through better deterministic systems that explicitly target these use-cases instead of focusing on improving performance. User studies and the study of real-world deployments of determinism will also be invaluable for measuring determinism's benefits and informing the design of next-generation deterministic systems that are even more helpful. Ultimately, these efforts will

broaden determinism's impact and help make parallelism, with its attendant performance and energy benefits, more accessible to a wide audience.

REFERENCES

1. Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *Queue*, 10:4 April 2012, 10–27.
2. Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing Parallel Software Efficiently with Lithe. In *PLDI*, 2010.
3. Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *HotPar*, 2009.
4. Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys*, 2010.
5. Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24:8 August 1991, 28–33.
6. Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, 1989.
7. Cyrille Artho, Klaus Havelund, and Armin Biere. High-Level Data Races. *Journal on Software Testing, Verification & Reliability*, 13:4 2003, 220–227.
8. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.
9. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL*, 2005.
10. Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.
11. Marvin Zelkowitz. Reversible execution as a diagnostic tool. PhD Dissertation, Cornell University, 1971.
12. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
13. Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.
14. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI*, 2007.
15. VMware: VMware Workstation Zealot: Enhanced Execution Record / Replay in Workstation 6.5. 2008. http://blogs.vmware.com/workstation/2008/04/enhanced-execut.html.
16. GDB and Reverse Debugging. http://www.gnu.org/software/gdb/news/reversible.html.
17. Corensic Concurrency Debugger and Thread Debugger for Parallel Applications and Multi-Core Software. http://www.corensic.com/.
18. William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, 2002.
19. Guy Blelloch. NESL: A Nested Data-Parallel Language. 1992.
20. Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.
21. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.
22. Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, 2009.
23. Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.
24. Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

25. Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. *IEEE Micro*, 30:1 January 2010, 40–49.
26. Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS*, 2011.
27. Joseph Devietti, Dan Grossman, and Luis Ceze. The Case For Merging Execution- and Language-Level Determinism with MELD. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2012.
28. Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA*, 1995.
29. Wolfgang Straßer. Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten. PhD Dissertation, Technische Universität Berlin, 1974.
30. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.
31. The IBM zEnterprise EC12 - proven hybrid computing designed to manage multiple workloads, with the simplicity of a single system. 2012. http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=AN&subtype=CA&htmlfid=897/ENUS112-155&appname=USN.
32. IBM Corporation. Transactional Memory. http://pic.dhe.ibm.com/infocenter/compbg/v121v141/index.jsp?topic=%2Fcom.ibm.xlcpp121.bg.doc%2Fproguide%2Fbg_tm_concept.html.
33. James Reinders. Transactional Synchronization in Haswell. 2012. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell.
34. Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.
35. Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29:12 December 1996, 66–76.
36. Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6:3 May 2011, 1–212.
37. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:7 July 1978, 558–565.
38. Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA*, 2010.
39. Laura Effinger-Dean, Hans-J. Boehm, Dhruva Chakrabarti, and Pramod Joisha. Extended Sequential Reasoning for Data-Race-Free Programs. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2011.
40. Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*, 2012.
41. Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.
42. D. Schonberg. On-the-fly detection of access anomalies. In *PLDI*, 1989.
43. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15:4 November 1997, 391–411.
44. Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, 2007.
45. Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28:2 March 2006, 207–255.
46. Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, 2006.

47. Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFX: a simple and efficient memory model for concurrent programming languages. In *PLDI*, 2010.
48. Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, 2009.
49. Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *PLDI*, 2010.
50. Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.
51. Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.
52. Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.
53. Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.
54. Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, 2008.
55. Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA*, 2007.
56. Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.
57. Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.
58. Brandon Lucia, Luis Ceze, and Karin Strauss. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA*, 2010.
59. Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.
60. Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.
61. Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do I Use the Wrong Definition? DeFuse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *OOPSLA*, 2010.
62. Brandon Lucia, Benjamin P. Wood, and Luis Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI*, 2011.
63. Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.
64. Jaroslav Ševčík and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, 2008.
65. Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28:9 September 1979, 690–691.
66. Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *ICPP*, 1991.
67. Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, 1999.
68. José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, 2002.
69. Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *SPAA*, 1997.

70. Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, 2001.
71. Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA*, 2007.
72. Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, 2006.
73. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:7 July 1970, 422–426.
74. Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. InvisiFence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA*, 2009.
75. Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. Efficient Processor Support for DRFx, a Memory Model With Exceptions. In *ASPLOS*, 2011.
76. Christoph von Praun, Harold W. Cain, Jong-Deok Choi, and Kyung Dong Ryu. Conditional Memory Ordering. In *ISCA*, 2006.
77. TBB Home. http://threadingbuildingblocks.org/.
78. Grand Central Dispatch - Mac OS X Technology Overview - Apple Developer. http://developer.apple.com/technologies/mac/snowleopard/gcd.html.
79. Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *OOPSLA*, 2009.
80. Doug Lea. A Java fork/join framework. In *JAVA*, 2000.
81. Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the Cell BE architecture. In *SC*, 2006.
82. Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA*, 1997.
83. John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6:2 March 2008, 40–53.
84. Khronos Group. OpenCL. http://www.khronos.org/opencl/.
85. Microsoft Corporation. Compute Shader Overview (Windows). http://msdn.microsoft.com/en-us/library/ff476331.aspx.
86. Arun Ramamurthy. Towards Scalar Synchronization in SIMT Architectures. Master's Thesis, University of British Columbia, 2011.
87. Michael Boyer, Kevin Skadron, and Westley Weimer. Automated Dynamic Analysis of CUDA Programs. In *Workshop on Software Tools for MultiCore Systems*, 2008.
88. Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *PPoPP*, 2011.
89. Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware transactional memory for GPU architectures. In *MICRO*, 2011.
90. Jim Gray. Notes on data base operating systems. In Bayer, R., Graham, R. and Seegmüller, G., eds., *Operating Systems*. Springer Berlin / Heidelberg, 1978, 393–481.
91. Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.
92. Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *ISCA*, 2005.
93. Milo Martin, Colin Blundell, and E. Lewis. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5:2 July 2006, 17–17.
94. Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *ISCA*, 2007.
95. Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logTM. In *ASPLOS*, 2006.

96. Colin Blundell, Arun Raghavan, and Milo M.K. Martin. RETCON: transactional repair without replay. In *ISCA*, 2010.
97. Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *MICRO*, 2006.
98. Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware atomicity for reliable software speculation. In *ISCA*, 2007.
99. Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. In *ASPLOS*, 2012.
100. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, 1995.
101. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.
102. Microsoft Corporation. STM.NET. http://msdn.microsoft.com/en-us/devlabs/ee334183.aspx.
103. Intel Corporation. Intel® C++ STM Compiler, Prototype Edition. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.
104. TransactionalMemory - GCC Wiki. http://gcc.gnu.org/wiki/TransactionalMemory.
105. Joe Duffy. A (brief) retrospective on transactional memory. *Generalities & Details: Adventures in the High-tech Underbelly*, 2010. http://www.bluebytesoftware.com/blog/2010/01/03/ABriefRetrospectiveOnTransactionalMemory.aspx.
106. Cliff Click. And now some Hardware Transactional Memory comments…. *Cliff Click's Blog*, 2009. http://www.azulsystems.com/blog/cliff/2009-02-25-and-now-some-hardware-transactional-memory-comments.
107. Tom Knight. An architecture for mostly functional languages. In *ACM Conference on LISP and Functional Programming*, 1986.
108. Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, 1998.
109. Venkata Krishnan and Josep Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *ICS*, 1998.
110. Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *ICS*, 1998.
111. J. Gregory Steffan and Todd Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. 1998.
112. Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *HPCA*, 1998.
113. Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS*, 2010.
114. Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable Speculative Parallelization on Commodity Clusters. In *MICRO*, 2010.
115. Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. In *PADD*, 1988.
116. Thomas J. Leblanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36:4 April 1987, 471–482.
117. David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *PADD*, 1991.
118. Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD*, 1993.
119. Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.
120. Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, 2005.

121. Min Xu, Mark D. Hill, and Rastislav Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.
122. Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS*, 2006.
123. Derek R. Hower and Mark D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, 2008.
124. Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, 2008.
125. Gwendolyn Voskuilen, Faraz Ahmad, and T. N. Vijaykumar. Timetraveler: exploiting acyclic races for optimizing memory race recording. In *ISCA*, 2010.
126. Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. Karma: scalable deterministic record-replay. In *ICS*, 2011.
127. Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. LReplay: A Pending Period Based Deterministic Replay Scheme. In *ISCA*, 2010.
128. Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors. In *MICRO*, 2011.
129. Jeff Huang, Peng Liu, and Charles Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *FSE*, 2010.
130. Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, 2011.
131. Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17:2 May 1999, 133–152.
132. Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.
133. Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.
134. Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In *ESOP*, 2009.
135. Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, 2009.
136. Jacob Burnim and Koushik Sen. DETERMIN: inferring likely deterministic specifications of multithreaded programs. In *ICSE*, 2010.
137. Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20:3 May 1998, 483–545.
138. Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP*, 2009.
139. Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal - Nested Data Parallelism in Haskell. In *Euro-Par*, 2001.
140. Robert Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.
141. Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. Inferring Method Effect Summaries for Nested Heap Regions. In *ASE*, 2009.
142. Robert L. Bocchino and Vikram S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *ECOOP*, 2011.
143. Stephen Heumann and Vikram Adve. Tasks with Effects: A Model for Disciplined Concurrent Programming. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2012.

144. Derek R. Hower, Polina Dudnik, David A. Wood, and Mark D. Hill. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, 2011.
145. Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.
146. Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.
147. Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.
148. Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.
149. Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.
150. Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, 2010.
151. Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: a model for parallel and incremental computation. In *OOPSLA*, 2011.
152. Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Guri Sohi. Speculative Versioning Cache. In *HPCA*, 1998.
153. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.
154. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, 2008.
155. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
156. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.
157. Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.
158. Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, 1990.
159. Enrique Vallejo, Marco Galluzzi, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenström, James E. Smith, and Mateo Valero. Implementing Kilo-Instruction Multiprocessors. In *ICPS*, 2005.
160. Michel Dubois, Jin Chin Wang, Luiz A. Barroso, Kangwoo Lee, and Yung-Syau Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *SC*, 1991.
161. John Ousterhout. Scheduling Techniques for Concurrent Systems. In *ICDCS*, 1982.
162. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI*, 2002.
163. Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *ISSTA*, 2008.
164. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.
165. Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL*, 1999.
166. Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, 2009.
167. Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, 1999.

168. Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum Benchmark suite. In *ACM-ISCOPE Conference on Java Grande*, 2001.

VITA

Joseph Devietti was born in Glendale, California. In 2006 he earned a Bachelor of Science in Engineering degree in Computer Science and a Bachelor of Arts in English from the University of Pennsylvania. In 2009 he earned a Master of Science degree in Computer Science and Engineering from the University of Washington and he earned a Doctor of Philosophy in Computer Science and Engineering at the University of Washington in 2012.