SIGNATURES IN TRANSACTIONAL MEMORY SYSTEMS

by

Luke Yen

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

2009

Abstract

Transactional memory (TM) is an emerging parallel programming paradigm. It seeks to ease common problems with lock-based parallel programming (e.g., deadlock, composability) through the use of novel language-level constructs. Researchers have proposed many TM systems to implement TM programming semantics, including all-hardware, all-software, or a hybrid of hardware and software support. We focus on hardware TM systems (HTMs) due to their performance advantages over the other implementations.

Early HTM proposals restrict the type of transactions that can be run — either its size or its time — and this may unnecessarily burden TM programmers. Such burdens may slow or even squelch the adoption of TM as a wide-spread parallel programming model. Subsequent HTM proposals support virtualizing transactions (in both space and time), but may incur high overheads when handling common virtualization events (e.g., cache replacements of transactional data), or complicate existing hardware (HW) designs (e.g., L1 cache). An ideal HTM design should have the following characteristics: (1) Allow the execution of transactions of arbitrary size and time and (2) Minimize changes to existing processor cores and systems. This dissertation makes several contributions in the design and analysis of a HTM with these properties.

First, this dissertation contributes a new HTM system called LogTM Signature Edition (LogTM-SE). LogTM-SE synergistically combines Log-Based Transactional Memory (LogTM)'s log for version management (i.e., mechanisms for tracking old and new data values) with an imprecise hardware mechanism, called signatures, for conflict detection (i.e., mechanisms for detecting conflicts amongst concurrent transactions). Signatures keep track of an unbounded

number of transactional load and store addresses using fixed hardware, but may indicate a conflict when none exists — a false conflict. We show LogTM-SE allows transactions to be virtualized using a HTM system coupled with operating system support that minimizes changes to existing HW structures, at the cost of performance degradations due to signature false conflicts.

Second, we contribute to the design and analysis of LogTM-SE's signatures through Notary, a coupling of two ideas to enhance signatures. We explore the implementation overheads of the H_3 signature hash function designed to minimize false conflicts. We find that H_3's area and power overheads may be too great for HTM designs. This dissertation proposes Page-Block-XOR (PBX), a lower-cost hash function that performs as well as H_3. Additionally, we find existing signature proposals may unnecessarily incur false conflicts caused by the insertion of thread-private memory references. This dissertation proposes privatization techniques based on the removal of stack references from signatures and a heap-based privatization interface that enables the TM programmer to indicate which addresses should not participate in conflict detection.

Third, we contribute a new performance debugging framework for HTM systems. The additional critical-section parallelism enabled by TM makes it harder for the HTM designer to understand the performance impact of different HTM designs. In addition, TM programmers may not fully understand why TM programs execute with varying performance on different HTM systems. This dissertation contributes TMProf, a lightweight profiling framework that accounts for the frequency of common HTM events and its overheads using a set of hardware performance counters implemented in each processor core. We propose two versions of TMProf, base (BaseTMProf) and extended (ExtTMProf), and show it can be used to explain a multitude of key parameter

changes in two example HTM designs. TMProf enables HTM designers and TM programmers to better understand the performance impact of HTM designs.

Fourth, we contribute six new extensions to signature designs. These six extensions focus on reducing signature false conflicts by leveraging static properties of the program (e.g., not all static transactions conflict with one another), using high-level program information (e.g., the set of objects accessed within transactions), optimizing for a program's spatial locality, and adapting to the dynamic nature of transactional conflicts. This dissertation shows that although most of these six extensions do not offer substantial performance improvements, future research can focus on the most promising extension (optimizing signatures for spatial locality).

Acknowledgments

This dissertation is dedicated to my lovely wife Ting. She was courageous enough to move from a warm state to join me in the frigid environment of Wisconsin. She has consistently provided invaluable moral and financial support. I am deeply honored by her patience, understanding, and commitment throughout this entire process.

I would like to thank my family and non-Wisconsin friends. They have provided many experiences and perspectives to last a lifetime. I would also like to acknowledge my two pets, Latte the crazy Lab and Nimbus the pear-loving Chinchilla, both of whom have provided numerous hours of entertainment and hilarity.

My education at the University of Wisconsin, Madison has been a life-changing experience. I would first like to thank my advisor Mark Hill for his many years of patience, wisdom, and encouragement. I sincerely appreciate all his advice and anecdotes, and he has certainly made my graduate experience an enjoyable one. Second, I thank David Wood. As the other director of the Multifacet project he has provided a wealth of information and advice. Third, I thank Mike Swift and Ben Liblit. I appreciate their time and advice on all aspects of my research, and for providing insightful feedback during my paper submissions. Fourth, I thank Stark Draper. He was generous enough to meet with me many times to discuss entropy and how it applies to my research. I am grateful for his insight and advice.

I would also like to thank Guri Sohi. He gave me valuable advice during my first year of graduate school, and let me attend some of his Multiscalar group's meetings. I also thank the other professors in my committee, Mikko Lipasti and Karu Sankaralingam. They have provided valuable feedback on my research and presentations. My breadth of knowledge has certainly been expanded by being students in their classes and through my interactions with them outside of class.

I also thank the numerous Computer Science professors I had the privilege to take classes with, including: Remzi Arpaci-Dusseau, Susan Horwitz, Miron Livny, Jeff Naughton, and Raghu Ramakrishnan. Their classes were enlightening and their individual teaching styles made each class enjoyable.

Graduate school is a very unique environment, where a multitude of people with lots of talent come together for a common purpose. I am grateful to have met a wonderful group of people, including Nidhi Aggarwal, my study partner for the quals and a source of interesting stories, Alaa Alameldeen, who was a good mentor when I first joined Multifacet, Matt Allen, who always had insightful feedback for my research and in reading group, Brad Beckmann, a colleague that offered great advice and a wealth of knowledge, Jayaram Bobba, a fellow LogTMer who always had useful feedback and viewpoints on all things TM, Koushik Chakraborty, a helpful colleague and a killer racquetball player, Igor Grobman, who always had hilarious stories to tell, Natalie Enright Jerger, a colleague with interesting viewpoints and political comments, Dan Gibson, whose intelligence and wit are truly unique, and he and his wife Megan hosted numerous poker nights, Mike Marty, who helped me on cache coherence and was a good mentor when I first joined Multifacet, Andy Phelps, who provided first-hand knowledge about his previous industrial experiences, Dana Vantrease, a colleague from my first year in Wisconsin and always available to discuss research ideas, Yasuko Watanabe, a good colleague that provided excellent feedback on my research, Philip Wells, who was always available to talk about research or help out, and Min Xu, who was a great mentor and person to discuss research ideas with. I am also grateful to have met some new graduate students, including Marc De Kruijf, Polina Dudnik, Derek Hower, and James Wang. They collectively bring in new enthusiasm and vigor into our department. I also thank my previous office-mates Bhavesh Mehta and Michelle Moravan. Bhavesh was laid back and fun to work with. Michelle was a former member of our LogTM group, and she brought me up to speed and answered many of my questions when I first joined LogTM. Haris Volos is my current office-mate, and I enjoy exchanging ideas and stories with him.

I also thank the teachers that influenced me to pursue the field of Computer Science and eventually computer architecture: Mr. Coe and Mrs. Chapman, who taught Computer Science at Plano Senior High and got me interested in programming, and Doug Burger, my undergrad advisor who taught very entertaining and informative classes at UT and encouraged me to go to grad school at Wisconsin.

Lastly, I thank the Wisconsin Computer Architecture Affiliates for their time and feedback, the Computer Systems Laboratory for machine and software support, the Wisconsin Condor project, and the National Science Foundation for providing me with a Graduate Research Fellowship.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The reality of multi-core chip-multiprocessors (CMPs) is here. Many major vendors (Intel, AMD, IBM, and Sun) have released products which range from two to eight cores on a single chip [53,6,61,105]. Berkeley researchers have even made bold predictions proclaiming future CMPs could contain thousands of cores [8]. However, most software has not matured enough to take advantage of the increased hardware resources. A primary reason for this is because parallel programming is a difficult task, and has been for the last 30 years. This is an important problem for computer architects to tackle, because architects can no longer expect the additional transistors enabled by Moore's Law to continue to be used with the same efficiency as in the past to increase single-thread performance. Recently, researchers have proposed *Transactional Memory* (TM) [48,64] as a new parallel programming paradigm that seeks to ease the task of writing parallel programs.

Transactional memory introduces the idea of language-level atomic blocks to replace all uses of locks in a multi-threaded parallel program. A major advantage of atomic blocks is that, unlike locks, programs written using them will not deadlock or have composability problems. Each atomic block forms a *transaction*. These atomic blocks have the Atomicity, Consistency, and Isolation (ACI) properties of database transactions: Atomicity to ensure either all instructions in the atomic block commit successfully or aborts, Consistency to ensure subsequent transactions view consistent memory state, and Isolation to ensure that each transaction's operations appear isolated

from other transactions, no matter how many transactions are concurrently executing. A successful transaction *commits*, while an unsuccessful one that *conflicts* with a concurrent transaction *aborts*. The job of the underlying TM system is to implement this abstraction. There have been numerous TM systems proposed, including all-hardware (e.g., Herlihy and Moss' HTM [48], TCC [43], LogTM [80], Bulk [23], OneTM [13]), all-software (e.g., TL2 [34], Intel's STM [52]), or a hybrid of the two (e.g., HyTM [33]). Larus and Rajwar [64] present a thorough overview of a wide variety of TM systems that have been proposed. This dissertation focuses on hardware TMs (HTMs) due to their performance advantages over the other types of TM systems.

1.1 Version Management and Conflict Detection

In order to enforce the ACI properties any transactional memory system must support *version management* and *conflict detection*. Version management refers to the mechanisms which simultaneously stores newly written values for use on transaction commit and old values for use in restores on transactional aborts. Conflict detection refers to the mechanisms to track the *read-* and *write-sets* of transactions and only allowing non-conflicting transactions to commit. The read- and write-sets of a transaction denote all of the memory locations read or written while in a transaction. A conflict between transactions occurs when two or more transactions access the same memory location and at least one of the transactions is performing a write to that location. A conflict is resolved by stalling, or alternatively, aborting one or more transactions and restoring the old values captured by the version management system. Conflict detection can be implemented in hardware (for hardware TM (HTM) systems) or in software (for software TM (STM) systems). This dissertation focuses on conflict detection in HTMs.

1.2 Thesis Contributions

We now present an overview of this dissertation's contributions related to conflict detection using signatures in HTMs, as well as performance debugging HTMs. These contributions are important because they enable HTMs which efficiently handle transactions that run in arbitrary length of time and are of arbitrary size. Furthermore, our contribution to performance debugging enable both HTM designers and TM programmers to better understand the performance of HTM systems.

1.2.1 LogTM Signature Edition (LogTM-SE)

Chapter 4 contributes a novel HTM called LogTM Signature Edition (LogTM-SE). LogTM-SE's design reaps the benefits of prior HTM proposals. It combines the version management mechanisms from LogTM [80] (its log) and the conflict detection mechanisms from Bulk [23] (its signatures). Signatures are finite hardware structures that imprecisely track the read- and write-sets of transactions. They are usually implemented as Bloom filters [12], which may incur false positives (indicating the presence of an address when it doesn't exist in the filter). False positives induce false conflicts, which can degrade the performance of a TM system.

LogTM-SE implements low-complexity hardware in each processor core that is then used by the operating system to support various events that might otherwise abort transactions or incur high performance overheads in other HTM designs. These events include thread switches, page remapping, and unbounded nesting. Additionally, since LogTM-SE inherits LogTM's version management mechanisms, no additional mechanisms are needed to handle replacements of transactional data.

We evaluate LogTM-SE using a variety of lock-based programs converted to use transactions. We show TM programs executing under LogTM-SE execute similar or faster than its lock-based equivalents. Realistic (finite) signatures are comparable to the performance of perfect (infinite) signatures, even when small signature sizes are used. Finally, we show victimization of transactional data from caches can occur and LogTM-SE handles this event with minimal complexity and resources.

1.2.2 Notary

Chapter 5 contributes Notary, a coupling of two hardware techniques to enhance signatures. Notary tackles two problems with signature designs. First, the best proposed signature hashing function (H_3) can incur high area and power overheads. Second, signature false conflicts can be caused by signature bits that are set by thread-private memory references.

Notary tackles these problems separately. First, Notary details how the idea of address-bit entropy can be used to zero in on the most random input address bits. We use entropy as the basis behind Page-Block-XOR (PBX), a new hash function that incurs lower area and power overheads than H_3. We evaluate Notary with a variety of TM programs. We find that PBX achieves similar performance to H_3.

Second, Notary introduces privatization techniques which remove thread-private memory accesses from signatures. This is implemented by removing stack references from signatures and a new heap-based privatization interface which enables programmers to declare memory objects which do not participate in conflict detection. Notary's heap-based privatization interface improves performance for several workloads.

Finally, we find Notary is applicable to other signature-based TM and non-TM systems. Since signature hashes are not programmer-visible, PBX hashing can be directly implemented in any system which uses signatures. Second, if a program correctly synchronizes accesses to private data, a signature-based non-TM system can take advantage of our privatization techniques to avoid tracking or checking these memory accesses. In general, the idea of filtering addresses based on some property may be applicable to other signature-based systems.

1.2.3 TMProf

Chapter 6 contributes TMProf, a lightweight profiling infrastructure that is used to performance debug HTM systems. The additional critical-section parallelism created by TM increases the amount of thread interleavings in multi-threaded programs. This increases the difficulty of the evaluation of HTM designs, in both prototype and production systems. Furthermore, TM programmers may not have information which helps them understand why their programs perform better on one HTM versus another.

TMProf is implemented as a set of performance counters on each processor core. The base implementation, BaseTMProf, tracks the frequency and cycle overheads of common HTM events (commit, wasted transactions, useful transactions, stalls, aborts, non-transaction, and implementation-specific). An alternative implementation, ExtTMProf, builds upon BaseTMProf and adds additional transaction-level profiling (frequency and sizes of aborted transactions and amount of transaction work until commit after a write-set prediction).

We evaluate TMProf on two HTMs, LogTM-SE and an approximation of Stanford's TCC [43]. First, we find that TMProf is able to explain HTM performance when its parameters change. Second, we find empirical evidence indicating counter-based profiling infrastructures such as

TMProf is not sufficient to understanding subtle timing and control-path dependent effects in HTM systems. Therefore we advocate future research into hardware support for tracking the critical-path of TM programs.

1.2.4 Extensions to Signatures

Chapter 7 contributes six extensions to signatures. These extensions aim to reduce signature false conflicts, and therefore improve the execution times of TM programs running on top of signature-based TM systems.

The first extension focuses on static transaction independence. That is, the idea that some sets of static transactions may be statically guaranteed never to dynamically conflict with another set of transactions. If the compiler or programmer can convey this information to signatures, signatures can choose dynamically when to turn on and off conflict detection for certain transactions.

The second extension focuses on program-level information, and how it can be used to help signatures reduce false conflicts. Specifically, signatures use information about which memory objects are touched within transactions, and use this information on signature lookups to decide whether there is potentially a transaction conflict.

The next three extensions focus on the spatial locality property of programs. We note that if a program exhibits spatial locality, memory addresses are likely to be located near each other. Thus its low-order memory address bits may have some commonality. We examine signature implementations which ignore this commonality. We also examine implementations which customize hash functions based on the different properties of high-level and low-level address bits.

The final extension deals with the dynamic nature of transactional conflicts. We note that signature conflicts may be due to bad luck in hashing. If the signature hash function changes, these false conflicts may be transient conflicts. We examine a simple dynamic hashing scheme based on bit-rotation, which rotates the bits of the input address before providing them as inputs to the hash functions.

We evaluate these six extensions using a variety of TM programs, and find the majority of them do not significantly improve program execution time. The most beneficial appears to be signature optimizations that take advantage of spatial locality, and we advocate future research target this area.

1.3 Dissertation Structure

Chapter 2 presents a background on conflict detection in TM systems, and presents an overview of prior solutions. Chapter 3 discusses the tools, methodology, and workloads we use in this dissertation's evaluations. Chapter 4 develops the LogTM-SE HTM design. Chapter 5 develops Notary, two techniques to enhance signature designs. Chapter 6 develops TMProf, a hardware profiling framework for performance debugging HTMs. Chapter 7 develops six extensions to signatures. Finally, Chapter 8 concludes and offers reflections on the research.

1.4 Relationship to My Prior Work

This dissertation includes work that previously appeared in two conference publications, and a chapter that has been submitted for publication.

A preliminary version of Chapter 4 (LogTM-SE) was previously published in the proceedings of the 13th High Performance Computer Architecture (HPCA) in 2007 [127]. Co-authors on this

work include Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Chapter 4 extends that work by adding an example of LogTM-SE's implementation using Sun's Gigaplane broadcast interconnect, detailing several design alternatives for out-of-order processor cores, and including comparisons to TokenTM [16].

A preliminary version of Chapter 5 (Notary) was previously published in the proceedings of the 41st International Symposium on Microarchitecture (MICRO) in 2008 [128]. This work was co-authored with Stark C. Draper and Mark D. Hill. Chapter 5 adds in-depth sensitivity analysis results, including overlaps between PBX's bit-fields, the effects of removing stack references from signatures, and execution time results for signatures that use four hash functions.

Chapter 6 (TMProf) has been submitted for publication.

Chapter 2

Background on Conflict Detection in Transactional Memory Systems

This chapter reviews the idea of conflict detection and describes prior proposals which implement conflict detection mechanisms in transactional memory (TM) systems.

2.1 Overview of Conflict Detection

Conflict detection refers to mechanisms in a TM system which track the read- and write-sets of transactions (the set of memory addresses that are read and written inside transactions) and prevent conflicting transactions from committing. Conflicts occur whenever there are two or more transactions accessing the same memory address, and at least one of the accesses is a write.

In the most basic form the conflict detection mechanism must support the following operations in order to track the read- and write-sets of transactions: insertion of addresses, performing lookup on addresses to see if they exist, and the removal of all addresses (on commit and abort). The key observation in conflict detection is that it is correct for the conflict detection mechanism to over-approximate the read- and write-sets of transactions, but it is *incorrect* to miss any transactional reads and writes. The over-approximations may lead to *false positives,* in which addresses are indicated as belonging to the read- or write-sets when they are actually not, but we always want to disallow *false negatives,* in which addresses are not reported as belonging in read- or write-sets when they actually are. An ideal conflict detection mechanism would have the fol-

lowing properties: be fast, easy to implement in hardware, have low false positives, be software-visible (for flexibility in policy or usage), handle both small and large transactions, and induce low overheads on virtualization events (e.g. context switching and paging).

2.2 Conflict Detection in Hardware Transactional Memory (HTM) systems

This dissertation focuses on conflict detection mechanisms implemented in hardware transactional memory (HTM) systems. We now present an overview of prior proposals of conflict detection in HTMs.

2.2.1 Access Bits

Transactional conflicts can be detected *eagerly* or *lazily* [80], with eagerly meaning conflicts are detected when the conflicting request occurs and lazily meaning conflicts are detected when the transaction tries to commit. The first hardware transactional memory system was proposed by Herlihy and Moss [48], in which all transactional loads and stores operated from a transactional cache, separate from the primary L1 cache. Transactional conflicts are eagerly detected through the cache coherence protocol by checking the states of lines in the transactional cache. Rajwar and Goodman's Speculative Lock Elision (SLE) [91] and follow-on Transactional Lock Removal (TLR) [92] also eagerly detects conflicts by adding an "access" bit to indicate if a cache line was read or written transactionally, and then leveraging the cache coherence protocol to check this bit in parallel with the tag lookup.

Hammond et al.'s Transactional Coherence and Consistency (TCC) [43] introduced the idea of separate read (R) and write (M) bits coupled with cache lines in order to implement lazy con-

flict detection. At transaction commit, the write-set (denoted by the M bits) is broadcast to all other processors in the system so that active transactions can detect conflicts.

In Ananian et al.'s Unbounded Transactional Memory (UTM) [7], every memory block is tagged with read and write bits so that conflicts can be detected independent of the cache coherence protocol, either when transactions overflow on-chip caches (due to cache replacement or conflicts) or when transactions are virtualized. In the same paper they described a more practical system called Large Transactional Memory (LTM), in which each cache line is extended with a "T" bit to indicate transactional access to that line, and conflicts are detected eagerly via the cache coherence protocol. On cache overflows of transactional data, an "O" bit associated with the cache line is set and an overflow handler is invoked to check for subsequent conflicts to that line.

Rajwar et al.'s Virtual Transactional Memory (VTM) [93] employs a XADT structure that tracks transactional state for overflowed blocks for use in eager conflict detection. Each overflowed block has a XADT entry, and each entry is tagged with read and write bits to indicate whether the line has been transactionally read or written. A XF Bloom filter is used to filter out non-overflowed memory addresses from possibly overflowed transactional addresses, and optimizes whether the XADT needs to be checked for conflicts.

Moore et al.'s log-based transactional memory (LogTM) [80] extends each cache line with separate read and write bits, and also detects conflicts eagerly via the cache coherence protocol. The write bit serves a dual purpose: it is used to filter undo-log write requests so that duplicate entries are not logged, as well as to indicate whether the line has been written to transactionally. Cache overflows are handled in the directory-based system by setting sticky-M and sticky-S directory states so that conflicting requests can continue to be checked correctly. The sticky-M

state denotes that a transactionally written line has overflowed on-chip caches and any subsequent read or write to this line needs to forward its request to the marked processor in order to correctly perform conflict detection. The sticky-S state denotes that a transactionally read line has overflowed on-chip caches and subsequent write requests (and corresponding invalidate messages) to this line need to be forwarded to the transactional reader for conflict detection. In addition this coherence protocol allows requestors to be NACKed in the presence of conflicts, and for them to retry their requests at a later time.

The HTM systems described above have the advantage that read- and write-sets can be perfectly tracked for transactions that fit in on-chip caches. For those that support transactions that are unbounded in space and time (UTM/LTM, VTM, and LogTM), additional mechanisms may be employed that involve complex hardware (UTM/LTM, VTM) or may lead to false positives (LogTM) for transactions that overflow on-chip caches. However, LogTM does not support context switching or paging.

2.2.2 Access signatures

Ceze et al. [23] introduce the idea of hardware signatures in their Bulk HTM to track the read- and write-sets of transactions. The key idea in Bulk is to decouple read- and write-set tracking from the on-chip caches by moving this state from the cache lines to separate hardware signatures. This allows for no modifications to be made to existing caching structures and allows for conflict detection of both in-cache and out-of-cache transactional data using the same hardware. Unlike access bits, a signature is an imprecise mechanism for tracking read- and write-sets. Signatures over-approximate the set of addresses in the read- and write-sets, and therefore are subject to false positives (but never any false negatives). Ceze describes various signature operations (intersec-

tion, union, membership, set decoding) that can be used for implementing lazy conflict detection in Bulk. Lazy conflict detection is implemented by broadcasting a transaction's write signature on transaction commit, and other processors check for conflicts by performing signature intersection between its own read and write signatures with the broadcasted signature. In contrast. eager conflict detection does not require signature broadcast, and can use a regular per-address lookup operation to check for conflicts.

Bulk implements signatures using Bloom filters [12]. Bulk's main signature configuration uses two hash functions that operate on distinct bit-fields of the input address (and sets two bits in the signature on each insertion). In general, there are two important parameters of Bloom filters that influence the amount of false positives that it experiences. The first is the size of the Bloom filter. Overall, if everything else is fixed, bigger signatures result in fewer false positives. The second parameter is the number of hash functions used in the signature. Depending on the number of elements inserted in the Bloom filter (details in Section 2.3.1), increasing the number of hash functions may reduce the probability of false positives.

Finally, signatures can be implemented using other mechanisms besides Bloom filters. For example, Sanchez et al. [97] describe the Cuckoo-Bloom signature, which incorporates the concept of cuckoo hashing [85] in hardware tables that degenerate to Bloom filters as the tables fill up.

2.3 Signatures and False Positives

Although signatures are an attractive implementation of conflict detection in HTMs, they have the disadvantage of false positives, indicating an address is present in the signature while it is actually not. The following sections describe the problem of false positives with Bloom filter sig-

nature implementations and motivate why tackling the false positive problem is of primary importance in signatures for conflict detection.

2.3.1 Bloom Filters: The Theory

Fan et al. [36] calculate the probability of false positives in Bloom filters. Assume a Bloom filter of size m bits that uses k independent hash functions (each with range $1..m$) to hash n elements into the Bloom filter. Also assume that the n elements are uniformly distributed amongst all m positions in the Bloom filter. They observe that after inserting these n elements the probability that a particular bit (position) is still 0 is exactly:

$$P(\text{empty slot}) = \left(1 - \frac{1}{m}\right)^{kn}$$

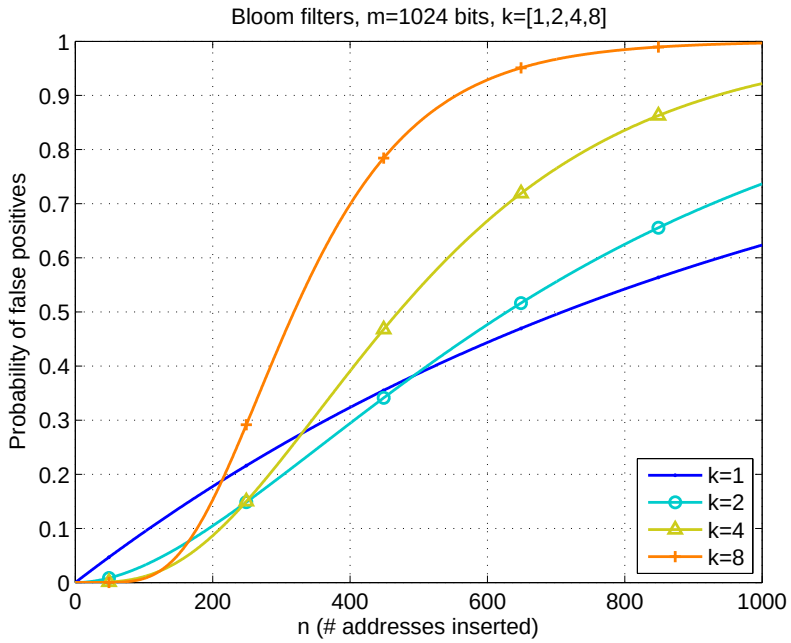


FIGURE 2-1. Bloom filter false positives

Hence the probability of a false positive is:

$$P(\text{false positive}) = \left(\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}\right) \approx \left(1 - e^{\left(k\frac{n}{m}\right)}\right)^{k}$$

They show that the right-hand side is minimized for $k = \ln 2 \times \frac{m}{n}$. Figure 2-1 illustrates the probability of false positives for a Bloom filter with m=1,024 bits, k=1,2,4,8 hash functions and insertions of n (up to 1,000) elements.

2.3.2 Bloom Filters: The Reality

In real implementations of Bloom filters, however, the theory of false positives might not exactly follow the false positive rates seen in real implementations. There are several reasons for this. First, given a fixed hardware budget it is not always practical to implement the optimal k hash functions needed to minimize the false positive rate for a given Bloom filter size. Moreover the k hash functions are assumed to be independent of each other.

Second, the theory of Bloom filters relies on hash functions that create a uniform distribution of hash values. These hash functions are relatively simple to derive if input addresses are already uniformly random. However, real programs exhibit both temporal and spatial locality, in which addresses are likely to be referenced repeatedly in a time period and addresses whose locations are neighboring the current reference are likely to be referenced in the near future. In the worst case, the interaction of addresses which exhibit temporal and spatial locality with signature hash functions can lead to false positive rates that are higher than what the theory predicts.

In the context of signatures for conflict detection in HTMs there are additional factors that dilute the theory of Bloom filters. First, in multi-processor systems there are two choices in how

to maintain cache-coherence, either broadcast or directory. A directory system has an advantage over broadcast systems in that it already maintains a fine-grained "filter" of readers or writer of cache lines. Cache request messages are routed only to processors which may have accessed a cache line instead of to all the processors in the system. Thus conflict detection need only be performed on the subset of processors indicated by the directory, which can greatly reduce false positives by reducing the frequency of signature checks. Lastly, the number of signature checks is directly related to the transactional duty cycle, the fraction of total execution time a program spends executing transactions. Higher transactional duty cycles imply more signature checks.

Even with all this reasoning one might wonder if the problem of false positives will actually be common in practice. The Birthday Paradox problem (described in the next sub-section), is a concrete example of how this problem can occur in everyday life.

2.3.3 Birthday Paradox Problem

This problem has its roots in probability theory and is stated as follows. What is the minimum number of people in a group such that the probability of any pair of people having the same birthday is greater than 50%? Surprisingly, it turns out that, contrary to intuition, this number is actually quite small: 23 people [121]. Zilles et al. [132] also examine this phenomenon, and provide both quantitative evidence and an analytic model which illustrate the implications of the Birthday Paradox for conflict detection in transactional memory.

The Birthday Paradox problem shows that it is possible for conflicts to exist even amongst a small number of participants. The implications of this are significant for multi-processor systems with large numbers of processors (or threads). Once the number of transactional threads passes a threshold value (which depends on properties of the system and workload) and signatures are

used for conflict detection, it is likely that there will be conflicts when signature checks are performed. Furthermore, depending on the signature's size and the number of elements inserted in the signature, these conflicts may be false conflicts rather than true conflicts.

In summary, we have shown how signatures can be used for conflict detection, and how signature false positives can be detrimental to system performance. Chapter 4 describes a signature-based HTM that enables the execution of transactions that run in arbitrary time and are of arbitrary size.

Chapter 3

Evaluation Methodology

This chapter presents the common evaluation methodology used for the dissertation.

3.1 Full-System Simulation Tools

We use full-system simulation to evaluate our proposed systems. Full-system simulation enables us to evaluate proposed systems running realistic workloads on top of actual operating systems. It also captures subtle timing affects not possible with trace-based evaluation. For example, different executions of the same program might cause the execution to take different code paths that would not be reflected in a single trace.

We use the Wisconsin GEMS simulation environment [70], which is based on Virtutech Simics [117]. Simics is a commercial product from Virtutech AB that provides full-system functional simulation of multiprocessor systems. GEMS is a set of modules that extends Simics with timing fidelity for our modeled system. GEMS consists of two primary modules: Ruby and Opal. Ruby models memory hierarchies and uses the SLICC domain-specific language to specify protocols. Opal models the timing of an out-of-order SPARC processor.

Support for modeling hardware transactional memory (HTM) systems is built on top of Opal and Ruby. The most recent version of GEMS as of this dissertation (version 2.1) contains support for multiple HTM systems implemented with a chip-multiprocessor (CMP). Section 3.4 contains additional details on which HTMs we simulate in this dissertation.

The compilation infrastructure for our transactional memory (TM) programs is built using an open-source C compiler (gcc) with no compiler source-code modifications. Language-level support for TM instructions (e.g., `tx_begin` and `tx_end`) is implemented through Simics "magic instructions". For each TM workload we manually add these magic instructions to simulate TM instructions, and then compile them using gcc. A magic instruction is a special assembly instruction (i.e., `sethi` to `%g0` in SPARC) that is functionally a no-op when executed on a real machine, but is intercepted by Simics and processed by the underlying timing modules (i.e., Opal and Ruby). Our timing modules implement the functionality of the associated TM instruction.

3.2 Methods

Evaluating HTM systems requires good metrics. We do not focus on traditional uniprocessor metrics such as instructions-per-cycle (IPC). IPC is not a good measure since it may not be representative of the actual useful work the system performs (e.g., spin loops in the operating system exhibit high IPC) [5].

Since our TM workloads are multi-threaded, we measure only the interesting parts of the program. We skip the program initialization (i.e., setting up per-thread data structures) and model the timing of the execution of the parallel phase of the program. Most of our programs execute to completion in the simulator. However, some workloads (e.g., our micro-benchmarks) are designed to run indefinitely.

For these workloads we terminate timing simulation after a certain number of high-level work units have completed. A work unit is workload-dependent and represents a high-level notion of useful work. Examples of work units within our workloads include one operation (e.g., insertion or delete) or the processing of one task. We annotate work units in the programs with additional

magic instructions. Our infrastructure catches these magic instructions and keeps track of how many work units have been performed.

The primary metric we care about is overall execution time. We define this as the time it takes to finish the parallel portion or some number of work units in our TM programs. We also examine secondary metrics. Secondary metrics focus on specific factors that directly or indirectly impact overall execution time. Examples include the read and write false positive rate of hardware signatures, and transaction-level statistics such as the read- and write-set sizes of transactions. Other metrics specific to each chapter will be discussed where appropriate.

Pseudo-random variations are added to each simulation run because of non-determinism in real systems [4]. This support is implemented by varying the memory latency in our infrastructure based on a random variable. We perform multiple simulations and compute the average runtime for each workload with an arithmetic mean. Error bars in our runtime results approximate a 95% confidence interval. To display the runtime results across multiple workloads, we show normalized runtime rather than raw cycles. Whenever appropriate, we include the raw cycle numbers in Appendices A-D.

3.3 Workload Descriptions

This section describes the workloads used in the evaluations of this dissertation. All workloads except BIND run on top of the Solaris 9 operating system. BIND runs on top of the OpenSolaris (Solaris 11) operating system.

- BTree Micro-benchmark: The BTree micro-benchmark represents a common class of concurrent data structures found in many applications. Each thread makes repeated accesses to a shared tree, randomly performing a lookup (with 60% probability) or an insert (40%). Both operations are implemented as transactions. The tree is a 9-nary B-tree initially 6 levels deep. We use per-thread memory allocators for scalability. We execute this micro-benchmark for 100,000 operations (lookups or inserts).
- Sparse Matrix Micro-benchmark: This benchmark corresponds to operations on a sparse matrix from a dense column vector multiplication kernel (spmxv). The sparse matrix uses Compressed Column Storage to store the nonzero entries. With this format the matrix multiplication by the column vectors implies an irregular array reduction in the body of a doubly nested loop, with the outer loop running over the compressed columns and the inner loop running over the nonzero column entries. Updates to this array in the innermost loop are protected by a transaction. Our experiments use the fidap011 matrix [18], a real non-symmetric 16,614 x 16,614 sparse matrix with 0.39% sparsity, from the SPARSKIT collection. We execute the entire parallel phase of this workload.
- LFUCache Micro-benchmark: The LFUCache micro-benchmark is based on the workload presented by Scherer et al. [98]. It uses common concurrent data structures, a hash table and a priority queue heap, to simulate cache replacement in a HTTP web proxy using the least frequently used (LFU) algorithm. The hash table holds pointers into the priority queue. Each thread in the micro-benchmark requests “pages” with a Zipf distribution and then updates the

cache, potentially replacing old data. The hash table is an array of 2k pointers and the priority queue is a fixed size heap of 255 entries (8-level deep binary tree) with lower frequency values near the root. We execute this micro-benchmark for 8,192 requests.

- Prefetch Micro-benchmark: This micro-benchmark is designed to test transactional contention at specific points in large transactions. Each thread performs transactions that are marked by three phases. The first phase performs reads on a large array. The second phase performs writes with 75% probability on a single cache line (small array). The last phase again performs reads on a large array. When all threads run concurrently, the contention occurs at the midpoint of the entire transaction. We execute this micro-benchmark for 496 operations (i.e., transactions).
- BerkeleyDB Micro-benchmark: BerkeleyDB is an open-source database storage manager library that is commonly used for server applications (such as OpenLDAP), database systems (MySQL), and many other applications. We base our workload on the open-source version distributed by Sleepycat software [106]. We convert the mutex-based critical-sections in BerkeleyDB to transactions. A simple multi-threaded driver program initializes a database with 1,000 words and then creates a group of worker threads that randomly read from the database. This driver stresses the BerkeleyDB lock subsystem due to repeated requests for locks on database objects. We execute this micro-benchmark for 128 database reads. This workload is only used in Chapter 4. Subsequent analysis of this workload revealed the driver program serialized threads and hurt program scalability. Hence we dropped this micro-benchmark in subsequent evaluations.

- Barnes, Cholesky, Mp3d, Radiosity, and Raytrace: These scientific programs are taken from the SPLASH [124] benchmark suite. They were selected because they show significant critical-section based synchronization. We replace the critical-sections with transactions while retaining barriers and other synchronization mechanisms. Raytrace was modified to eliminate false sharing between transactions. To reduce simulation times, we do not measure the entire parallel segment of the program for Cholesky, Mp3d, and Radiosity. Instead, we take representative sections of the program and measure performance in terms of work units. Cholesky, Radiosity, Raytrace, and Mp3d are used in the evaluation of Chapter 4. Overall, Barnes, Mp3d, and Raytrace exert the most pressure on hardware signatures, and we select these to represent SPLASH for evaluations in Chapters 5-7. In Cholesky, we simulate the factorization portion of the workload. In Radiosity, we execute 512 tasks. In Raytrace and Barnes, we execute the entire parallel phase. In Mp3d, we execute 512 steps.
- BIND: This workload is a transactional version of BIND 9.4.1 [54], which is a domain name service (DNS) server program. The lock-based BIND uses both mutex and reader-writer locks, and researchers incorporated better fine-grained locking techniques to eliminate poor scalability due to lock contention [56]. Through profiling with Solaris DTrace [75] we found that most of the application-level locking occurred in the internal red-black-tree structure which stores the zone structure information. We convert the tree manipulation code to coarse-grained transactions. DNSperf drives BIND by sending read-only DNS queries. Conflicts arise when worker threads contend to update per-node reference count fields in the tree. We execute BIND for 400 queries.

- Bayes, Delaunay, Genome, Intruder, Labyrinth, Vacation, Yada: These workloads are part of Stanford's STAMP suite [77] of transactional workloads. The key feature of this workload suite is that they contain much larger transaction sizes than the other TM workloads we use in our evaluations. These workloads stress the hardware signatures more than our other workloads. We do not perform evaluations with the remaining STAMP workloads (Kmeans and Ssca2) because they do not stress hardware signatures enough to yield interesting behaviors. The versions of Vacation and Genome used for our evaluations are modified from the original by converting coarse-grained transactions to fine-grained transactions. For Genome we use a chunk size of one to achieve fine-grained transactions. For Vacation we partition a client's travel request into smaller transactions. These changes improve workload scalability for smaller signature sizes.

TABLE 3-1. Workload parameters

Benchmark	Input	Unit of Work	Units Measured
BerkeleyDB	1,000 words	Database read	128
BTree	Uniform random	Insert or lookup	100,000
Sparse Matrix	fidap011	Whole parallel phase	1
LFUCache	8,192 ops	Page request	8,192
Prefetch	1 element small array, 8,192 element big array	Transaction	496
Barnes	1,024 bodies	Whole parallel phase	1
Raytrace	teapot	Whole parallel phase	1
Mp3d	128 molecules	Step	512 or 1,024
Cholesky	tk14.O	Factorization	1
Radiosity	batch	Task	512
Vacation	High contention	Whole parallel phase	1
Genome	-g256 -s16 -n16384	Whole parallel phase	1
Delaunay	-i inputs/gen4.2 -m 30	Whole parallel phase	1
Bayes	-v32 -r1024 -n2 -p20 -s0 -q1.0 -i2 -e2	Whole parallel phase	1
Labyrinth	random-x32-y32-z3-n64.txt	Whole parallel phase	1
Intruder	-a10 -l16 -n2038 -s1	Whole parallel phase	1
Yada	-a20 -i inputs/633.2	Whole parallel phase	1
BIND	DNSPerf driver queries	Query	400

Table 3-1 summarizes our workload parameters, what each workload's work unit is, and how many work units we measure for each workload.

TABLE 3-2. Workload characteristics for workloads in Chapter 4

Benchmark	Transactions	Avg. Read-Set	Avg. Write-Set	Max Read-Set	Max Write-Set
BerkeleyDB	1,120	8.1	6.8	30	28
Cholesky	261	4.0	2.0	4	2
Radiosity	11,172	2.0	1.5	25	45
Raytrace	47,781	5.8	2.0	550	3
Mp3d	17,733	2.2	1.7	18	10

We also present workload characteristics for the workloads in this dissertation. This includes the total number of transactions in the workload, the average read- and write-set sizes, as well as the maximum read- and write-set sizes of the workload. All read- and write-set sizes are given in number of 64B cache lines. Table 3-2 gives the workload characteristics for workloads in Chapter 4, which execute with 31 transactional threads. Table 3-3 gives the workload characteristics for the workloads in Chapters 5-7. Prefetch and BIND execute with 31 transactional threads, and the remaining workloads in the table execute with 15 transactional threads.

TABLE 3-3. Workload characteristics for workloads in Chapters 5-7

Benchmark	Transactions	Avg. Read-Set	Avg. Write-Set	Max Read-Set	Max Write-Set
BTree	100,000	13.4	1.3	21	16
Sparse Matrix	1,091,362	5.0	1.0	5.0	1.0
LFUCache	8,190	5.4	2.2	25	19
Prefetch	496	154	2	158	2
Barnes	2,364	6.1	4.6	40	40
Raytrace	47,765	5.3	2.0	596	3.0
Mp3d	35,030	1.6	1.2	12	7
Vacation	30,011	25	3.2	127	25
Genome	20,930	15	1.7	154	18
Delaunay	2,146	68	44	610	420
Bayes	550	73	37	2,061	1,597
Labyrinth	158	70	91	269	264
Intruder	27,108	9.5	2.5	69	26
Yada	5,167	61	37	704	457
BIND	5,327	6.4	5.0	50	28

TABLE 3-4. System model parameters

	System Model Settings
Processor Cores	5 GHz Chapter 4: Out-of-order, 2-way SMT Chapters 5-7: In-order, single-issue
L1 Cache	32 KB 4-way split, 64-byte blocks, 1 cycle uncontended latency
L2 Cache	8 MB 8-way unified, 64-byte blocks, 34-cycle uncontended latency
Memory	4 GB 500-cycle latency
Cache Coherence	MESI directory protocol
L2-Directory	Full-bit vector sharer list; 6-cycle latency
Interconnection Network	Grid, 64-byte links, 3-cycle link latency
Signatures	Perfect and 64b-64kb imperfect
HTM Systems	LogTM-SE [127], an approximation of Stanford's TCC [43]

3.4 Modeling a CMP with GEMS

Our simulations attempt to capture the first-order affects of HTMs implemented on a single CMP chip, including all messages required to implement the protocol on a given interconnect.

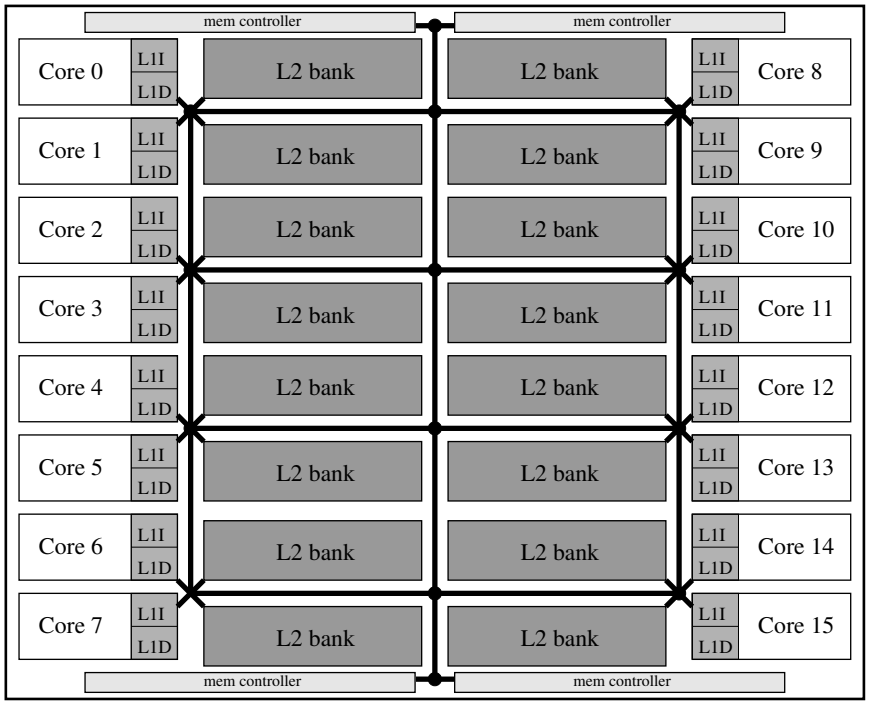


FIGURE 3-1. 16-processor CMP system

Table 3-4 summarizes the system model parameters that are used throughout this dissertation. Where appropriate, individual chapters describe additional idealizations of certain parameters in the HTMs. Figure 3-1 illustrates the 16-processor CMP system that we model in this dissertation.

The CMP memory model consists of various controllers connected via network links in a specified topology. Processor models interface with an L1 cache controller. L1 cache controllers then interact with other controllers (i.e., directory/memory controller) and interconnect links to model the timing of an L1 miss. Most timing is modeled by a controller specifying the delay of when a message is injected into the network, and the delay incurred through modeling the delivery of the message. Details of how the major components of the system are modeled are as follows:

Processors. We model both simple in-order and more aggressive out-of-order SPARC processing cores. The in-order model assumes every instruction executes in a single cycle barring any stall time in the memory system. The out-of-order model is loosely based on the MIPS R10000 [126] and further described in Mauer et al. [72]. Each out-of-order processor core also supports simultaneous multi-threading (SMT) [35]. Chapter 4 uses 16 out-of-order cores and Chapters 5-7 use 16 or 32 in-order cores. BIND and Prefetch use 32 cores (due to better workload scalability and higher contention, respectively), and the remaining workloads use 16 cores.

Caches and Cache Controllers. All evaluations in this dissertation use 32 KB, 4-way set-associative L1 split instruction and data caches. The L2 cache is unified, 8MB, 8-way set-associative and is shared amongst all processing cores. Both caches are write-back and implement perfect LRU cache replacement. We do not constrain the cache lookup bandwidth and instead model a fixed access latency.

A cache controller's behavior is specified using the SLICC specification language [108,69]. SLICC allows users to specify the events the cache controller handles. Events trigger transitions, which consist of a set of atomic actions. Actions can update state in the controller and also inject messages into the network. While we do limit the number of outstanding transactions a controller can handle, we do not model the detailed pipeline of the controller logic.

Directory Controllers and Memory. We model idealized memory controllers such that every access incurs a fixed delay (detailed in each chapter), plus a random component added to account for workload variability. The random component is a uniform random number between zero and six cycles. All other aspects of DRAM, including bandwidth, are idealized.

Interconnect. We use the default GEMS' networking model to approximate all of the target interconnection networks. For each target CMP, we specify the grid network topology using a configuration file. The file specifies the endpoints of the interconnect as well as the links between network switches. Each link is specified with fixed latency and bandwidth parameters. A message always incurs the latency specified plus any additional queuing delay due to insufficient bandwidth.

Signatures. We model the functionality of hardware signatures for conflict detection in HTMs and idealize its storage and latency overheads. We evaluate two types of signatures: perfect and imperfect. Perfect signatures are un-implementable, because they precisely track the block addresses of all read and write accesses within transactions. Imperfect signatures are finite hardware structures that imprecisely track transactional read and write accesses. Although program executions with perfect and imperfect signatures may not match (due to different thread interleavings and code paths), our evaluations compare the program execution times with perfect and

imperfect signatures since this comparison captures the first-order performance impact of using imperfect signatures.

Cache Coherence Protocol. All CMPs use a directory-based cache coherence protocol with the MESI states. The L1 and L2 caches are inclusive, and the directory maintains a full list of sharers or the exclusive owner of each cached block. The protocol supports silent-S replacements (L1 replacements of clean data). Other replacements of valid L1 blocks must inform the L2.

All HTMs rely on this protocol to detect conflicts amongst transactions (using hardware signatures). In the absence of SMT, signatures on remote processors are checked on L1 cache misses. Each hardware thread context in a SMT processor core must also check the signatures of other thread contexts on the same processor core on every L1 access. A remote processor signals a signature conflict through a Negative Acknowledgement (NACK) message which is sent to the requestor.

The Sticky-M (or Sticky-S) directory state [80] is set whenever a transactionally modified (or read) block is replaced from the L1 and inserted into the L2 cache. The L2 directory will selectively send out write or read and write signature checks depending on the state of a cache block in the system. Any non-cached block, for correctness, has to check both read and write signatures on all processor cores.

HTM Systems. We simulate two HTM systems: LogTM-SE (Chapter 4) and an approximation of Stanford's TCC [43]. We idealize the register checkpoint overheads (both storage and access latency) in both HTMs. None of our workloads require the transaction virtualization capabilities of LogTM-SE (i.e., there are no system calls or I/O, no context-switches, and no page remapping within transactions).

Where appropriate, each of the evaluations in the subsequent chapters elaborate on more specific details of the above components.

Chapter 4

LogTM-SE: Decoupling Hardware Transactional Memory from Caches

This chapter proposes a hardware transactional memory (HTM) system called *LogTM Signature Edition (LogTM-SE)*. LogTM-SE uses *signatures* to summarize a transaction's read- and write-sets and detects conflicts on coherence requests (eager conflict detection). Transactions update memory "in place" after saving the old value in a per-thread memory *log* (eager version management). Finally, a transaction commits locally by clearing its signature, resetting the log pointer, etc., while aborts must undo the log.

LogTM-SE achieves two key benefits. First, signatures and logs can be implemented without changes to highly-optimized cache arrays because LogTM-SE never moves cached data, changes a block's cache state, or flash clears bits in the cache. Second, transactions are more easily virtualized because signatures and logs are software accessible, allowing the operating system and runtime to save and restore this state. In particular, LogTM-SE allows cache victimization, unbounded nesting (both open and closed), thread context switching and migration, and paging.

4.1 Introduction

Hardware accelerates transactional memory with two key capabilities. First, hardware provides *conflict detection* among transactions by recording the read-set (addresses read) and write-set (addresses written) of a transaction. A conflict occurs when an address appears in the write-set

of two transactions or the write-set of one and the read-set of another. Second, hardware provides *version management* by storing both the new and old values of memory written by a transaction. Most HTMs achieve their good performance in part by making demands on critical L1 cache structures. These demands include *read/write* (R/W) bits for read- and write-set tracking [7,43, 80,93], flash clear operations at commits/aborts [7,43,80,93], and write buffers for speculative data [23,43]. In addition, some depend on broadcast coherence protocols, precluding implementation on directory-based systems [23].

We see three reasons future HTMs may wish to decouple version management and conflict detection from the L1 cache tags and arrays. First, these are critical structures in the design of high performance processors that are better left untouched by an emerging idea like transactional memory. Second, the desire to support both T-way multi-threaded processors and L-level nested transactions leads to T x L copies of the state. Third, having transactional state integrated with the L1 cache makes it more difficult to save and restore, a necessary step to *virtualize* transactional memory—i.e, support cache victimization, unbounded nesting, thread suspension/migration, and paging [7,93].

Fortunately, two HTMs provide complementary partial solutions to decoupling HTM demands from L1 caches.

LogTM [80] decouples version management from L1 cache tags and arrays. With LogTM, a transactional thread saves the old value of a block in a per-thread log and writes the new value in place (eager version management). LogTM's version management uses cacheable virtual memory that is *not* tied to a processor or cache. It never forces writebacks to cache speculative data, because it does not exploit cache *in*coherence, e.g., where the L1 holds new transactional values

and the L2 holds the old versions [43,7,23]. Instead, caches are free to replace or writeback blocks at any time. No data moves on commit, because new versions are in place, but on abort a handler walks the log to restore old versions. LogTM, however, fails to decouple conflict detection, because it maintains R/W bits in the L1 cache.

Bulk [23] decouples conflict detection by recording read- and write-sets in a hashed signature separate from L1 cache tags and arrays. A simple 1K-bit signature might logically OR the decoded 10 least-significant bits of block addresses. On transaction commit, Bulk broadcasts the write signature and all other active transactions compare it against their own read and write signatures. A non-null intersection indicates a conflict, triggering an abort. Due to aliasing, non-null signature intersection may occur even when no actual conflict exists (a false positive) but no conflicts are missed (no false negatives). Moreover, Bulk's signatures make it easier to support multi-threading and/or nested transactions since replicating signatures doesn't impact critical L1 structures. Bulk's version management, however, is still tied to the L1 cache: the cache must (i) write-back committed, but modified blocks before making speculative updates, (ii) save speculatively modified blocks in a special buffer on cache overflow, and (iii) only allow a single thread of a multi-threaded processor to have speculative blocks in any one L1 cache set. In addition, it depends on broadcast coherence for strong atomicity [14] and requires global synchronization for ordering commit operations.

LogTM-SE. In this chapter, we propose *LogTM Signature Edition* (*LogTM-SE)*, which decouples both conflict detection and version management from L1 tags and arrays. LogTM-SE combines Bulk's signatures and LogTM's log, but adapts both to reap synergistic benefits. With LogTM-SE, transactional threads record conflicts with signatures and detect conflicts on coherence requests.

Transactional threads update memory in place after saving the old value in a per-thread memory log. Like LogTM, LogTM-SE does not depend on broadcast coherence protocols. Finally, a transaction commits locally by clearing its signature and resetting its log pointer—there are no commit tokens, data writebacks, or broadcast—while aborts locally undo the log.

Transactions in LogTM-SE are virtualizable, meaning that they may be arbitrarily long and can survive OS activities such as context switching and paging, because the structures that hold their state are software accessible. Both old and new versions of memory can be victimized transparently because the cache holds no inaccessible transactional state. Similarly, the ability to save and restore signatures allows unbounded nesting. LogTM-SE achieves this using an additional *summary signature* per thread context to summarize descheduled threads. Finally, LogTM-SE supports paging by updating signatures using the new physical address after relocating a page.

Using Simics [68] and GEMS [70] to evaluate a simulated transactional CMP, we show that LogTM-SE performs comparably with the less-virtualizable, original LogTM. Furthermore, for our workloads even very small (e.g., 64 bit) signatures perform comparably or better than locking.

In our view, LogTM-SE contributes an HTM design that (1) leaves L1 cache state, tag, and data arrays unchanged (no in-cache R/W bits or transactional write buffers), (2) has no dependence on a broadcast coherence protocol, (3) effectively supports systems with multi-threaded cores (replicating small signatures) on one or more chips (with local commit), and (4) supports virtualization extensions for victimization, nesting, paging, and context switching because signatures are easily copied. In Section 4.9 we detail how LogTM-SE differs from existing HTMs.

4.2 LogTM-SE Architecture

This section describes the LogTM-SE architecture, while Section 4.5 develops a specific example LogTM-SE system.

Tracking Read- and Write-Sets with Signatures. LogTM-SE tracks *read- (R)* and *write- (W)* sets with conservative signatures inspired by Bulk, as well as others who conservatively encode sets [12,82,87,100]. A *signature* implements several operations. Let O be a read or a write and A be a block-aligned physical address. INSERT(O, A) adds A to the signature's *O-set*. Every load instruction invokes INSERT(read, A) and every store invokes INSERT(write, A). CONFLICT(read, A) returns whether A *may* be in a signature's write-set (thereby conflicting with a read to A). CONFLICT(write, A) returns whether A *may* be in a signature's read- or write-sets. Both tests may return false positives (report a conflict when none existed), but may not have false negatives (fail to report a conflict). Finally, CLEAR(O) clears a signature's O-set. Section 4.5 discusses specific signature implementations.

Eager Conflict Detection. LogTM-SE performs eager conflict detection like LogTM, except that LogTM-SE uses signatures (not read/write bits in the L1 caches) and handles multi-threaded cores. Consider conflict detection with single-threaded cores first. A load (store) that misses to block A generates a GETS(A) (GETM(A)) coherence request. A core that receives a GETS (GETM) request checks its write (read and write) signatures using a CONFLICT(read, A) (CONFLICT(write, A)) operation. A core that detects a possible conflict responds with a negative acknowledgement (NACK). The requesting core, seeing the NACK, then resolves the conflict. LogTM-SE adopts LogTM's conflict resolution mechanism: the core stalls, retries its coherence

operation, and aborts on a possible deadlock cycle. More sophisticated future versions could trap to a contention manager.

LogTM-SE forbids a core's L1 cache from caching a block (no M, O, E, or S coherence states) that is in the write-set of a transaction on another core. Nor may it exclusively cache a block (no M or E) that is in the read-set of a transaction on another core. (Note that a core may, however, cache data that is in the read- or write-set signature of another core due to aliasing.) This provides isolation by ensuring that data written by one transaction cannot be read or written by others before commit. With the above invariants, loads that hit in the core's L1 cache (states M, O, E, or S) and stores that hit (M or E) need no signature tests. Significantly, LogTM-SE does *not* enforce the converse of these invariants—a block in a transaction's read- or write-set need not be locally cached. Section 4.3.1 discusses how LogTM-SE cores check the signature for blocks evicted from its L1, which allows victimization of transactional data.

Signatures have the potential to cause interference between memory references in different processes. If thread t_a in process A running on core C1 accesses a memory block residing on core C2, which is running t_b from process B, a signature on C2 may signal a false conflict. While not affecting correctness, this interference could allow one process to prevent all other processes from making progress. LogTM-SE prevents this problem by adding an address space identifier to all coherence requests. Requests are only NACKed if the signature signals a potential conflict *and* the address space identifiers match, preventing false conflicts between processes.

Multi-threaded cores require additional mechanisms to detect conflicts among threads on the same core. Each *thread context* maintains its own read and write signatures. Loads or stores to blocks in M (and stores to E) must query the signatures of other threads on the same core. This

check should not impact performance because conflicts need only be detected before the memory instruction commits.

Eager Version Management. LogTM-SE adopts LogTM's per-thread log, but adds a new mechanism to suppress redundant logging. Like a Pthread's stack, the log is allocated in thread-private memory. Before a memory block is first written in a transaction, its virtual address and previous contents must be written to the log. It is correct, but wasteful, to write the same block to the log more than once within a transaction. LogTM reuses the W bit in the L1 cache, which records whether a block has been written by the active transaction, to suppress redundant logging. However, this optimization does not extend to LogTM-SE because signatures permit false positives. If the hardware fails to log a block due to a false positive in the write-set signature, it would be impossible to correctly undo the effects of a transaction.

Instead, LogTM-SE uses an array of recently logged blocks for each thread context as a simple but effective log filter. When a thread stores to a block *not* found in its log filter, LogTM-SE logs the block and adds its address to the log filter. Stores to addresses in the log filter are not logged. Much like a TLB, the array can be fully associative, set associative, or direct mapped and use any replacement algorithm. As with write buffers in multi-threaded cores, the filters are logically per-thread, but can be implemented in a tagged shared structure. Because the filter contains virtual addresses and is a performance optimization not required for correctness, it is always safe to clear the log filter (e.g., on context switch).

Local Commit & Abort. LogTM-SE's transactional commit is a fast, local operation that also avoids LogTM's flash-clear of L1 cache read/write bits. To commit, a thread must only clear its local signatures to release isolation on its read- and write-sets and reset its log pointer. Since eager

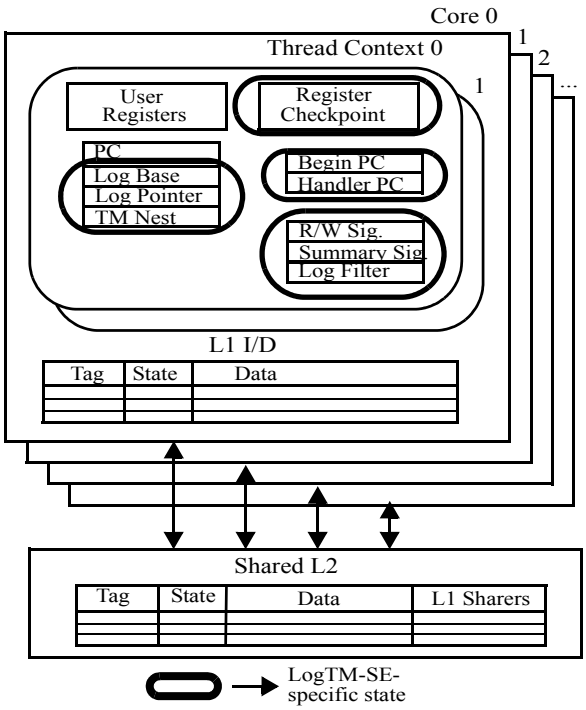


FIGURE 4-1. LogTM-SE hardware overview.

version management updates data in place, no data movement is necessary. Thus, commit, which should be much more common than abort, is a fast, thread-local operation requiring no communication or synchronization with other threads or cores. Like LogTM, LogTM-SE permits multiple non-conflicting transactions to commit in the same cycle.

LogTM-SE implements abort, the uncommon case, using a software handler. A thread aborts a transaction by trapping to an abort handler, which first walks the log in last-in-first-out (LIFO) order to restore transactionally modified blocks. Once memory is restored to pre-transaction values, the handler releases isolation by clearing the thread's signature. Although abort takes time proportional to the number of blocks written by a transaction, it does not require any global resources.

Summary. The circled items in Figure 4-1 illustrate what LogTM-SE adds to each thread context to support TM. Like LogTM, LogTM-SE adds a register checkpoint and registers to store the log

address, nesting depth, and abort handler address. LogTM-SE also adds two signatures, a log filter, and a summary signature (described in Section 4.4.1), but makes no changes to the critical L1 and L2 caches and has no structures that explicitly limit transaction size.

4.3 Virtualizing LogTM-SE

Application programmers reason about threads and virtual memory, while hardware implements multi-threaded cores, caches, and physical memory. Operating systems (OSs) provide programmers with a higher-level abstraction by virtualizing physical resource constraints, such as memory size and processor speed, using mechanisms such as paging and context switching. To present application programmers a suitable abstraction of transactional memory, the OS must *virtualize* the HTM's physical resource limits, using hardware and low-level software mechanisms that are fast in common cases, correct in all cases, and, if possible, simple [7,93].

This section discusses how LogTM-SE efficiently executes transactions unbounded in size and nesting depth using limited hardware. The following section discusses context switching and paging. LogTM-SE has two key advantages with regard to virtualization. First, LogTM-SE's version management is naturally unbounded, since logs are mapped into per-thread virtual memory. Second, LogTM-SE's signatures and logs are software accessible, allowing software to save and restore signatures to/from the log.

4.3.1 Cache Victimization

Caches may need to evict transactional blocks when a transaction's data size exceeds cache capacity or associativity. Multi-threaded cores make this more likely and unpredictable, due to interference between threads sharing the same L1 cache. Furthermore, after eviction, an HTM

must continue to efficiently handle both version management and conflict detection. This is important, since cache victimization is likely to be more common than other virtualization events (e.g., thread switching and paging).

Notably, cache victimization has *no effect* on LogTM-SE's version management. Like LogTM, both new values (in place) and old values (in the log) may be victimized without resorting to special buffers, etc.

LogTM-SE's mechanism for conflict detection depends upon the underlying cache coherence protocol. Like all HTMs with eager conflict detection, LogTM-SE relies on the coherence protocol to direct requests to all caches that might represent a conflict. With broadcast coherence, cache victimization has no effect on conflict detection, because LogTM-SE can check all signatures on every broadcast.

With a naive directory protocol, cache victimization could lead LogTM-SE to miss some signature checks and hence miss some conflicts. LogTM-SE avoids this case by extending the directory protocol to use LogTM's *sticky* states [80]. As in many MOESI protocols, LogTM-SE's caches silently replace blocks in states E and S and write back blocks in states M and O. When evicting a cache block (e.g., core C1 replaces block B), however, LogTM-SE *does not change the directory state*, so that the directory continues to forward conflicting requests to the evicting core (e.g., a conflicting operation by C2 is still forwarded to C1, which checks its signature). Thus, LogTM-SE allows transactions to overflow the cache without a loss in performance.

4.3.2 Transactional Nesting

To facilitate software composition, HTMs must allow transactional nesting: invoking a transaction within a transaction [83]. This is trivially done by *flattening*: only committing transactional state when the outer-most transaction commits. Unfortunately with flat nesting, a conflict with the inner-most transaction forces a complete abort all its ancestors as well. An improvement is *closed nesting with partial aborts* that, for the above case, would allow an abort of just the inner-most transaction. To increase concurrency, some also argue for *open nesting* [120] which allows an inner transaction to commit its changes and release isolation before the outer transactions commit. In addition, some proposed language extensions for transactional memory, such as *retry* and *orelse*, depend on arbitrarily deep nesting [45]. Ideally, HTMs should provide unbounded nesting to fully support these language features. Otherwise, some composed software may fail when transactions nest too deeply.

LogTM-SE supports unbounded transactional nesting with no additional hardware by virtualizing the state of the parent's transaction while a child transaction is executing. Following Nested LogTM [81], LogTM-SE segments a thread's log into a stack of frames, each consisting of a fixed-sized header (e.g., register checkpoint) and a variable-sized body of undo records. LogTM-SE augments the header with a fixed-sized signature-save area.

A nested transaction begins by saving the current thread state: LogTM-SE copies the signature to the current transaction's log frame header and allocates a new header with a register checkpoint. To ensure the child correctly logs all blocks, it clears the log filter. Loads and stores within the child transaction behave normally, adding to the signature and log as necessary. On commit of a closed transaction, LogTM-SE merges the inner transaction with its parent by discarding the

inner transaction's header and restoring the parent's log frame. An open commit behaves similarly, except that it first restores the signature from the parent's header into the (hardware) signature to release isolation on blocks only accessed by the committing open transaction.

On an abort, LogTM-SE's software handler first unrolls the child transaction's log frame and restores the parent's signature. If this resolves the conflict, the partial abort is done and a retry can begin. If a conflict remains with the parent's signature, the handler repeats this process until the conflict disappears or it aborts the outer-most transaction.

LogTM-SE supports unbounded transactional nesting with a per-thread hardware signature, saved to the log on nested begins. To reduce overhead, each thread context could provide one or more extra signatures to avoid synchronously saving and restoring signatures. On a nested begin, for example, hardware can copy the current signature S to S_{backup}. Inner commit of a closed transaction discards S_{backup}, while inner commit of an open transaction and all inner aborts restore S_{backup} to S. Like register windows, the benefit depends on program behavior.

4.4 OS Resource Management

While OS resource management events, such as context switches and paging, may be infrequent relative to the duration of a transaction, they must still be handled correctly. This section discusses how LogTM-SE allows threads executing in transactions to be suspended and rescheduled on other thread contexts and how pages accessed within a transaction can be relocated in memory.

4.4.1 Thread Suspension/Migration

Operating systems (OSs) increase processing efficiency and responsiveness by suspending threads and rescheduling them on any thread context in the system. To support thread context switch and migration, the OS must remove all of a thread's state from its thread context, store it in memory, and load it back, possibly on a different thread context on the same or a different core. For HTMs that rely on the cache for either version management or conflict detection, moving thread state is difficult because the transactional state of a thread is not visible to the operating system. One simple approach is to abort transactions when a context switch occurs. This is difficult for eager version management HTMs, though, because aborting is not instantaneous. In addition, some long-running transactions may never complete if they are forced to abort when preempted. A better approach allows thread preemption, but ensures that transactional state is saved and restored with the thread's other state.

In LogTM-SE, all of a thread's transactional state—its version management and conflict detection state—is accessible to the OS. Both old and new versions of transactional data reside in virtual memory and require no special OS support. The log filter is purely an optimization and can be cleared when a thread is descheduled.

A thread's conflict detection state can be saved by copying the read/write signatures to the log's current header. We call the copies of the signatures that are saved to the log header the *saved signatures*. However, the hardware must continue to track conflicts with the suspended thread's signatures to prevent other threads from accessing uncommitted data. For example, another thread in the same process may begin a transaction on the same thread context and try to access a block in its local cache. The system must check this access to ensure that the block is not in the write-set

of a descheduled transaction. The challenge is to ensure that all active threads check the signatures of descheduled threads in their process on every memory reference.

LogTM-SE achieves this goal using an additional *summary signature*, which represents the union of the suspended transactions' read- and write-sets. The OS maintains the following invariant for each active/summary signature pair: *If thread t of process P is scheduled to use an active signature, the corresponding summary signature holds the union of the saved signatures from all descheduled threads from its process P.* On every memory reference, including hits in the local cache (both transactional and non-transactional), LogTM-SE checks the summary signature to ensure that the request does not conflict with a descheduled transaction. Multi-threaded cores, where each thread on a core may belong to a separate process, require a summary signature per thread context.

The OS maintains, in software, a summary signature for the entire process. When descheduling a thread, the OS merges the thread's saved signatures into its process summary signature. It then interrupts all other thread contexts running threads from the process and installs the new summary signature. These thread contexts may or may not be running threads with virtualized transactions, but they all still receive the new summary signature. Descheduled threads from the process do not receive the new summary signature, as they cannot make any memory references. In contrast to the normal signature, the summary signature is checked on memory references but not on coherence requests (because it is present on all active thread contexts running in the same process). Any memory request that conflicts with a saved signature immediately traps to a conflict handler, since stalling is not sufficient to resolve a conflict with a descheduled thread.

When the OS reschedules a thread, it copies the thread's saved signatures from its log into the hardware read/write signatures. However, the summary signature is not recomputed until the thread commits its transaction, to ensure that blocks in sticky states remain isolated after thread migration. The thread executes with a summary signature that does not include its own signatures, to prevent conflicts with its own read- and write-sets. On transaction commit, LogTM-SE traps to the OS, which pushes an updated summary signature to active threads.[1]

Thus, with a single additional signature per thread context and small changes to the operating system, LogTM-SE supports both context switching and thread migration. The cost of context switching within a transaction is relatively high, and for that reason we expect operating systems to support preemption control mechanisms [114] that defer context switches occurring within a transaction if possible. In addition, aborting short transactions may be preferable to incurring the overhead of propagating new summary signatures.

4.4.2 Virtual Memory Paging

HTMs must support paging of transactional data for several reasons. First, an OS may page out data in the read- and write-sets of active transactions and page it back in at a different physical address. If transactions are short, swapping of transactional data to disk is unlikely, because the memory was touched recently. However, paging may be required because one or more transactions' read- or write-sets exceed the physical memory size (but we hope this case is uncommon). Second, OS techniques, such as copy-on-write, may also cause a page that was read to subse-

1. To efficiently compute summary signatures, the OS could maintain a *counting signature* data structure to track the number of suspended threads setting each summary signature bit, similar to VTM's XF data structure [93].

quently be relocated when it is written. HTMs should therefore work correctly in the presence of paging and should not cause an automatic abort (to handle large transactions).

LogTM-SE's version management operates on virtual addresses and is not tied to cores or caches. Thus, both new (in place) and old (in log) versions can be transparently paged. Moreover, eager version management allows a transaction to commit without restoring paged-out pages, since the new version is already in place. In contrast, lazy version management, in which memory is updated on commit, would require restoring paged-out pages at commit time, removing any benefit of paging them out in the first place.

LogTM-SE's signatures do not lose any information when a page is removed from memory, so transactional data remains isolated. However, because signatures operate on physical addresses, false conflicts may arise if the page is remapped to a different virtual address within the same address space. As with other false positives, this is acceptable if it is infrequent (as it should be).

More important are false negatives, indicating loss of isolation, that can arise when *all* of the following hold: (a) a page was transactional, (b) was paged out, (c) then paged back in at a different physical address (d) while the original transaction was still active. Since paging transactional data should be very rare, we propose a correct solution and leave optimization to future work.

When bringing a page back into a process at a different physical address, LogTM-SE notifies all threads to update their signatures with the new physical address for the page. For active threads, this requires interrupting each thread and, for those executing a transaction, walking the signature and testing whether it contains any blocks from the old address of the page. If so, the same blocks are inserted in the signature using their new physical address. The OS queues a signal

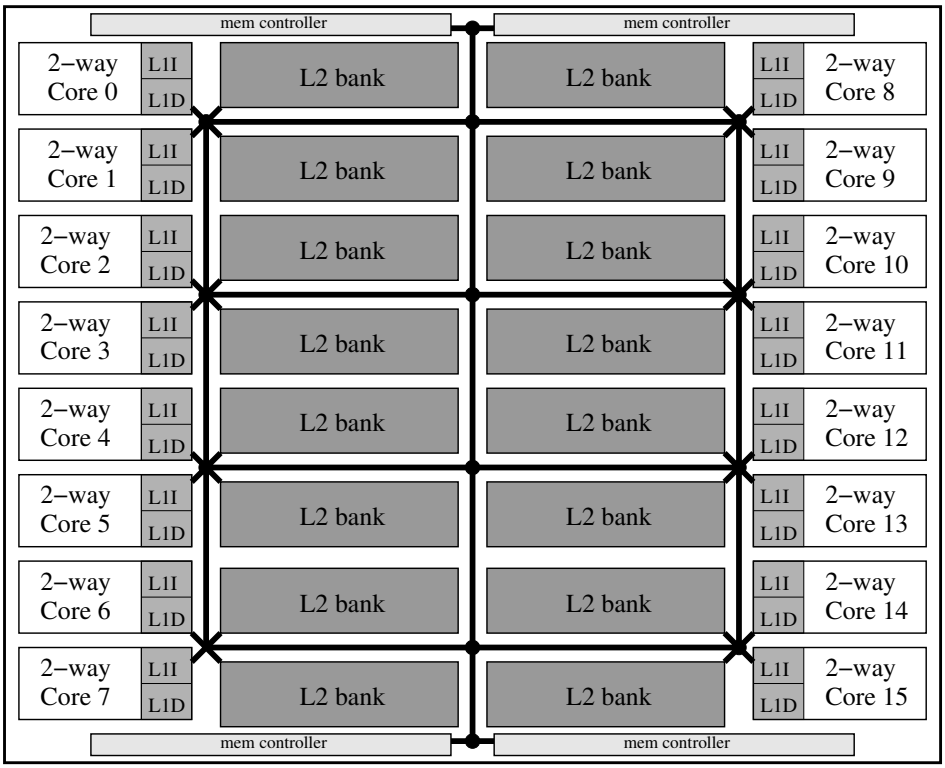


FIGURE 4-2. Baseline CMP for LogTM-SE.

for descheduled transactions to update their summary signatures (as well as signatures in the log from nesting) before they resume execution. Thus, the updated signatures contains both the old and new physical addresses for read- and write-set elements on the page.

This simple mechanism requires no additional hardware support and will incur little overhead if paging within a transaction is rare. If paging proves more frequent (i.e. if large transactions become the norm), additional mechanisms can detect whether a page has been touched during a transaction to avoid unnecessary signature updates.

4.5 A LogTM-SE Implementation

This section presents a specific LogTM-SE implementation for a CMP with non-broadcast coherence, which will be important for future larger-scale CMPs.

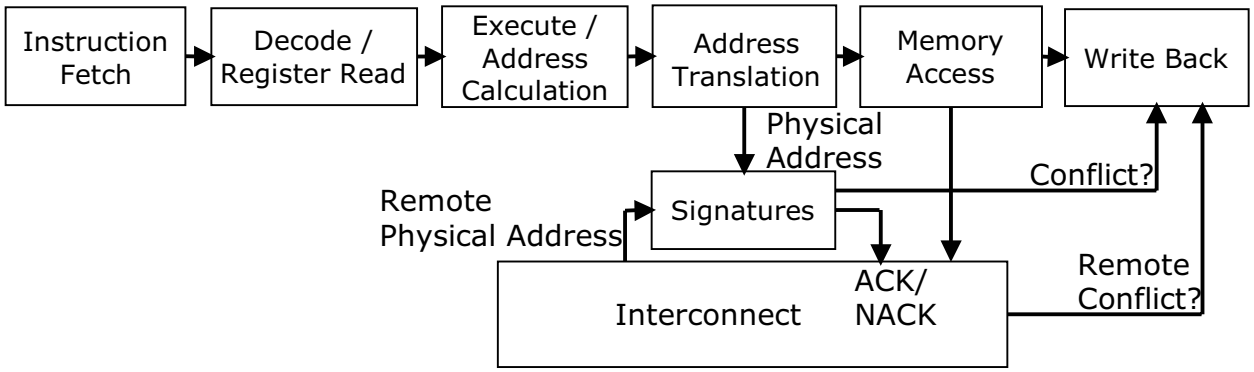


FIGURE 4-3. How signatures fit into a processor's pipeline.

Base CMP. Figure 4-2 illustrates the baseline 16-core LogTM-SE system and Table 3-4 from Chapter 3 summarizes the system parameters. Each of the 16 cores executes instructions out-of-order and supports 2-way multi-threading, providing 32 thread contexts on chip. The cores are 4-way-issue superscalar, use a 15-stage pipeline, 64-entry issue window, 128 entry reorder buffer, YAGS branch predictor, and have abundant fully-pipelined functional units (2 integer ALU, 2 integer divide, 2 branch, 4 FP ALU, 2 FP multipliers, and 2 FP divide/square root). The number of functional units are chosen so that these resources are not a bottleneck for the processor. This allows us to focus on bottlenecks in the memory subsystem, which includes the conflict detection mechanisms in LogTM-SE. The parameters of the memory subsystem are described in Chapter 3.

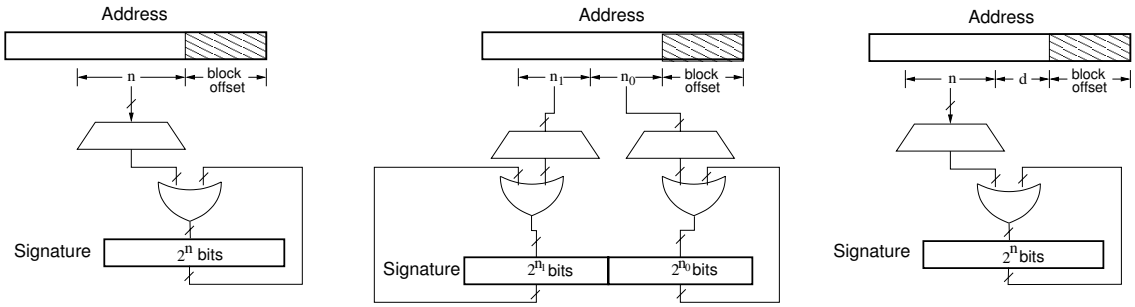
Figure 4-3 shows how signatures fit into a processor's pipeline. We illustrate the major pipeline stages, and include address translation from virtual to physical addresses as a separate stage. The operation of signatures in the pipeline works as follows. Physical addresses from addresses calculated by the processor are routed to the signatures (which comprise of physical and summary signatures) for lookup operations. A hardware thread context checks the physical signatures belonging to other hardware thread contexts only if the processor supports multiple hardware thread contexts (i.e., for detecting conflicts in SMT processors). The signatures' lookup opera-

tions indicate whether a conflict exists, and this conflict information is encoded in a signal that is sent to the write back stage and stored with the write back state associated with the memory instruction. The design of signatures and its associated wires must be such that a memory instruction is not allowed to retire unless the conflict information is available for the instruction. Only non-conflicting instructions are allowed to retire. Signatures also take as input physical addresses sent by remote processors. These remote addresses also check for conflicts with a processor's signatures. Any conflicts with remote addresses is encoded as ACKs or NACKs that is sent on the interconnect. These responses are turned into signals associated with the memory instruction on the remote processor indicating whether a conflict occurred due to signature lookups on remote processors. Thus a memory instruction can have transactional conflicts due to signatures on its own processor or due to signatures on remote processors.

A MESI directory protocol provides cache coherence with less bandwidth demand than a broadcast protocol. The protocol enforces inclusion and each L2 tag contains a bit-vector of the L1 sharers and a pointer to the exclusive copy, if it exists. To eliminate a potential race, an E replacement from an L1 cache sends a control message to update the exclusive pointer, but S replacements are completely silent. An alternative implementation of the on-chip directory could use shadow tags instead of an inclusive L2 cache.

Coherence Protocol Changes. LogTM-SE modifies the baseline MESI coherence protocol to support CONFLICT(O, A) operations. GETS(A) requests from other cores invoke CONFLICT(read, A) and GETM(A) requests invoke CONFLICT(write, A).

If an L1 cache replaces transactional data, the L2 cache does *not* update the exclusive pointer or sharer's list (like the sticky-S and sticky-M states in LogTM [80]). This ensures that subse-



(a) bit−select (BS) (b) double−bit−select (DBS) (c) coarse−bit−select (CBS)

FIGURE 4-4. Three example signature implementations.

quent requests will still be forwarded to the evicting L1 cache, allowing it to perform the signature check needed to preserve correctness.

If the L2 cache replaces transactional data, it loses the corresponding directory information since the external DRAM does not maintain a directory. As a result of the inclusion property, subsequent references to the same data result in an L2 miss. To preserve correctness, the L2 conservatively broadcasts the coherence request to the L1s, allowing them to check their signatures. To avoid multiple broadcasts for the same block, the L2 rebuilds the directory state by recording the L1s' responses. If an L1 NACKs the request due to a conflict, the L2 directory goes to a new state that requires L1 signature checks for all subsequent requests. A block leaves this state when the request finally succeeds.

Signature Design. The signature compactly represents the read- and write-sets of a transaction. A perfect filter, which precisely records the addresses read and written, can be implemented as bit vector with a bit for each block in the address space. However, this is unnecessary and inefficient, as false conflicts represent a performance, rather than a correctness, issue. The key goals for a practical signature mechanism are (1) size, (2) accuracy, and (3) simplicity. We focus on signatures that can be computed from simple binary operations, such as shifting, ORing, and decoding.

Figure 4-4 shows three signature implementations, where an actual signature needs two copies of the illustrated hardware for read- and write-sets, respectively. Part (a) illustrates inserting a block address A into a simple *bit-select (BS)* signature implementation of size $N = 2^n$ bits. The insert merely decodes the n least-significant bits of A's block address and logically ORs the result with the current signature. While not illustrated, a CONFLICT(O, A) operation simply tests the appropriate bit, while a CLEAR(O) zeros the signature. Part (b) illustrates *double-bit-select (DBS)* that decodes two fields, setting both on an INSERT(O, A) and signaling a conflict only when both are set. DBS is similar to Bulk's default signature mechanism, which permutes the address and then decodes two 10 bit fields. Finally, part (c) illustrates *coarse-bit-select (CBS)* that tracks conflicts at a coarser granularity than blocks (e.g., pages). CBS targets large transactions whose read- or write-sets at the block granularity would fill a small signature.

The next section shows that these simple signatures perform well for current transactional workloads. More creative signatures may prove necessary if larger transactions and deep nesting become the norm.

4.6 Evaluation

This section evaluates the LogTM-SE implementation described in Section 4.5. Results show that signature-based transactional memory generally performs comparably to lock-based synchronization, small, simple signature implementations suffice, and cache victimization occurs rarely for most workloads.

4.6.1 Methodology

We evaluate LogTM-SE using full-system simulation, with details in Chapter 3.

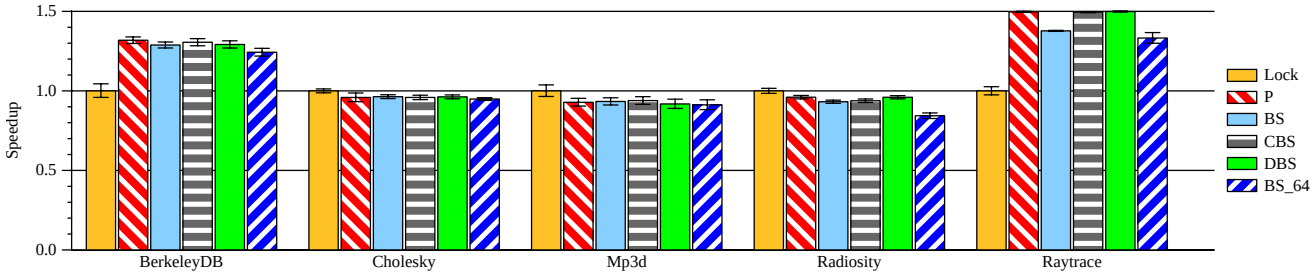


FIGURE 4-5. Speedup normalized to locks.

4.6.2 Workloads

In order to observe a range of program behavior, we converted a variety of multi-threaded workloads to use transactions. These include a database storage library, BerkeleyDB [106], and four SPLASH benchmarks, Cholesky, Radiosity, Raytrace, and Mp3d [124]. Workload descriptions and characteristics are provided in Chapter 3. In each case, we converted the original lock-based multi-threaded program to use transactions in place of lock-protected critical sections. We also execute the original lock-based versions and compare the execution times of the lock-based versions with the transactional versions. Subsequent chapters (Chapters 5-7) of this dissertation focus only on the execution times of the transactional versions of workloads.

4.6.3 Results

Performance with Perfect Signatures. We begin by showing that LogTM-SE with idealized signatures generally performs at least comparably to lock-based programs. We use idealized signatures (which do not incur false positives) to gauge their performance differences with realistic signatures (which can incur false positives). These comparisons are valid because it is useful to compare the performance impact of having infinite versus finite storage for tracking read- and write-sets. Appendix A gives the raw execution time numbers for all configurations (locks and

signatures). For each benchmark, Figure 4-5 presents the execution time speedups for different TM variants relative to the left-most bar which represents the lock-based programs (Lock). The second bar, P, displays the performance of LogTM-SE using perfect signatures—idealized signatures that record exact read- and write-sets, regardless of their size.

Result 1: LogTM-SE with un-implementable perfect signatures performs comparable to locks or better. BerkeleyDB and Raytrace perform 20-50% better using transactions, while the differences for Cholesky, Mp3d, and Radiosity are not statistically significant (note the 95% confidence intervals denoted by the error bars).

Implication 1: LogTM-SE's eager version management and local commit allows programmers to use the easier TM programming model without sacrificing performance, provided that realistic signature implementations do not degrade performance.

Performance with Realistic Signatures. To evaluate realistic signature implementations, Figure 4-5 presents the speedups for LogTM-SE with 2 Kb signatures using bit-select (BS), coarse-bit-select (CBS), and double-bit-select (DBS). BS decodes the least-significant 11 bits of the block address. CBS decodes the least-significant 11 bits of a 1 KB macro-block (sixteen 64-byte blocks). DBS separately decodes the 10 least-significant bits of a block address and the next 10 address bits, setting and checking two signature bits.

Result 2: LogTM-SE with the CBS and DBS signatures performs comparably to LogTM-SE with perfect signatures, while the simplest scheme, BS, degrades performance modestly for Radiosity and Raytrace.

Implication 2: If these results generalize to future TM workloads, LogTM-SE can use simple signatures to approximate perfect signatures and perform well.

Signature Sizing. Smaller signatures reduce implementation cost, but increase the probability of false positives. Given the well-known *birthday paradox* [132,121], one might expect small signatures to perform poorly. The last bar in Figure 4-5 presents the speedup for a 64 bit BS signature (BS_64).

Result 3: The 64 bit BS signature performs comparably to perfect signatures for 3 of the 5 benchmarks, but performs up to 20% slower for Radiosity and Raytrace. Small signatures suffice because most transactions have small read and write sets (Table 3-2 in Chapter 3) and spend most of their time executing non-transactional code (not shown).

Implication 3: These results suggest that small signatures may allow initial HTM implementations to use modest resources until the nature and importance of TM applications becomes clear.

Importance of Victimization. We also studied how often these benchmarks victimize transactional data from L1 or L2 caches. Only Raytrace had more than 20 transactions that evicted transactional data from its caches.

Result 4: Raytrace victimized transactional L1 or L2 blocks 481 times in 48K transactions, while other benchmarks victimized transactional blocks less than 20 times.

Implication 4: If these results generalize to future TM workloads, HTM should handle victimization, but do so with minimal complexity and resources.

TABLE 4-1. Impact of signature size on conflict detection

Benchmark	Perfect P			Size (Bits)	BitSelect BS			CoarseBitSelect CBS			DoubleBitSelect DBS		
	# Transactions	# Aborts	# Stalls		False Positive %	# Abort	# Stalls	False Positive %	# Aborts	# Stalls	False Positive %	# Abort	# Stalls
Raytrace	47,781	20,436	66,833	**64**	49	21,668	614,857	42	24,333	124,248	40	23,948	158,754
				128	48	24,288	426,510	40	24,748	117,519	41	24,489	165,350
				256	46	24,508	222,964	41	24,263	118,501	38	17,218	207,219
				512	43	24,395	120,609	29	20,398	68,859	4.1	20,492	70,166
				1024	32	24,515	104,215	16	20,704	68,699	0.5	20,333	68,984
				2048	36	24,692	116,479	2.7	20,516	68,306	0.0	20,353	66,497
BerkeleyDB	1,120	737	27,470	**64**	72	667	43,954	82	742	43,566	60	634	33,891
				128	68	745	45,655	78	777	39,651	62	661	35,693
				256	71	701	43,660	82	701	37,006	65	707	34,058
				512	63	610	35,594	80	787	42,641	11	747	28,706
				1024	66	724	33,748	79	763	38,443	18	742	28,946
				2048	60	688	30,979	51	688	28,228	19	718	27,851

In More Detail. To gain further insight, Table 4-1 presents additional information on Raytrace and BerkeleyDB. For both benchmarks, Table 4-1 presents the number of transaction commits, transaction stalls (i.e., the number of times transactions have a request NACKed), and transaction aborts for both perfect and practical signatures. It also presents the fraction of conflicts that arise from false positives. For 2 Kb signatures, for example, false positives account for 0-60% of all conflicts. This increases to 40-82% of all conflicts as the signature size shrinks to 64 bits. While false positives increase stalls for both benchmarks, the impact on aborts differs. For BerkeleyDB with all signature schemes and Raytrace with CBS and DBS, the number of aborts is comparable for 2 Kb and perfect signatures. Raytrace with 2 Kb BS signatures incurs roughly 21% more aborts. Furthermore, while reducing the signature size to 64 bits has little discernible effect on BerkeleyDB's abort frequency, it increases the number of aborts for Raytrace by 18% for CBS and DBS, but decreases them by a third for BS. This illustrates a complex interaction: false posi-

tives may lead to false cycles (and thus aborts) or to serializing transactions (and thus no aborts). To see why, consider a single bit signature, which effectively acts as a global lock, eliminating the need to ever abort a transaction.

The large number of stalls relative to aborts indicates that given time, many conflicts will resolve themselves. Thus stalling a transaction may be preferable to aborting it and discarding otherwise useful work. While the stall to abort ratio is highest for small signatures, even with a perfect signature there are more stalls than aborts. BerkeleyDB has many more stalls than transactions, which occurs because a transaction may retry a coherence operation multiple times before the conflict clears and it makes progress.

The false positive rate roughly correlates to the size of transactional read- and write-sets. Table 3-2 shows the average and maximum number of cache lines in each workload's read- and write-sets using perfect signatures. Since read-sets average 2 to 8 blocks and write-sets 1 to 7 blocks, few signature bits are set on average. However, the read- and write-set distribution can be highly skewed, resulting in some transactions that set many signature bits and create many false conflicts. Raytrace's 550-block maximum read-set size represents the worst case, which helps explain why Raytrace's performance falls off with the 64 bit BS signature.

4.7 Alternative LogTM-SE Implementations

The LogTM-SE approach should work well with other shared-memory systems, including a single CMP with snooping coherence and a multiple-CMP system.

A Snooping CMP. Consider a single CMP as described in Section 4.5—per-core writeback L1 caches, multi-banked shared L2 cache, standard off-chip DRAM—but change the MESI coher-

ence protocol to use broadcast snooping. As is common, assume that L1 and L2 banks determine whether a coherence request has an L1 owner (one or more L1 sharers) via a logically-ORed *owner* (*shared*) signal.

Adding LogTM-SE to this snooping system requires the same additions to the core as in Section 4.5, but different coherence changes. With snooping, LogTM-SE requires a third logically-ORed signal, called *nack*, that cores use to NACK coherence requests when their signatures detect a conflict. Because snooping protocols broadcast all coherence requests, they eliminate the need for sticky states or other special mechanisms to reach all necessary signatures. Because directories provide a first-level filter, broadcast snooping systems may need larger signatures to achieve comparable false positive rates.

As a concrete example, we consider the implementation of LogTM-SE using Sun's Gigaplane broadcast bus. Gigaplane is used as the interconnect in Sun Enterprise E6500/E5500/E4500/E3500 SMP servers, as well as the interconnect to connect two to four of these SMP servers to form the multiple SMP Wildfire system [41]. The most important feature LogTM-SE leverages from the Gigaplane implementation is the *ignore* coherence state (and signal). In the Wildfire, the ignore signal is used to remove a node's snoop request from the local snoop order when the local memory tags indicate insufficient coherence permissions to service a node's request. After the request is removed the request is routed to the global coherence mechanisms, which then provides the data and permissions to the requestor. The requestor then retries its request, which will then be satisfied locally. In LogTM-SE, the nack functionality can be implemented by the combination of a shared nack signal and the ignore signal. However the semantics of the ignore signal would be different for LogTM-SE. Any time a processor has a signature conflict with external coherence

requests it signals the nack line. When the requestor notices the nack line is signaled it sets its own ignore signal to remove its conflicting request from the snoop order. In one possible implementation LogTM-SE requires that the nack signal be asserted on a specific cycle (determined by the coherence protocol) after a request has been issued on the bus. If the nack signal is not asserted on that cycle it is safe to assume the request has not been nacked and can be completed. Furthermore if multiple requests are pipelined for concurrency then the Gigaplane system would need to support transaction identifiers for the nack and ignore signals (to uniquely identify each memory transaction from a node). A memory request that is nacked can then be retried just like an ignored request in the Wildfire system.

Multiple CMPs. Consider a system with four CMPs (attached to standard DRAM) interconnected with a reliable point-to-point network. Assume that intra-chip coherence is maintained with the L2 directory of Section 4.5. Assume that inter-chip coherence is maintained with full-map directory protocol requiring a few state bits and 4 sharer bits per memory block. Directory state can be stored in memory bits freed by calculating SECDED ECC on 256 bits rather than the standard 64 bits [84]. For speed, directory state can be cached in a structure beside the home CMP's L2 cache.

LogTM-SE extends this multiple CMP system by adding the on-chip changes of Section 4.5 and altering the inter-chip directory coherence protocol to support NACKs on transaction conflicts and sticky states to handle victimization. An L2 cache that wishes to victimize a transactionally-modified block, for example, does a writeback to the directory at memory, so the directory can store the block and enter "sticky M". While these changes are conceptually straightforward, a full paper may be required to address the details.

4.8 Design Alternatives for Out-of-order Processor Cores

This section discusses the design decisions regarding the LogTM-SE support in the pipeline of our out-of-order (OoO) processor cores. We also discuss alternative designs. Our discussion assumes processors which support sequential consistency (SC), and later describes what is needed in order to support more relaxed memory models.

4.8.1 Choosing When to Update Signatures

We first state the invariant that needs to be maintained in order to achieve eager conflict detection in processors supporting SC, and then describe three solutions for signatures which maintain this invariant for eager conflict detection.

Invariant. For eager conflict detection, conflicts must be detected before a memory instruction retires. In a processor supporting SC, once coherence permissions are obtained and the memory instruction is ready to retire, then that memory instruction is considered conflict-free. For load instructions, once the load instruction is ready to retire and obtains its coherence permissions then there are no conflicts with that load. A store that is the oldest instruction in the reorder buffer can be moved to a separate write buffer to await coherence permissions. However, the store must, in a correct implementation of SC, stall the retirement of the first subsequent load instruction while the store gets the appropriate cache permissions. Once the store gets the coherence permissions it is considered conflict-free, and can unstall any waiting load instructions.

Three solutions. We note that signatures may be updated speculatively (by speculative memory instructions) or non-speculatively (by retiring memory instructions). For the first solution, we choose to update signatures speculatively in the processor. This solution correctly maintains con-

flict detection because all transactional memory requests update the signature, regardless of whether they eventually retire or not. These extra address insertions can only negatively impact performance (due to false positives) but not correctness.

The second solution updates signatures non-speculatively (at instruction retire). In order to detect conflicts with in-flight loads and stores, processors utilize a snooping load store queue (LSQ) that is checked for conflicts in addition to the non-speculative signature. Transactional loads and stores in the LSQ are tagged with an additional bit indicating whether it is a transactional request or not. Any external coherence request that indicates a match with a LSQ entry or the signature will signal a conflict. Conflicts can be resolved in favor of the requestor, by forcing the responding processor to squash dependent instructions after the first instance of the conflict and then retry the conflicting load or store before it is allowed to retire the conflicting instruction. Otherwise conflicts can be resolved in favor of the responder (by NACKing the requestor on conflicts). Because the LSQ tracks in-flight loads and stores (some of which could be speculatively squashed), false positives could arise during conflict detection, which could decrease performance.

The third solution also uses a non-speculative signature. However instead of relying on snooping LSQs, conceptually all load instructions verify that cache permissions are still valid at instruction commit. Conflicts by external stores would cause cache lines to be updated (and cache permissions to be invalidated), and this scenario is detected by comparing a load's value during the execution stage with the value read at instruction retire. This is the load-value replay technique proposed by Cain et al. [19]. In the presence of conflicts, the processor needs to squash

dependent instructions after the conflict and retry the conflicting load request until the load obtains the correct cache permissions to retire.

Extending to relaxed memory models. Under more relaxed memory models (e.g., Total Store Order (TSO)), adhering to the definition of eager conflict detection is more complex than for SC. For example, TSO allows the use of a write buffer, and processors which insert stores in the write buffer can consider the stores to be retired. Furthermore, loads which hit to stores in the write buffer are allowed to use those values in the write buffer, even before those values are seen by other processors. Without modifying the definition of eager conflict detection, a correct (and un-optimized) solution for processors which support TSO would be to never allow loads to bypass from stores in the write buffer before draining all stores, and to not insert stores into the write buffer until the store is the oldest instruction in the reorder buffer. Both of these actions ensure memory is updated in a SC-like fashion. Higher-performance implementations may need to loosen the definition of eager conflict detection in order to deal with non-SC implementations of the memory consistency model supported by the processor.

4.8.2 Concurrent Execution of Top-Level Transactions

In our processor implementation, the transactional commit instruction of the outer-most transaction serializes our OoO processor's pipeline. That is, we do not allow multiple top-level transactions to execute concurrently in the pipeline. The reason for this is because we did not want to implement the extra logic to both store multiple logical signature sets (one for each top-level transaction begin), as well as to provide the lookup logic to check the correct set of signatures. For example, if transaction A is followed by transaction B in the pipeline, transaction B also needs to check A's signatures. Furthermore, all this logic gets even more complicated in order to handle

mis-speculations, exceptions, and transaction aborts. Finally, the version management hardware would need to be augmented to handle multiple top-level transactions in the pipeline. However, HTM designers may choose to implement this hardware support in order to reap the additional performance benefits of supporting multiple concurrent transactions in the processor's pipeline.

4.9 Related Work

HTMs. LogTM-SE builds on the large body of research on HTM systems [7,23,28,30,43,48,80, 93]. LogTM-SE derives most directly from LogTM [80] and Bulk [23].

LogTM-SE improves upon LogTM by removing flash-cleared R and W bits from L1 caches and by improving virtualization. The R and W bits in LogTM do not scale easily with multi-threaded cores (requiring T copies for T hardware thread contexts) or nesting levels (requiring L copies for L levels of nesting support). In addition, LogTM's R and W bits pose a challenge for virtualizing transaction support as R and W bits can not be easily saved or restored. As a result, LogTM-SE supports thread suspension and migration while LogTM does not.

LogTM-SE differs from Bulk by making commit a local operation, supporting non-broadcast coherence protocols and allowing arbitrary signatures. Bulk's commit operation broadcasts the write signature of the committing transaction to all cores and possibly restores victimized transactional data to their original locations in memory. LogTM-SE's commit, by contrast, simply clears the committing transaction's signatures and resets its log pointer. In order to maintain strong atomicity, all Bulk cores must check their read signatures to see if it might contain the address of any non-transactional stores executed by any other core in the system even if that core is not currently caching the block. LogTM-SE, on the other hand, leverages LogTM's sticky states to ensure that coherence requests are sent to all necessary signatures without relying on broadcast. Finally,

because LogTM-SE's version management is independent of caching, it eliminates Bulk's requirement that each signature precisely identify (no false negatives or positives) the cache sets of all addresses it represents (e.g., using 1K bits for a cache with 1K sets).

Virtualization. LogTM-SE, similar to UTM [7], VTM [93], UnrestrictedTM [15], PTM [28] and XTM [30], supports the virtualization of transactions. Compared to these systems, LogTM-SE adds less hardware, uses its virtualization mechanism less frequently, and requires less work to process cache misses and transaction commits after virtualization events.

UTM virtualizes transactions using state (including a pointer) added to each memory block and an additional level of address translation. VTM supports virtualization with a combination of software and firmware, which stores transactional data and read- and write-sets in software tables when transactional data are evicted from the cache or when a transactional thread is suspended. UnrestrictedTM virtualizes transactions by allowing only one unrestricted transaction at a time to execute after cache victimization (but allowing the execution of multiple restricted transactions). XTM and PTM leverage paging and address translation mechanisms to virtualize transactions. Both provide software solutions and propose hardware mechanisms to accelerate common operations (XTM-g and PTM-Select).

TokenTM [16] is a recent HTM that requires little cache coherence modifications (i.e., additional payload on coherence messages), and handles virtualization through a combination of meta-data on each cache block (that moves with the data block at all times) and a more complex software manager to handle transaction commits and aborts. TokenTM has several advantages over LogTM-SE. First, the transactional performance of TokenTM is thread-independent. This means each thread can independently check for conflicts (without the existence of false conflicts)

and trap to a software manager to handle conflicts, without serializing behind other threads. In LogTM-SE, if signatures fill up, all threads degrade to serialization, and this can hurt system performance. Second, other than additional payload on coherence messages TokenTM does not require any additional cache coherence modifications (e.g., NACKs, directory sticky states). This may ease the implementation and verification costs of TokenTM. Third, unlike LogTM-SE, TokenTM does not perform inter-processor interrupts in order to support thread-switches and paging events. Thus, TokenTM handles these virtualization events with lower overheads than LogTM-SE.

TokenTM also has several disadvantages compared with LogTM-SE. First, its implementation requires more hardware state than LogTM-SE. This hardware state is mainly due to the extra meta-data associated with each cache block in main memory (e.g., 16 bits for every 64 bytes). In contrast, LogTM-SE does not add meta-data to the memory subsystem. Second, in order to support fast token release (an optimized form of transaction commit for transactions that don't overflow the L1 cache), TokenTM implements flash-clear and flash-OR circuits in the L1 cache designs of each processor core. This support may be too burdensome for optimized L1 cache designs. Third, TokenTM's cache coherence protocol uses non-silent evictions for correctness (in order to precisely track the number of transactional readers). In contrast, LogTM-SE supports silent-S evictions in its cache coherence protocol.

TABLE 4-2. Comparison of HTM virtualization techniques

	Before Virtualization				After Virtualization						Legend
	$Miss	Commit	Abort	$Eviction	$Miss	Commit	Abort	$Eviction	Paging	Thread Switch	
UTM [7]	-	-	-	H	H	H	HC	H	H	H	Shaded = virtualization event - = handled in simple hardware H = complex hardware S = handled in software A = abort transaction C = copy values W = walk cache V = validate read set B = block other transactions
VTM [93]	-	-	-	S	S	SC	S	S	S	SWV	
UnrestrictedTM[15]	-	-	-	A	B	B	B	B	AS	AS	
XTM [30]	-	-	-	ASC	-	SCV	S	SC	SC	AS	
XTM-g [30]	-	-	-	SC	-	SCV	S	SC	SC	AS	
PTM-Copy [28]	-	-	-	SC	S	S	SC	SC	S	S	
PTM-Select [28]	-	-	-	S	H	S	S	S	S	S	
TokenTM [16]	-	-	SC	-	-	S	SC	-	-	-	
LogTM-SE	-	-	SC	-	-	S	SC	-	S	S	

Table 4-2 presents a rough comparison of the different systems' efficiencies by displaying the actions they take on various system and cache events. As indicated by the "Before Virtualization" columns (left), all of the previous systems handle the common case of non-virtualized small transactions using simple hardware mechanisms. All these systems have a conceptual virtualization mode, which they switch to after evicting transactional data from the cache, or a paging operation or context switch during a transaction. As indicated by the "After virtualization" columns, most of these systems either restrict concurrency or require complex hardware or slow software for at least one common case operation. UnrestrictedTM blocks all other transactions until the virtualized transaction commits. VTM and PTM-Copy require slow software-based conflict detection on cache misses. UTM and PTM-Select perform similarly complex operations in hardware on cache misses. XTM and XTM-g require expensive page-based validation of transactions' read-sets at commit.

Like these systems, LogTM-SE requires little hardware overhead to support virtualization—one summary signature per thread context. In LogTM-SE, however, virtualization does not force the use of software for conflict detection, nor restrict the concurrency of transactions. LogTM-SE requires the least effort and expense to handle cache misses and commits—the most frequent events—after virtualization. Most importantly, in LogTM-SE, cache victimization of transactional data does not require virtualization.

TokenTM handles paging and thread switches through the use of additional hardware state (additional bits and flash-OR and flash-clear circuits for each L1 cache block). In contrast, LogTM-SE uses operating system support to implement this functionality (e.g., computing and updating summary signatures).

Hybrid transactional memory systems [33, 62] provide virtualization by integrating an HTM with an STM. Small transactions, in the absence of virtualization events, execute as hardware transactions, while transactions that require virtualization execute as software transactions. HyTM [33] requires the least amount of hardware support of any of the virtualization schemes (it can run purely in software). However, hybrid schemes add overhead to hardware transactions in order to detect conflicts with concurrent software transactions. Initial results with HyTM indicate that a virtualized HTM will perform better in the presence of cache victimization [33].

4.10 Conclusions and Future Work

This chapter proposes a hardware transactional memory (HTM) system called *LogTM Signature Edition (LogTM-SE)* that combines features of prior HTM systems—especially *LogTM, Nested LogTM,* and *Bulk*. LogTM-SE stores principal transactional state in two structure types—*signature* and *log*—to achieve two key benefits. First, signatures and logs can be implemented

without changes to highly-optimized cache arrays. Leaving critical cache arrays untouched may facilitate HTM adoption by reducing risk. Second, signatures and logs are software accessible to allow OS and runtime software to manipulate them for virtualization. With little extra hardware, LogTM-SE handles cache victimization, unbounded nesting (both open and closed), thread context switching and migration, and paging.

In the future, researchers should examine the design of the summary signatures in more detail. Furthermore, researchers should expand the set of workloads that contain interesting virtualization events. The workloads we use in the evaluation does not utilize the summary signatures. The TM community would benefit from exploring additional workloads with virtualization activity and examining their impact on summary signatures.

Chapter 5

Notary: Hardware Techniques to Enhance Signatures

Hardware signatures have been recently proposed as an efficient mechanism to detect conflicts amongst concurrently running transactions in transactional memory systems (e.g., Bulk, LogTM-SE, and SigTM). Signatures use fixed hardware to represent an unbounded number of addresses, but may lead to false conflicts (detecting a conflict when none exists). Previous work recommends that signatures be implemented with parallel Bloom filters with two or four hash functions (e.g., H_3).

Two problems exist with current signature designs. First, H_3 implementations use many XOR gates. This increases hardware area and power overheads. Second, signature false positives can result from conflicts with signature bits set by private memory addresses that do not require isolation.

This chapter develops Notary, a coupling of two signature enhancements to ameliorate these problems. First, we use address entropy analysis to develop Page-Block-XOR (PBX) hashing and show it performs similar to H_3 at lower hardware cost. Second, we introduce a privatization interface that explicitly allows the programmer to declare shared and private heap memory allocation. Spear et al. [109] previously defined privatization as the act of segregating shared and private data. These data categories are treated differently by conflict detection mechanisms. It is a global operation, meaning all threads agree not to check for conflicts on private data. Privatization

reduces false conflicts arising from private memory accesses and can lead to a reduction in the signature size used.

Results from transistor-level layouts of H_3 and PBX, along with full-system simulation of a 16-core chip-multiprocessor implementing LogTM-SE, show (a) PBX hashing performs similar to H_3 hashing while requiring up to 21% less area and 4.7% less power overhead and (b) privatization can improve execution time by up to 86% (by reducing false conflicts by up to 96%).

5.1 Introduction

High-performance transactional memory (TM) systems execute multiple transactions concurrently and simultaneously commit only those that do not conflict. A *conflict* occurs when two or more concurrent transactions perform an access to the same memory address and at least one of the accesses is a write. In order to detect conflicts, the TM system must record the set of addresses that a transaction reads (the *read-set*) and writes (the *write-set*). Recently, *signatures* have been proposed as a hardware mechanism to track read- and write-sets in a compact manner. Signatures were first proposed for Bulk [23]. Several recent systems have also adopted its use, including BulkSC [24], LogTM-SE [127], and SigTM [77]. These systems implement signatures with per-thread hardware Bloom filters [12]. Read- and write-sets are tracked by inserting the addresses of transactional loads and stores into the read and write signatures, respectively. Address lookups are implemented either as a test operation (LogTM-SE, SigTM) or by intersecting two signatures (Bulk, BulkSC). A test or intersection operation may signal a conflict when none existed (a *false positive*), but never misses a conflict (no *false negatives*). Finally, signatures are cleared on commit and abort.

False conflicts arising from false positives determine the effectiveness of signatures as a mechanism for conflict detection. As Section 5.6 shows, small signatures can increase execution time by up to seven times. Increasing signature size usually reduces false conflict rates (much as increasing cache size reduces miss rates), but at some point this solution becomes prohibitively expensive. For example, when using 64kb signatures, there is a 1307% increase in area and a 106% increase in power over 1kb signatures (details in Section 5.3.6). Moreover, larger signatures increase the overheads when signatures move (e.g., Bulk's signature broadcasts [23] and LogTM-SE's saving/restoring signatures [127]). Finally, depending on the implementation, smaller signatures may result in lower access latencies than larger signatures. For these reasons, this chapter will focus on effectively using signature hardware once its size is fixed.

First, we review signature background in Section 5.2. In particular, Sanchez et al. [97] advocate that signature implementations should use parallel Bloom filters (not true Bloom filters) with two or four hash functions (e.g., from the H_3 family [21,94]).

Second, in Section 5.3 we develop the ***Page-Block-XOR (PBX)*** hash function that we will show performs similar to the H_3 hash functions at much less hardware cost. Specifically, we examine address bit entropy [102]. We find that using a single-level of XOR gates operating on non-overlapping fields of an address' page and cache-index bit-fields works well, and name this PBX. In particular, we find that PBX improves on H_3 hashing by using up to 21% less area and 4.7% less power.

Third, in Section 5.4 we develop a ***privatization interface*** that reduces false conflicts by giving programmers the ability to inform the TM system of addresses that *cannot* cause transaction conflicts. The TM system uses this information to keep these addresses out of signatures. Thereby

fewer signature bits are set that might result in false conflicts. We show that privatizing the stack has a negligible effect, so we instead concentrate on heap data by supporting private and shared memory allocators that seek to put private and shared data on different pages.

Finally, in Section 5.5 we present our evaluation methodology, and in Section 5.6 we evaluate the combination of our ideas that we call ***Notary***. Notary is a coupling of enhancements to signatures that consists of Page-Block-XOR hashing and a privatization interface. Results for a 16-core CMP implementing LogTM-SE [127] show (a) PBX hashing performs similar to H_3 hashing (and requires less hardware) and (b) privatization can improve execution time by up to 86% (by reducing false conflicts by up to 96%). In Section 5.6.3 we discuss how Notary can be implemented in any signature-based TM system and in signature-based non-TM systems.

5.2 Signature Background

5.2.1 Prior Signature Systems

Signatures are a hardware mechanism to concisely represent a set of addresses without the possibility of false negatives. They are usually implemented as Bloom filters. Signatures for TM were first proposed by Ceze et al. [23] for use in Bulk, which broadcasts signatures in order to support both a hardware TM (HTM) system and a speculative multi-threading system. Other HTM systems have also used signatures to detect conflicts amongst concurrent transactions.

Signatures are also used for purposes besides TM and speculative multi-threading. Examples include BulkSC [24], which uses hardware-defined (implicit) transactions in order to track violations of the sequential consistency memory model. The SoftSig framework [113] defines a general software programming interface for signatures. Atom-Aid [67] uses signatures to detect

potential atomicity violations amongst implicit transactions. Finally, signatures can be used in the design of race-recording mechanisms for multi-threaded programs, and examples include DeLorean [78] and Rerun [50].

5.2.2 Prior Signature Results

Although numerous papers have proposed systems that use signatures, Sanchez et al. [97] focus exclusively on important implementation and performance issues for TM systems. They present four conclusions stemming from their analysis of signature design and performance: (1) signature false positives are detrimental to performance (e.g., up to 50% performance degradation with 512b signatures), (2) parallel Bloom signatures should be used for implementation efficiency, (3) H_3 hashing should be used in favor of the lower-quality bit-selection hash function, and (4) signatures should use two or more hash functions. This chapter's focus is on reducing signature implementation overheads and on a privatization interface for signatures.

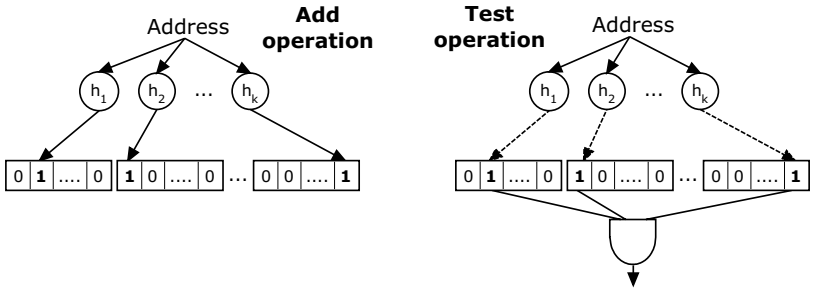


FIGURE 5-1. Parallel Bloom signature design.

Parallel Bloom signatures partition a single, logical signature, and each partition is accessed independently by a hash function. Figure 5-1 illustrates this signature and its two operations (add and test) for *k* hash functions. During an add operation, each of the *k* hash functions hashes the input address and sets one bit in its designated signature partition. A test operation indicates the presence of an address only if all individual hash functions indicate its partition's bit is set during lookup.

5.2.3 H_3 Hash Functions

The H_3 class of hash functions creates uniformly distributed hash values [21,94]. For a straightforward implementation of H_3 using fixed Boolean matrices to select address bits, on average half of the input address bits can affect a given bit of a hash value. Figure 5-2 illustrates this for 32-bit addresses, in which sixteen address bits produce each bit of a *c*-bit hash value. While flexible, this implementation uses a large number of XOR gates, even after optimizations

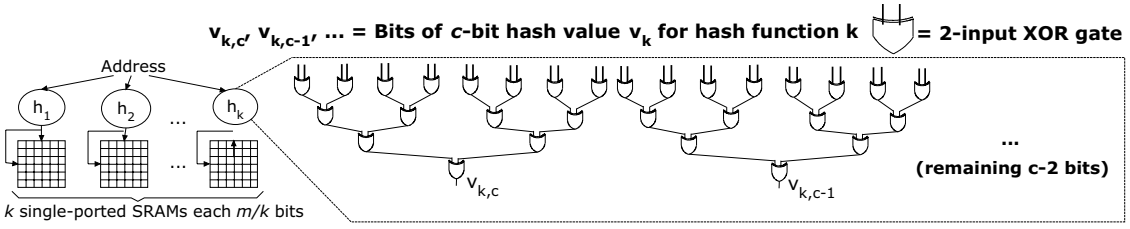


FIGURE 5-2. Bloom signature implementation with *k* H_3 hash functions, *c*-bit hash values, and a *m*-bit signature.

[115]. In general, given fixed Boolean matrices, the number of 2-input XOR gates required to implement k hash functions producing c-bit hash values is:

$$\begin{aligned}&\text{Num XOR}\\&= \sum_{j=1}^{k} \sum_{i=1}^{c} \left\lceil \frac{\text{Number of 1s in column } i \text{ of matrix } j}{2} \right\rceil\\&\approx \frac{\text{address length in bits}}{4} \times c \times k\end{aligned}$$

For example, assume 2kb signatures with two hash functions (k=2) are used. Each hash generates a 10-bit hash value (c=10). If sixteen bits from a 32-bit address (e.g., sixteen ones in each column of the Boolean matrix) generate a bit of a hash value, then a total of 160 XOR gates per signature will be required. The overhead increases if processor cores support multiple thread contexts and each context uses separate signatures. Similar overheads can be calculated using XOR gates with higher fan-in (e.g., 4-input XORs). We present quantitative analysis of the additional area and power overheads of H_3 hash functions in Section 5.3.6.

5.2.4 XOR Hashing

Our proposed PBX hashing, detailed in Section 5.3, is an XOR-based hashing function. This section describes prior uses of and alternatives to XOR hashing. In memory hierarchy designs, hash functions are primarily used to reduce cache, bank, or DRAM row-buffer conflicts. In particular, both XOR-based hash functions [40,101,130] and polynomial hash functions [95] have been proposed. Both are successful at reducing memory storage conflicts.

Kharbutli et al. [59,60] propose two alternative hash functions to XOR that reduce the probability of bad hash values: prime modulo hashing and odd-multiplier displacement hashing. While

efficient implementations exist (e.g., using adders and selectors for prime modulo indexing and adders for odd-multiplier displacement), it can require modifications to existing hardware structures (e.g., additional bits in each hardware TLB entry) or additional hardware (adders). In contrast, PBX uses only a small number of XOR gates and no modifications to hardware (e.g., to L1 cache, TLB).

Vandierendonck et al. [115] provide a detailed examination of XOR hashing by using the concepts of null and column spaces from linear algebra. They show that elementary matrix operations (replacing and swapping columns) can minimize the fan-in and maximum fan-out of XOR gates, and also discuss polynomial hash functions (both reducible and irreducible [95]). Both H_3 and PBX benefit from their results, by reducing the fan-in and fan-out of the XOR gates used in implementations.

5.3 PBX: Using Entropy to Reduce H_3 Costs

5.3.1 Motivation

H_3 hashing requires many XOR gates, which increases the area, power, and latency of signature implementations. Since most programs do not use the entire virtual address space, some address bits exhibit more randomness than other bits. This suggests that it may be unnecessary for H_3 to use multi-level XOR trees to produce random hash values, as the input bits already have randomness. The insight behind PBX is that **if the input bits have sufficient randomness and those bits are used as inputs to the hash functions, then random hash values result.** PBX combines a simple yet powerful technique of selecting the input bits to the hash functions with a much simpler hash function. This reduces the overall area and power overheads compared with

H_3. Moreover, simulations show that the randomness in our workloads' addresses is far from the maximum, and the most random bits are localized to less significant bit-fields in the address.

5.3.2 Entropy

We calculate the randomness of addresses using *entropy* [102], an important measure of randomness. Entropy is a measure of the uncertainty of a random variable *x*, and is calculated as:

$$Entropy = -\sum_{i=1}^{N} p(x_i)\log_2 p(x_i)$$

where $p(x_i)$ is the probability of the occurrence of value x_i, *N* is the number of sample values the random variable *x* can take on, and *Entropy* is the amount of information required on average to describe an outcome of the random variable *x*. The units are bits. For example, if every bit pattern in a *n*-bit field occurs equally often, entropy is *n* bits. At the other extreme, if only a single pattern is possible, the entropy is zero bits. All other distributions have entropy greater than zero and less than *n* bits.

To estimate the entropy of a workload's addresses, we use the following definitions in our calculations. Let $\chi = \left\{0, 1, 2, \ldots, 2^n - 1\right\}$, where *n* is the length of the address field. Our data

consists of d addresses $\{a_j\}_{j=1}^{d}$, where $a_j \in \chi$ for all $j \in \{1, \ldots, d\}$. We use the empirical entropy of the data sequence as our estimate of entropy, which we calculate as:

$$Entropy = -\sum_{i=1}^{2^n} \hat{p}(x_i)\log_2 \hat{p}(x_i)$$

$$\hat{p}(x_i) = \frac{1}{d}\sum_{j=1}^{d} I(a_j = x_i)$$

where $I(a_j = x_i) = 1$ if $(a_j = x_i)$, 0 otherwise and

$$0\log 0 = 0$$

We choose to calculate entropy over the data addresses of an entire workload's execution (with some filtering, described below). Although representative, this might hide entropy changes resulting from program phase changes. Nevertheless, we find our choice yields good overall entropy trends that PBX exploits. Our address filtering removes addresses that do not operate on signatures. For example, we ignore supervisor-level addresses, as these are never added to signatures and bypass signature tests. We calculate the entropy of load and store addresses separately, and differentiate between entropies of virtual and physical addresses. We show entropy results for physical memory accesses inside transactions. Similar trends exist for the virtual addresses.

Our entropy calculations assume 32-bit virtual and physical addresses, a cache-block size of 64B, and a page size of 4kB. We assume addresses are little-endian, such that the least significant bit is bit 0. In our entropy results, we represent the ending and starting bit indices of a bit-field of the memory address as [end:start]. The bit-field representing the entire input address is [31:0]. Using 64-bit virtual and physical addresses would not significantly affect our entropy results, since our workloads have working sets that do not exceed 32-bit addresses. We discuss entropy results of larger workloads in Section 5.3.5. Finally, increasing the interleaving of a program's

address space in physical memory would also likely increase the entropy of the higher-order physical address bits.

We calculate two types of entropy for each workload, one on the entire input address (*global* entropy), and the other on bit-fields of the input address (*local* entropy). For the parameters given above this means the maximum global entropy value for any of our workloads is 26 bits (e.g., the address bits remaining after masking the block offset[1]). These two types of entropy give the upper bound of a workload's address entropy and local entropy trends within sub-fields of the input address (e.g., among the lower-order or the higher-order bits).

5.3.2.1 Previous Uses of Entropy

Entropy has previously been used as a measure of randomness in computer architecture research. For example, Hammerstrom et al. [42] use entropy to measure the overheads of addressing memory in an instruction-set architecture. Park et al. [86] describe how address bits can be transferred with smaller bit overheads through the use of a Base Register Cache. This mechanism works because higher-order bits have lower entropy and many programs exhibit spatial locality. Becker et al. [11] extend this work by using Dynamic Huffman Coding to further improve bit compression. Citron et al. [31] describe a mechanism which compacts and expands both addresses and data values over a shared bus. Ballapuram et al. [10] propose low-power TLB designs through the calculation of the entropy of virtual page numbers of heap, global, and stack addresses. We differ from their work in the following ways. First, we examine entropy trends within an address. Our focus is on the average entropies across the entire workload, rather than on

1. Specific to our HTM implementation. Other TMs may choose different address masking granularities.

changes in entropy values over time. Second, we believe we are the first to incorporate the idea of entropy in the selection of the input bits for XOR hashing.

5.3.3 Results

In this section we present entropy results gathered from a suite of fifteen transactional workloads running on top of a simulated 16-core CMP, with details in Section 5.5.

Entropy results for our workloads are shown in Figure 5-3. We first describe the results for the load and store global entropies ("LD/ST Global" lines). First, we note that our workloads' global entropy values are much lower than the maximum value of 26 bits. For our workloads the global entropy values are between 1-16 bits. Second, with the exceptions of Raytrace, BIND, and Prefetch, the store global entropy value is larger than the load global entropy value. In Raytrace, BIND, and Prefetch, the set of transactional store addresses that is frequently accessed is small.

Local entropy is calculated by using a bit-window that moves from lower- to higher-order address bits. This bit-window is 16 bits wide, since global entropy values never exceed 16 bits, and a bit-window of this size allows local entropy values to be close to or equal global entropy values. Larger window sizes (> 16 bits) might miss fine-grain trends in local entropy (e.g., which bits lead to changes in local entropy). Smaller window sizes (< 16 bits) might miss the set of bits which reach the maximum global entropy value. An alternative implementation of local entropy may choose to measure the entropy of each address bit separately. This implementation may yield additional information that is lost by calculating entropy on contiguous address bits. However, we choose to measure the entropy of contiguous address bits because many programs exhibit spatial locality. This property affects the entropy of contiguous address bits, and we want to clearly illustrate this phenomenon for our workloads.

We vary the starting bit index of this window to capture local entropy changes as the window moves towards higher-order bits. This is done by varying N_{skip}, a variable representing the number of lower-order bits skipped before starting the bit-field. Using our notation, this bit-field is $[(N_{skip}+21):(N_{skip}+6)]$, where N_{skip} varies from 0-10.

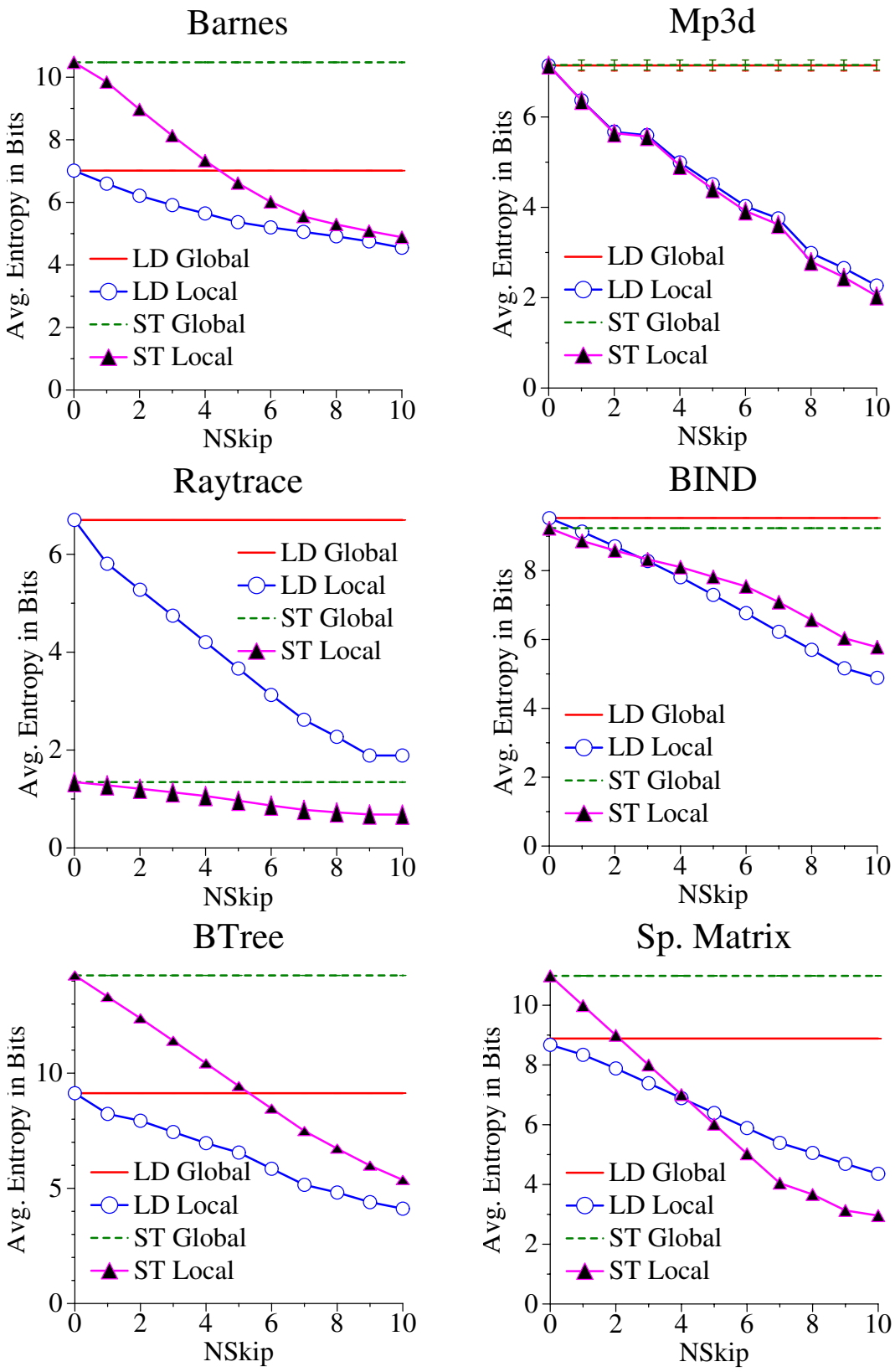


FIGURE 5-3. Entropy of transactional load and store physical addresses.

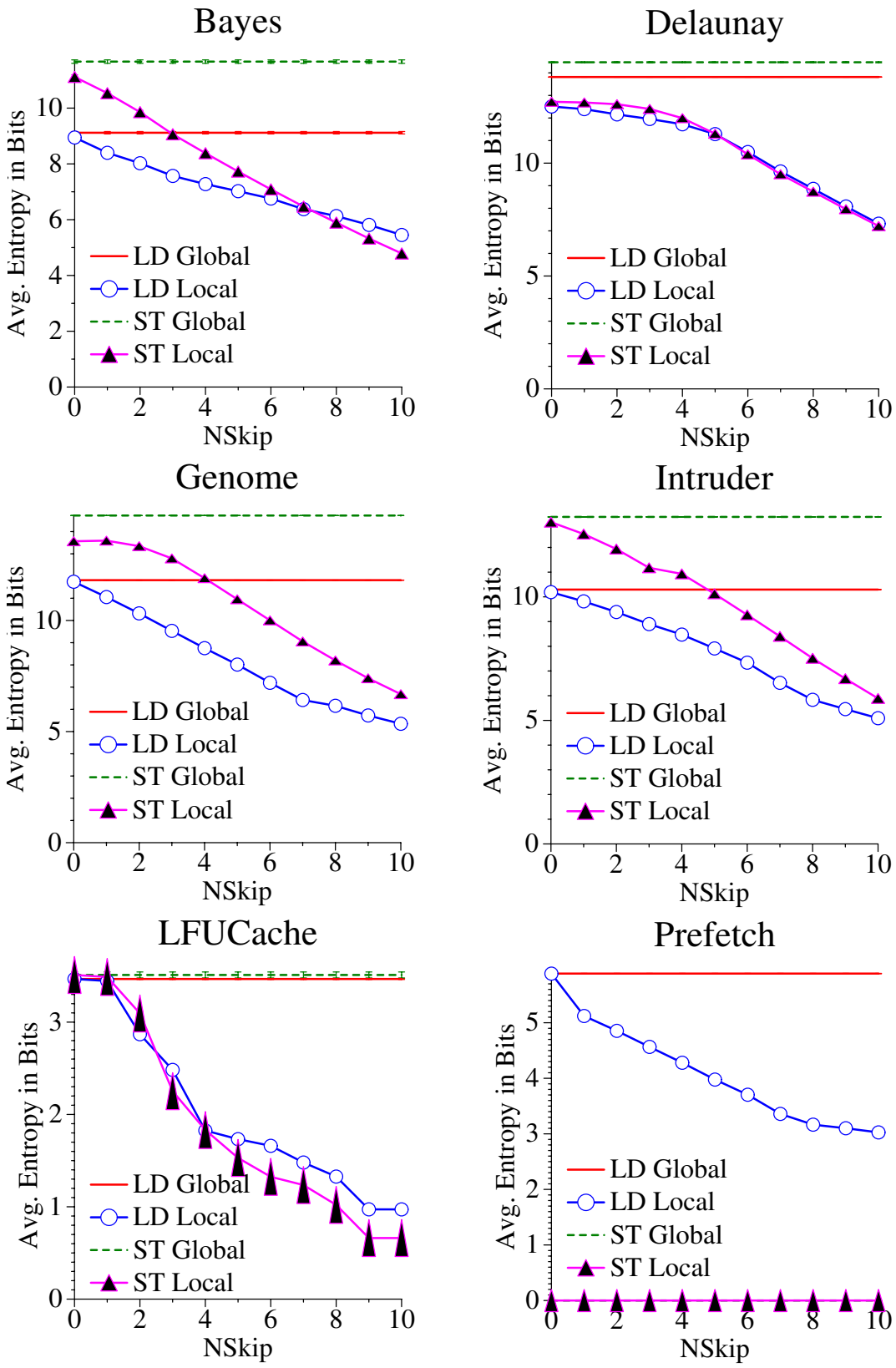


FIGURE 5-3. Entropy of transactional load and store physical addresses.

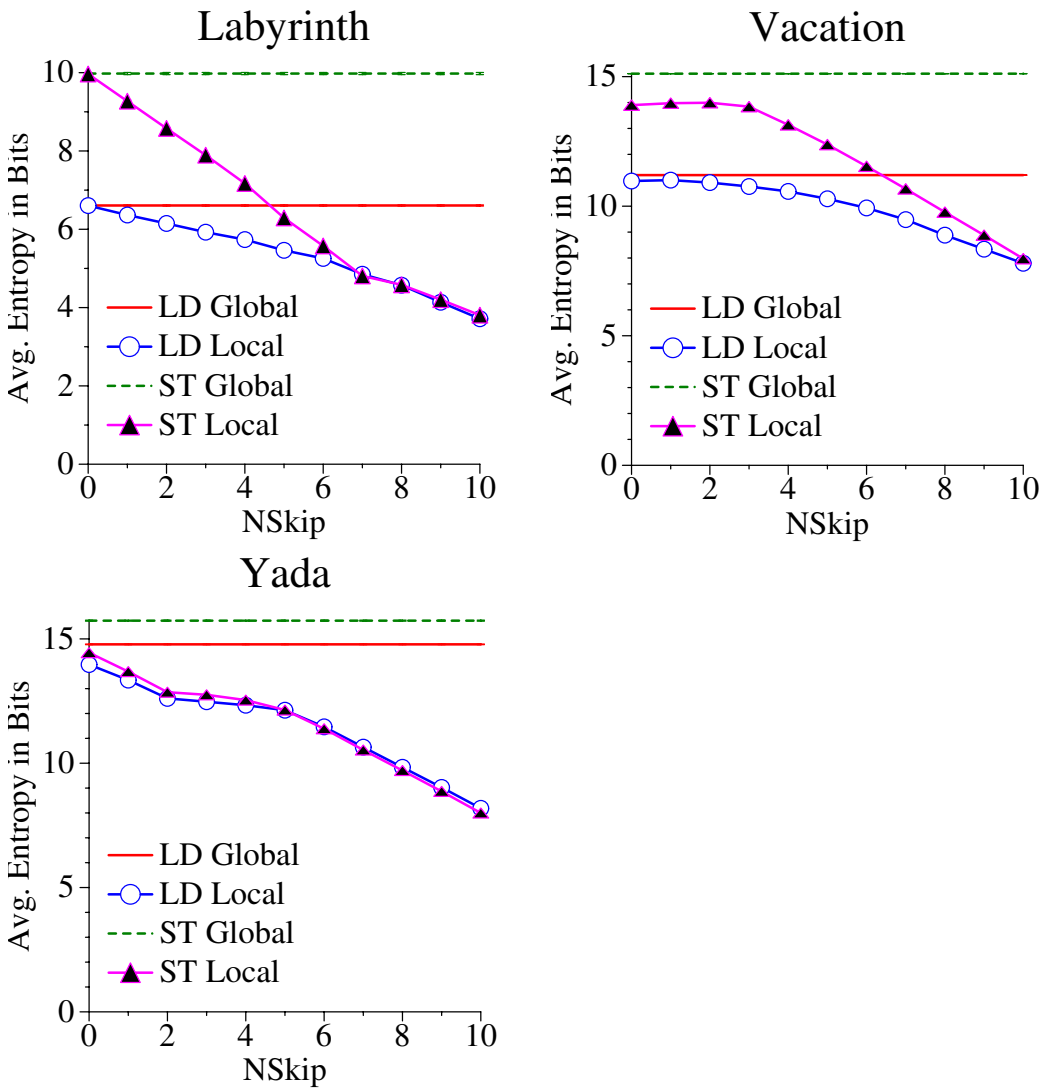


FIGURE 5-3. Entropy of transactional load and store physical addresses.

The results for local load and store entropies are the lines labeled “LD/ST Local” in Figure 5-3. To help understand these results, we focus on BTree’s results (third row, left plot). For load addresses, as the bit-window moves towards higher-order bits the local entropy decreases from an average of about 9 bits to 4 bits. Similarly, for store addresses entropy decreases from about 14 bits to 5.5 bits. Note that Prefetch has no entropy in the store addresses. For the majority of the workloads the global entropy values can be captured by the least significant 16 bits of the block address. For the remaining workloads this is not true, and the local entropy lines do not intersect with the global entropy lines (in Genome, Vacation, Delaunay, Bayes, and Yada). This is because not all of the global entropy value can be captured by a 16-bit window. The remaining can be captured if the window is increased to 26 bits.

The entropy trends all result in smooth, monotonically decreasing curves. This occurs for two reasons. First, our transactional workloads have small working set sizes, which limits the number of pages accessed and reduces local entropy for higher-order bits. Second, our entropy results are gathered on a newly booted simulated machine with little virtual memory activity. In systems with more processes and virtual memory activity the higher-order physical address bits are likely to exhibit more entropy. In the extreme case, if the system has random physical page frame assignments, then all address bits are equally likely to be random across a workload’s execution. If this is the case it may be prudent to examine the entropy of individual address bits rather than the entropy of a contiguous set of address bits. However, programs may exhibit spatial locality within certain phases, and thus our analysis of local entropy within contiguous address bits is still valid.

In summary, the key finding of these results is that for our workloads, local entropies monotonically decrease as we examine higher-order bits.

5.3.4 Exploiting Entropy

We propose PBX (Page-Block-XOR) hashing to exploit entropy characteristics. PBX is motivated by three findings: (1) lower-order bits have the most entropy, and therefore are the best candidates for hashing, (2) XORing two bit-fields produces random hash values, and (3) bit-field overlaps lead to higher false positives due to correlation between the bit-fields. Our discussion will reference Figure 5-3 and Figure 5-4, and PBX signatures using two hash functions. The pairs of numbers in parentheses in Figure 5-4 indicate the bits XORed together to produce each hash bit.

We now discuss these three findings. First, our local entropy results indicate lower-order bits are more random (higher entropy) than higher-order bits. This is intuitive because programs tend to exhibit spatial locality. According to Figure 5-3, the highest local entropy comes from the bit-window with zero N_{skip} bits. We found through further analysis that for our workloads address bits [26:6] are the most random (and bits [31:27] are mainly constant). If systems perform random assignment of physical pages to processes (due to OS paging algorithms or due to paging activity), there is a higher probability that all address bits have similar entropy. Thus it is less important to perform detailed entropy analysis to discover which bits exhibit the most randomness. If operating systems perform page-coloring to reduce conflict misses in caches, then there is correlation in the physical page numbers. Thus, the bits corresponding to the physical page numbers would exhibit less randomness, and bit entropy would be needed to uncover which bits exhibit the most randomness.

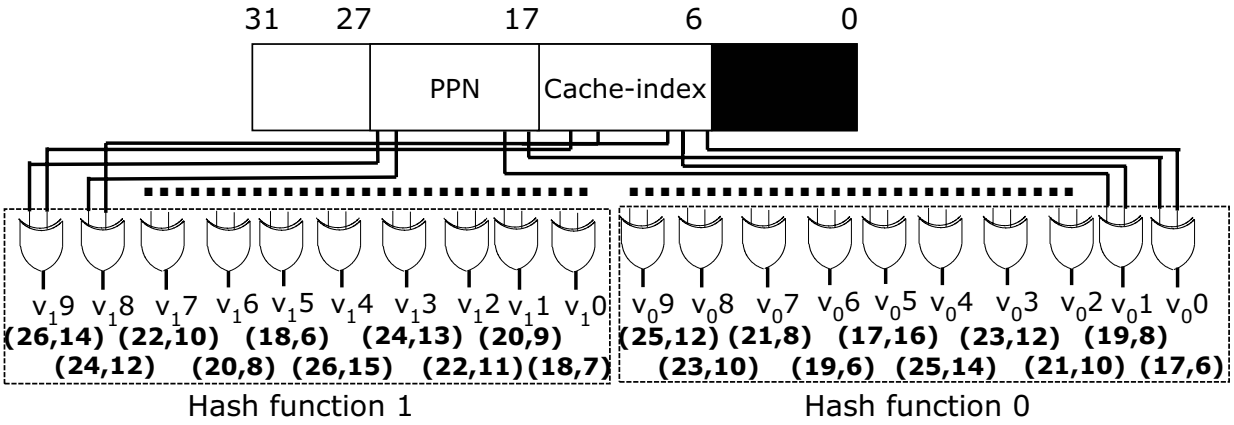


FIGURE 5-4. Bit-fields used for PBX.

Second, previous research has shown that XOR hashing of bit-fields produce random hash values [40,95,101,130]. We, like others, find that XORing two bit-fields produces random hash values. To achieve this we partition the bit-field [26:6] into two bit-fields, one of length 11 bits (bit-field [16:6]), and the other of length 10 bits (bit-field [26:17]). Other partitions are possible, but may not produce all possible hash values (e.g., at the extreme the bit-field [6:6] XORed with another bit-field produces at most two different hash values). Since bits [16:6] are traditionally known as the cache-index bits in caching literature we call this bit-field the cache-index bits. However the size of this bit-field is not tied to any specific cache configuration in our CMP. Similarly, we call bit-field [26:17] the physical page number (PPN) since these bits are traditionally part of the PPN derived during address translation. However like the cache-index bit-field this is not tied to any specific parameter (e.g., the page size) in our CMP. It is important to use most or all of the bits in the bit-fields as inputs to our PBX hashes, as more bits lead to higher probability that all possible hash values will be produced. We verified that our PBX implementation produces random hash values by analyzing the PBX hash matrices used to select the hash bits (implied by the connections in Figure 5-4). We confirmed they can produce all possible hash values.

Third, our cache-index and PPN bit-fields do not overlap. This is beneficial because overlaps can cause correlation in the resultant hash values and reduce the range of hash values produced. Our sensitivity analysis (Section 5.7.1) confirms that correlation can be detrimental to the performance of PBX.

In summary, as shown in Figure 5-4, our PBX signature configurations use two non-overlapping, consecutive bit-fields corresponding to the cache-index and PPN bit-fields. Since signatures use two hash functions, we distribute even-numbered bits in the bit-fields to one hash function, and the odd-numbered bits to the other hash function. Finally, in systems with random physical page assignment, all address bits may likely have equal randomness across a workload's execution. This makes it easier for PBX, since it may be sufficient for PBX to just select two random sets of bits to XOR. However, a program may exhibit spatial locality within specific phases, so the process of selecting the most random bits for PBX may be similar to what we have described above.

5.3.5 Generalizing to Other Workloads

The methodology presented in this section to obtain random address bit-fields applies to other workloads (with bigger working sets) and to larger addresses (e.g., 64-bits). We performed the same analysis as in Figure 5-3 using Wisconsin's Commercial Workload suite (Apache, Zeus, OLTP, and JBB) [3]. Figure 5-5 illustrates our entropy results. We found load global entropy values range from 13-15 bits and store global entropy values range from 13-17 bits. At N_{skip}=10, load local entropy values range from 9-11 bits, and store local entropy values range from 7-13 bits. The higher-order bits exhibit more randomness than in our transactional workloads. Thus the sizes of the cache-index and PPN bit-fields may be increased to capture these bits.

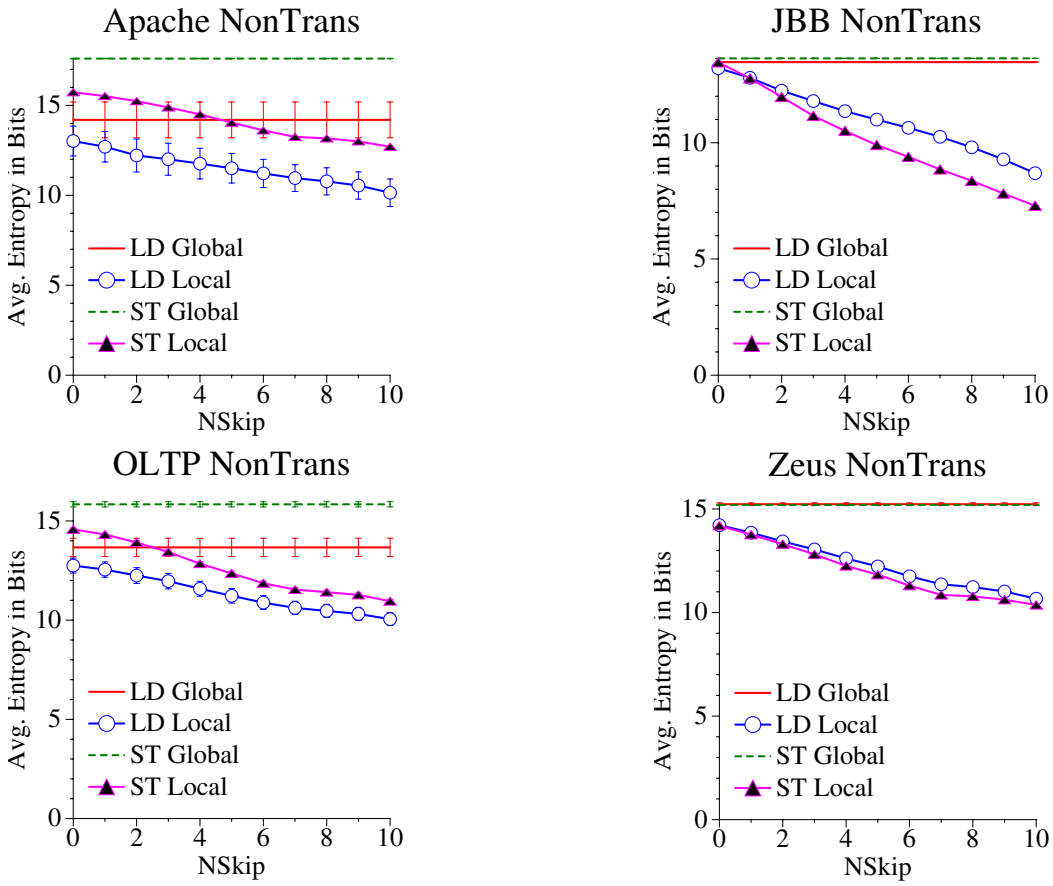


FIGURE 5-5. Entropy of Wisconsin's commercial workloads.

Finally, PBX's bit-field selection can be part of a feedback mechanism for its implementation. Initial runs of a newly-developed TM program can use pre-defined default bit-fields for the PBX signature. As entropy information is gathered during subsequent executions, the programmer or runtime system can more carefully tune the selection of bit-fields to enable better signature performance.

5.3.6 Savings in Area and Power Overhead

PBX translates to lower area and power overheads than H_3. To quantify this, we implement custom transistor-level layouts of different PBX and H_3 hash functions using Cadence's Virtuoso design tool (version 5.10.41). Our hash designs use the optimized transistor-level XOR proposed by Wang et al. [119]. The designs incorporate PMOS transistors having a width of 1200nm and length of 400nm and NMOS transistors having width 600nm and length 400nm. We use linear scaling to scale the area estimates from 400nm to 65nm technology, by dividing area estimates at 400nm by a constant factor of 36 to get the area estimates for 65nm technology. Because many variables are involved in scaling power with technology, we do not scale any of our power numbers from our base 400nm technology. We expect the relative power results to hold if the other variables affecting power also scale with smaller technology nodes. The area and power of the Bloom filters are estimated using CACTI 4.2 [110], and we use 65nm technology for area results. The Bloom filters are implemented as SRAM banks with one read and one write port, and a word size of eight bytes. Due to limitations of CACTI, we only collect results for signature configurations 1kb and larger.

TABLE 5-1. Area overheads (in mm^2) of H_3 and PBX sig.

Sig. config.	Bloom filter	H_3 hash	PBX hash	H_3 sig.	PBX sig.	% savings for PBX sig.
2kb, k=2	3.00e-3	7.83e-4	4.67e-5	3.78e-3	3.05e-3	19
2kb, k=4	4.67e-3	1.35e-3	7.83e-5	6.02e-3	4.75e-3	21

TABLE 5-2. Power overheads (in mW) of H_3 and PBX sig.

Sig. config.	Bloom filter	H_3 hash	PBX hash	H_3 sig.	PBX sig.	% savings for PBX sig.
2kb, k=2	1.10e2	5.80	5.68e-1	1.16e2	1.11e2	4.3
2kb, k=4	1.80e2	1.04e1	1.02	1.90e2	1.81e2	4.7

5.3.6.2 Overheads of PBX Compared to H_3

The area and power results of PBX and H_3 for 2kb signatures with two and four hash functions are presented in Table 5-1 and Table 5-2, respectively. The area estimates of the H_3 and PBX hashes in Table 5-1 are linearly scaled from our transistor-level design implemented at 400nm technology down to 65nm technology (i.e., a factor of about 36). We use CACTI to obtain the Bloom filter area results for 65nm technology. The power overheads of the H_3 and PBX signatures in Table 5-2 are calculated using the base 400nm technology (both in our transistor-level design and in CACTI). Our power estimates for the hashes are derived using inputs which triggers all hash gates to switch their outputs from a previous value. We defer comparing the signature overheads with existing core designs until Section 5.3.6.3.

The overheads are broken down according to the signature's components: the Bloom filter itself (column 2), and those from the hash function (columns 3 and 4). The overall overheads for H_3 and PBX signatures are summarized in columns 5 and 6, respectively. Our discussion focuses on signatures with two hash functions. Overall, PBX hash functions are 19% smaller than the

same size signature implemented with two H_3 hash functions. H_3 hash trees consume approximately 21% of the total H_3 signature area overhead, while PBX hash functions only consume 1.5% of the total PBX signature area overhead.

The power results of 2kb H_3 and PBX signatures are shown in Table 5-2. Overall, 2kb PBX signatures (with two hash functions) use 4.3% less power than H_3 signatures. H_3 hash trees account for 5% of the H_3 signature's total power. However, PBX hash trees consume less than 1% of the total PBX signature's power.

In summary, signatures which use PBX instead of H_3 hash functions require less area and save power (up to 21% savings for area and 4.7% savings for power). The savings increase as more hash functions are used.

TABLE 5-3. Area and power overheads of PBX signatures

	Area (mm^2)			Power (mW)		
Sig. (k=2)	Bloom filter	PBX hash	PBX sig.	Bloom filter	PBX hash	PBX sig.
1kb	2.33e-3	4.00e-5	2.37e-3	9.20e1	5.12e-1	9.25e1
2kb	3.00e-3	4.67e-5	3.05e-3	1.10e2	5.68e-1	1.11e2
4kb	5.33e-3	5.17e-5	5.39e-3	1.10e2	6.27e-1	1.11e2
8kb	1.03e-2	5.17e-5	1.04e-2	1.40e2	6.83e-1	1.41e2
16kb	1.27e-2	5.83e-5	1.27e-2	1.60e2	7.39e-1	1.61e2
32kb	1.67e-2	6.00e-5	1.67e-2	2.00e2	7.95e-1	2.01e2
64kb	3.33e-2	6.83e-5	3.34e-2	1.90e2	8.55e-1	1.91e2

5.3.6.3 Overheads of Larger PBX Signatures

Although larger signature sizes reduce false positives, they do so at the cost of increased area and power. We calculate the area and power overhead trends of different PBX signatures ranging from 1kb to 64kb, all using two hash functions. The results are shown in Table 5-3. The area results in columns 2-4 are for 65nm technology (calculated by dividing 400nm area estimates by 36), and the power results in columns 5-7 are for 400nm technology. First, we examine the area and power overheads of different signature configurations relative to each other (columns 4 and 7, respectively). For example, increasing the signature size from 1kb to 4kb increases the signature's area overhead by 127% and power by 20%. Using 16kb signatures instead of 4kb signatures increases the signature's area overhead by 136% and power by 45%. Finally, increasing from

TABLE 5-4. Area overheads of PBX signatures in two modern processors designs

	POWER6, 47mm^2 core		Niagara, 13mm^2 core	
Sig. (k=2)	PBX area (mm^2)	% core area for sig.	PBX area (mm^2)	% core area for sig.
1kb	9.33e-3	0.02	5.50e-2	0.42
2kb	1.20e-2	0.02	7.00e-2	0.54
4kb	2.17e-2	0.05	1.23e-1	0.94
8kb	4.17e-2	0.09	2.38e-1	1.83
16kb	5.00e-2	0.11	2.93e-1	2.25
32kb	6.67e-2	0.14	3.85e-1	2.96
64kb	1.33e-1	0.28	7.68e-1	5.90

16kb to 64kb signatures increase area by 162% and power by 19%. Overall, larger signatures increase both area and power overheads.

Second, we compare signature area overheads relative to the area of two current micro-processor designs, the IBM POWER6 [65] and the Sun Niagara [61]. POWER6 is implemented in 65nm technology. The chip itself is 341mm^2, is dual-core, and each core is 47mm^2 (measured from a die photo) and supports two SMT thread contexts. Niagara is implemented in 90nm technology. The chip area is 379mm^2, is eight-core, and each core is 13mm^2 (measured from a die photo) and supports four thread contexts. We calculate the total per-core area overhead consumed by the PBX signatures after linearly scaling for technology (dividing 400nm area estimates by 36 for 65nm or dividing by 16 for 90nm). We assume each thread context has a set of read and write signatures. We then compute the percentage of per-core area consumed by the per-core PBX signatures. The results are shown in Table 5-4. From the table it is evident that for the larger POWER6 cores PBX signatures do not consume a significant fraction of the core's total area. The largest 64kb signatures consume only 0.28% of the core's total area. However, this trend does not hold for the smaller Niagara core. Although 1kb signatures consume only 0.42% of core area, this overhead increases to 5.9% with 64kb signatures. Thus, with smaller core designs it is important to carefully consider the area overheads of using larger signature sizes.

Although PBX reduces signature implementation overheads, it does not mitigate false conflicts arising from signature bits set by private memory references. We now examine how Notary's technique of privatization reduces this type of false conflict.

5.4 Notary's Technique of Privatization

5.4.1 Motivation

Previous signature proposals do not associate high-level semantic information with inserted addresses. All addresses are treated with equal importance. However, data for many multi-threaded programs can be segregated into private and shared, according to whether the data is accessible to a single thread or shared amongst more than one thread. The process of performing this categorization is called privatization [109]. With information about private and shared memory accesses, the TM system has the opportunity to only isolate shared addresses and drop isolation on private addresses. Fewer addresses are inserted in signatures, and fewer signature bits will be set. Thus the probability of false positives on lookups will decrease. Although conceptually simple, prior research has not described how to implement this support for HTMs to handle memory objects of arbitrary sizes.

5.4.2 Methods and Results

In order to illustrate the performance benefits of privatization, we execute a micro-benchmark containing a constant number of fixed-sized transactions, and only vary the fraction of the transaction's memory requests that are privatized (from 0.0 to 1.0, with 1.0 denoting all requests being privatized). We examine two different signature sizes (512b and 2kb) and two different numbers of hash functions (two and four). The micro-benchmark results are shown in Figure 5-6.

Focusing on the results for the 2kb signatures, we have several observations. First, privatization can greatly improve execution time for signatures. For example, with two hashes, 50% privatization leads to a 15% improvement in normalized execution time (compared to no privatization), and increases to a 32% improvement for 95% privatization. Second, the improvement benefits are still significant even after using more hash functions. For four hash functions, there is a 13% improvement in normalized execution time for 50% privatization, and 20% improvement for 95% privatization. Similar trends exist for the smaller 512b signature. We defer examining the results of our macro-benchmarks until Section 5.6.2.

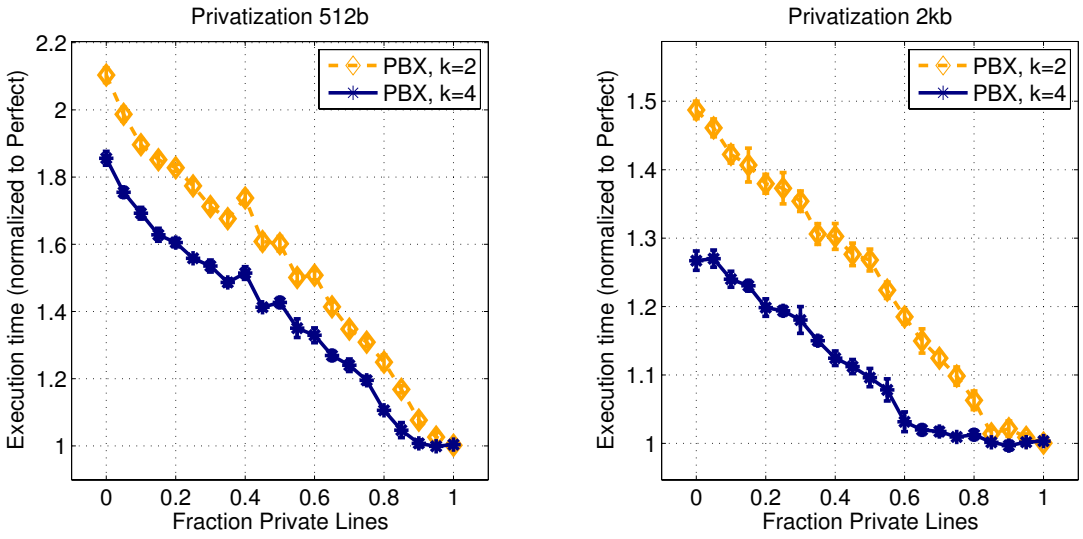


FIGURE 5-6. Privatization micro-benchmark.

5.4.3 Exploiting Privatization

5.4.3.4 Prior TM Privatization Approaches

Researchers have proposed programmer-visible privatization language constructs, but these constructs have been targeted for software TM systems (STMs) or hybrid TMs. Little research has been done to show how HTMs can take advantage of privatization for data structures of arbitrary sizes.

For example, Scott et al. [99] describe the use of four varieties of pointers which guard accesses to transactional objects. Two of these pointer types, `sh_ptr<T>` and `un_ptr<T>`, denote transactionally shared and transactionally private data, respectively. Furthermore, researchers recently proposed a transactional version of OpenMP [76], which support the additional keywords `exclude` and `only` to denote which variables to exclude and track for conflict detection. Similarly, OpenTM [9] supports the keywords `private` and `shared`.

A possible implementation of these interfaces for HTMs is for the compiler to produce code using special load and store instructions for private object accesses. Then the HTM system can elide transactional isolation for these instructions. However special instructions can increase the complexity of memory instructions in already-complex instruction set architectures (e.g., loads and stores to special ASIs in SPARC). Furthermore, compilers may not guarantee which, if any, memory accesses can be converted to privatized memory accesses, especially with the use of pointers in a program.

Abadi et al. [1] expose the functions `protect()` and `unprotect()` to the programmer to denote when C# objects are safe to use inside and outside of transactions, respectively. This interface is functionally similar to Notary's barriers (described below). Matveev et al. [71] propose adding a Virtual Memory Filter to cores and having the programmer declare a range of memory locations

TABLE 5-5. Notary's privatization programming interface

Privatization function	Usage
`shared_malloc(size), private_malloc(size)`	Dynamic allocation of shared and private memory objects.
`shared_free(ptr), private_free(ptr)`	Frees up memory allocated by shared or private allocators.
`privatize_barrier(num_threads,ptr, size), publicize_barrier(num_threads, ptr, size)`	Program threads come to a common point to privatize or publicize an object. Must be used outside of transactions.

to be transactional. This interface, although more general, is functionally similar to Notary's privatization interface. Both interfaces were concurrently proposed with Notary.

5.4.3.5 Notary's Privatization Support

Notary's privatization is implemented through a combination of compiler, library/runtime, operating system (OS), and minimal hardware support. The programmer invokes privatization through a set of library functions, and defines which objects are private or shared. Both static and dynamic variants of privatization are supported. Objects do not change sharing status in static privatization, whereas changes can occur in dynamic privatization.

The process of marking objects as private must be done with care, as mistakenly marking objects as private can lead to program errors. If there is any doubt on an object's sharing status, the programmer should mark it as shared. This might lead to performance degradations but not to

```
s_array = (int*)shared_malloc(arraysize*sizeof(int));
xact_begin;
p_array = (int*)private_malloc(arraysize*sizeof(int));
// no conflict detection on p_array
p_array[i] = myid;
// detect conflicts on s_array
s_array[i] = myid*10;
// delete p_array
private_free(p_array);
...
xact_end;
// This might violate strong atomicity
s_array[j]=myid*15;
// delete s_array
shared_free(s_array);
```

(a) private and shared memory allocation

```
x = (int*) private_malloc(arraysize*sizeof(int));
// use x as private variable
...
// transition x from private to shared
publicize_barrier(numthreads,x,arraysize*sizeof(int));
xact_begin;
// use x as shared variable
x[i] = myid;
...
xact_end;
// transition x from shared to private
privatize_barrier(numthreads,x,arraysize*sizeof(int));
```

(b) dynamic privatization using barriers

FIGURE 5-7. Example codes using Notary's privatization API.

correctness problems. Table 5-5 summarizes Notary's programming interface for privatization, which is described in more detail below.

Interfaces. If stack data is not shared amongst threads, the library/runtime statically marks all stack pages as private. Otherwise stack data is treated as equivalent to shared data, and handled similarly in the following discussion. Based on our experience, stack addresses may be passed to functions called and executed by a single thread (e.g., to update return values). But it is very unlikely that common code would pass stack addresses of variables belonging to a thread to a different thread that would later update these variables. Hence it is unlikely that stack pages would be both shared and private in the same program. In addition, modern languages such as Java prohibit explicit manipulation of stack addresses in programs, and the scenarios described above are not possible in these languages.

Heap-allocated data complicates privatization because most compilers cannot statically guarantee whether the data will remain private or shared throughout its lifetime, particularly when pointers are explicitly used (e.g., C/C++). We avoid this problem by providing library functions for dynamic shared and private memory allocation (`shared_malloc()` and `private_malloc()`), and the freeing of those dynamic memory (`shared_free()` and `private_free()`). Figure 5-7(a) shows an example code snippet using these allocation routines.

In order to support dynamic privatization, Notary uses barriers to synchronize a group of threads on the sharing status of memory objects. This mechanism is inspired by Scott et al. from their work on privatization in Delaunay triangulation [99], and we extend it to allow the changing of an object's sharing status in both directions (from private to shared, and vice versa). These functions are `publicize_barrier()` and `privatize_barrier()`. In order to avoid potential deadlocks

between threads, Notary prohibits these barriers from being used inside of transactions. Barrier functions take three arguments: the number of threads in the synchronization group to wait for before privatizing or publicizing the object, and the pointer and size of the object being made public or private. Figure 5-7(b) illustrates an example barrier usage.

Notary's page-based implementation. One possible implementation of Notary's privatization interfaces is to associate an object's sharing status with a page, and store the sharing status with address translation information. To implement `private_malloc()` and `shared_malloc()`, the library/runtime needs to actively separate private and shared objects with help from the OS. The OS enforces separation by allocating private and shared objects on separate virtual pages (and separate page frames), and marks shared pages with a privatization-specific "shared" bit. All objects on a specific page with shared or private status inherit that page's status, and a good policy for reducing the number of pages used is to co-locate objects with the same status on the same page. Any object with unknown status is considered shared for correct conflict detection. Meta-data on objects and their sizes are stored in OS-accessible data structures.

The page-based bits can be associated with existing virtual address translation information and cached in hardware TLB structures. The HTM reads the shared bit during address translation (along with the current transactional mode of the thread context) to decide whether to isolate the memory address in the read- and write-sets (the read and write signatures for signature-based TM systems).

The barrier functions `privatize_barrier()` and `publicize_barrier()` can be implemented purely in software or, if available, with hardware barrier instructions. In the OS, these barriers translate to TLB-shootdowns, in which the sharing status for the page is set or unset. Multiple objects on

the same page can be affected, and Notary can handle these cases with different levels of implementation complexity. When changing an object from private to shared, the OS can safely mark the entire page as shared and still maintain correct conflict detection. When changing an object from shared to private Notary's default action is to change the page's status only if there are no additional shared objects on the same page. The OS detects this case by examining the meta-data for objects on that page. In more complex implementations the OS throws an exception about the object being privatized. One way for the programmer to handle this exception is to free up this object's memory (with `shared_free()`), and re-allocate its memory using `private_malloc()`. If these actions cause a significant performance penalty, an alternative implementation may be needed. If fine-grained memory protection support is available (e.g., Mondrian [122]), the sharing status on individual memory words can change transparently to the programmer.

Although we lack evidence that a page-based implementation is best, our implementation is reasonable due to little changes in the hardware and operating system. Furthermore, software such as memory allocators and operating systems already manipulate pages. Fine-grained (e.g., object-sized) implementations may be needed if trade-offs change. For example, an alternative implementation may use more hardware in order to reduce the possible increase in physical page frames (and corresponding memory size) used by a program. In addition, page frame fragmentation in our page-based implementation could be problematic if objects repeatedly change sharing status, and may lead to the need to periodically coalesce objects with the same status on different physical page frames onto the same physical page frame.

5.5 Platform & Methodology

5.5.1 Base CMP System

Our simulations model a 16-core CMP system. Chapter 3 describes the details of our system. Due to better scalability [57], BIND runs on top of a 32-core CMP with the same memory system and interconnect parameters as above. The Prefetch workload also uses a 32-core CMP for higher contention in transactions. The CMP implements the LogTM-SE [127] HTM system.

5.5.2 Workloads

We use fifteen transactional workloads. They consist of four micro-benchmarks (BTree, Sparse Matrix, LFUCache, and Prefetch), three SPLASH-2 applications (Barnes, Raytrace, and Mp3d), seven STAMP applications (Bayes, Delaunay, Labyrinth, Vacation, Genome, Intruder, and Yada), and one domain name service (DNS) application (BIND). Chapter 3 provides descriptions and characteristics of our workloads.

5.5.3 Simulation Methodology

We evaluate Notary using full-system simulation, and details are provided in Chapter 3.

5.5.4 Signature Configurations

We simulate infinite-size, unimplementable “perfect” signatures (with no false positives), and finite signatures of size 64b to 64kb (8B to 8kB). 64b signatures match the word size in current cores, and 64kb signatures match the performance of perfect signatures for all workloads. LogTM-SE uses separate read and write signatures, with two hash functions for each signature.

One hash takes even-numbered bits from the bit-fields, and the other odd-numbered bits. We simulate both PBX and H_3 signatures. PBX uses an 11-bit cache-index bit-field and a 10-bit non-overlapping PPN bit-field. H_3 uses a fixed pseudo-random Boolean matrix.

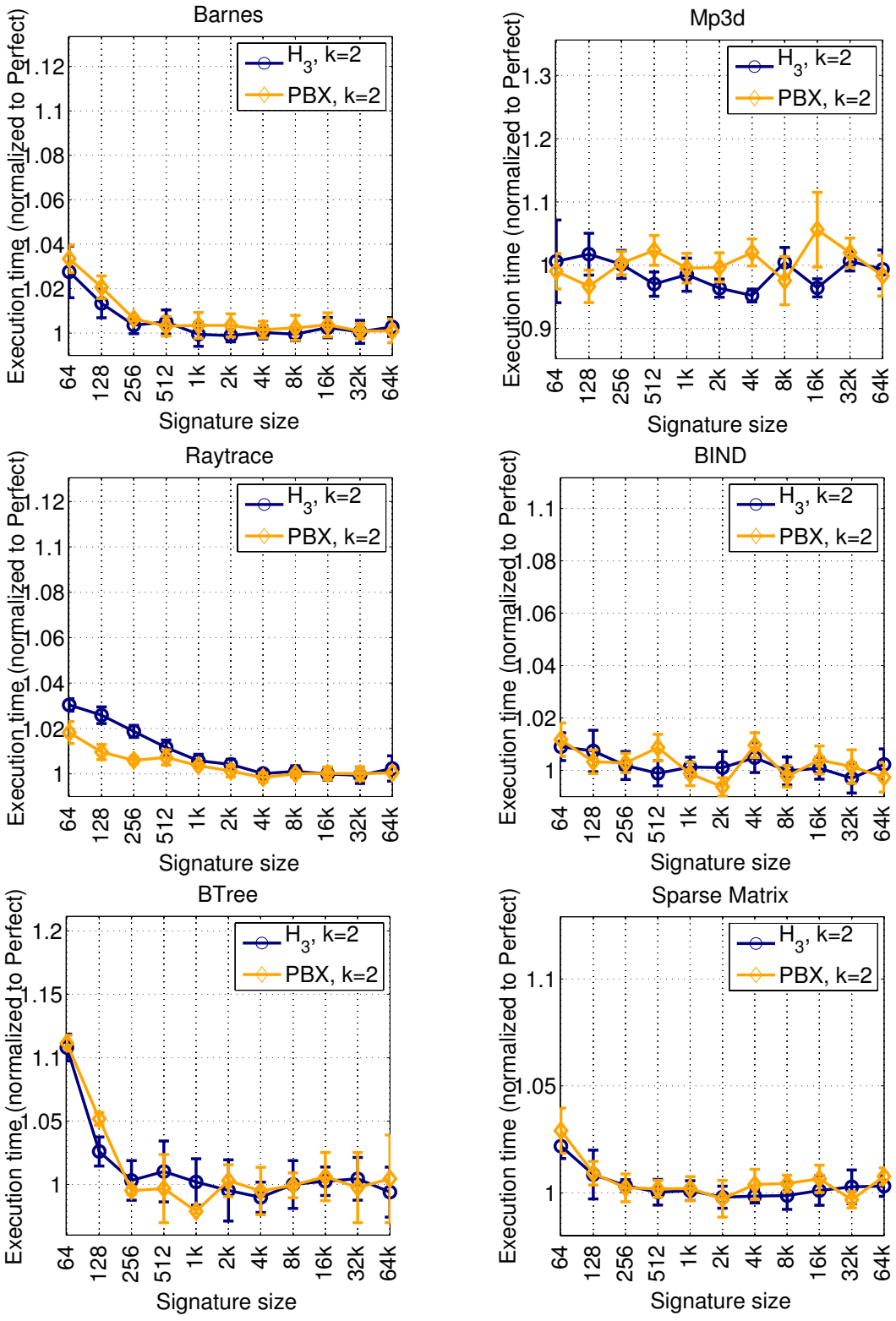


FIGURE 5-8. Execution time results of PBX versus H_3 hashing.

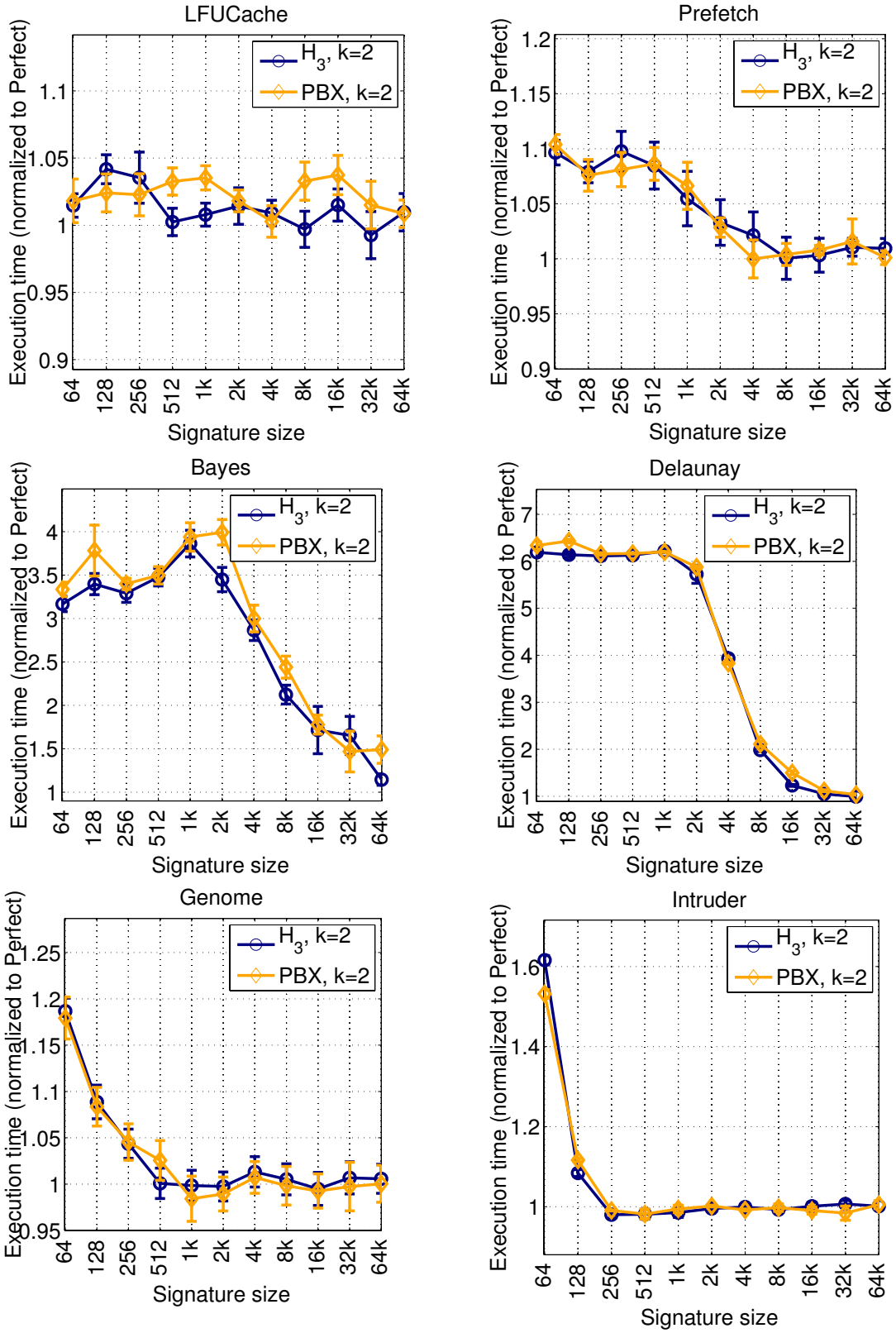


FIGURE 5-8. Execution time results of PBX versus H_3 hashing.

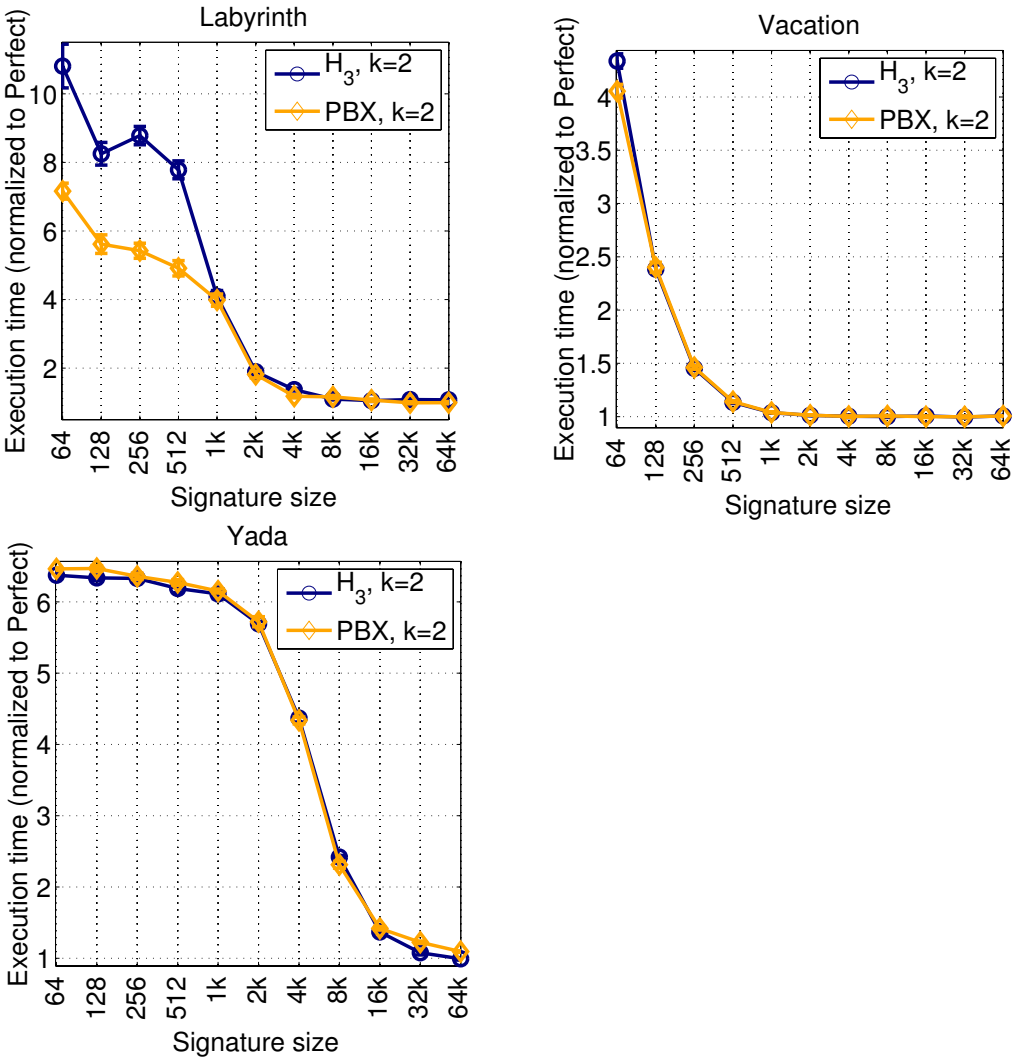


FIGURE 5-8. Execution time results of PBX versus H_3 hashing.

5.6 Evaluation Results

We first present execution time results of PBX versus H_3, and then the execution time results for Notary's privatization technique. All results are normalized to perfect signatures. Appendix B presents the raw execution times of perfect signatures for all our workloads.

5.6.1 Effectiveness of PBX Versus H_3 Hashing

Figure 5-8 shows execution times of PBX compared to H_3. For the fifteen workloads, the execution times for PBX and H_3 are similar, regardless of the signature size. Because they are different signature implementations, PBX may sometimes out-perform H_3 due to differences in signature utilization and conflict formation amongst concurrent transactions. In Labyrinth the differences arise from additional conflicts for H_3 in a latency-critical portion of a transaction, which copies a shared grid structure to a local copy of the grid. These conflicts are due to H_3 setting more signature bits than PBX in this transaction. This also occurs in Raytrace, in which H_3 sets more signature bits and leads to more non-transactional conflicts than PBX.

Implication 1: PBX achieves similar performance to H_3 hashing, but does so with much lower hardware cost.

5.6.2 Effectiveness of Privatization

We simulate privatization in two stages. First, all stack references are marked private, as stack data are thread-private in our workloads. We find the stack regions in programs by using the `pmap` utility in Solaris. Privatizing the stack did not significantly reduce execution time for our workloads. This is because the majority of memory references are to non-stack data. The full runtime results of this process is described in Section 5.7.2.

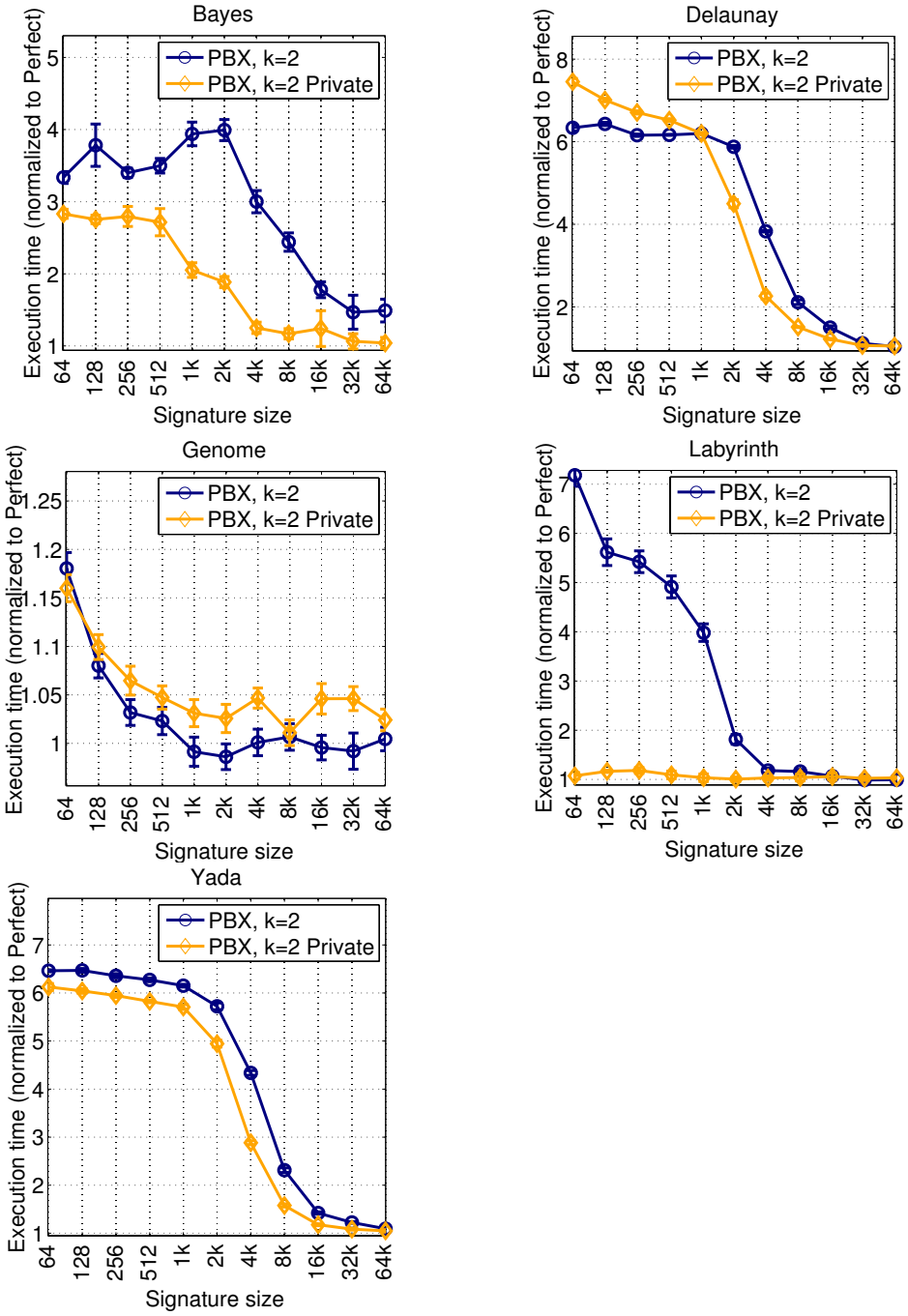


FIGURE 5-9. Privatization execution times.

Second, we simulate `private_malloc()` and `shared_malloc()` through separate heap regions that are pre-allocated at the beginning of the program. These regions service memory allocation requests. Isolation is dropped on the private heap region. We only show results for selected

STAMP workloads. The remaining workloads either did not have a high transactional duty cycle or did not use any private heap data structures in transactions (and thus would not benefit from privatization). The results after both stages of privatization were completed are shown in Figure 5-9.

This figure illustrates several interesting trends. First, privatization can reduce execution time. This allows smaller signatures to be used, which is attractive since in Section 5.3.6 we showed the power and area savings of smaller signatures. For example, in Bayes the execution time of 512b signatures after privatization is approximately equivalent to that of 8kb signatures before privatization (due to elimination of all false conflicts in the write signature). This is a reduction of 16x in the signature size needed! Second, these results parallel the results of our privatization micro-benchmark. Labyrinth is the workload that benefits most from privatization (approximately 86% reduction in normalized execution time from reducing false conflicts by 96%), and Table 5-6 reveals why this occurs. The largest reductions in transactional read- and write-set sizes occur in Labyrinth, with 88% and 94% reductions, respectively. Most of its reductions come from not isolating requests to per-thread private copies of the grid structure. In Genome, privatization is sometimes slower than the original because of an increased number of exceptions (e.g., register spill/

TABLE 5-6. Read- & write-set sizes

Benchmark	Before / After Privatization		% Reduction	
	Avg. read-set	Avg. write-set	Avg. read-set	Avg. write-set
Bayes	73 / 33	37 / 2.4	55	94
Delaunay	68 / 52	44 / 30	23	32
Genome	15 / 12	1.7 / 1.6	20	5.9
Labyrinth	70 / 8.5	91 / 5.1	88	94
Yada	61 / 47	36 / 24	23	33

fill, TLB misses). As a result, this increase in exception frequency can exercise different code paths (both user- and supervisor-level), which leads to different conflict formations that can be worse than the original program. In Delaunay, privatization sometimes hurts the smaller (e.g., 64b) signatures because it significantly alters conflict formation. After privatization, conflicts occur more frequently and at different locations in the program.

Implication 2: Privatization can be an effective technique in certain workloads to improve TM program execution time by reducing signature false positives.

The preceding results are gathered for signatures that use two hash functions. Section 5.7.3 describes sensitivity results of H_3 versus PBX, as well as privatization, both for four hash functions.

5.6.3 Notary's Applicability to Other Systems

PBX hashing may be implemented in any TM system which uses signatures (e.g., Bulk, LogTM-SE, SigTM) or in any signature-based non-TM systems (e.g., BulkSC, SoftSig, Atom-Aid, DeLorean, Rerun).

Notary's privatization support can be implemented on top of other TM systems. Privatization is attractive for software TMs which may incur high transactional bookkeeping overheads. It helps TM systems that store new values in place (by reducing abort overheads) and those that store new values in a separate buffer (by reducing commit overheads). For signature-based TMs, privatization can greatly reduce signature false conflicts. Signature-based non-TM systems can also use it. Programs which correctly classify and synchronize accesses to private data are not susceptible to memory races or memory consistency violations on that data.

Notary can also be used in conjunction with other techniques aimed at optimizing signature sizes. One such technique customizes signature sizes based on profile information of transaction read- and write-set sizes [77].

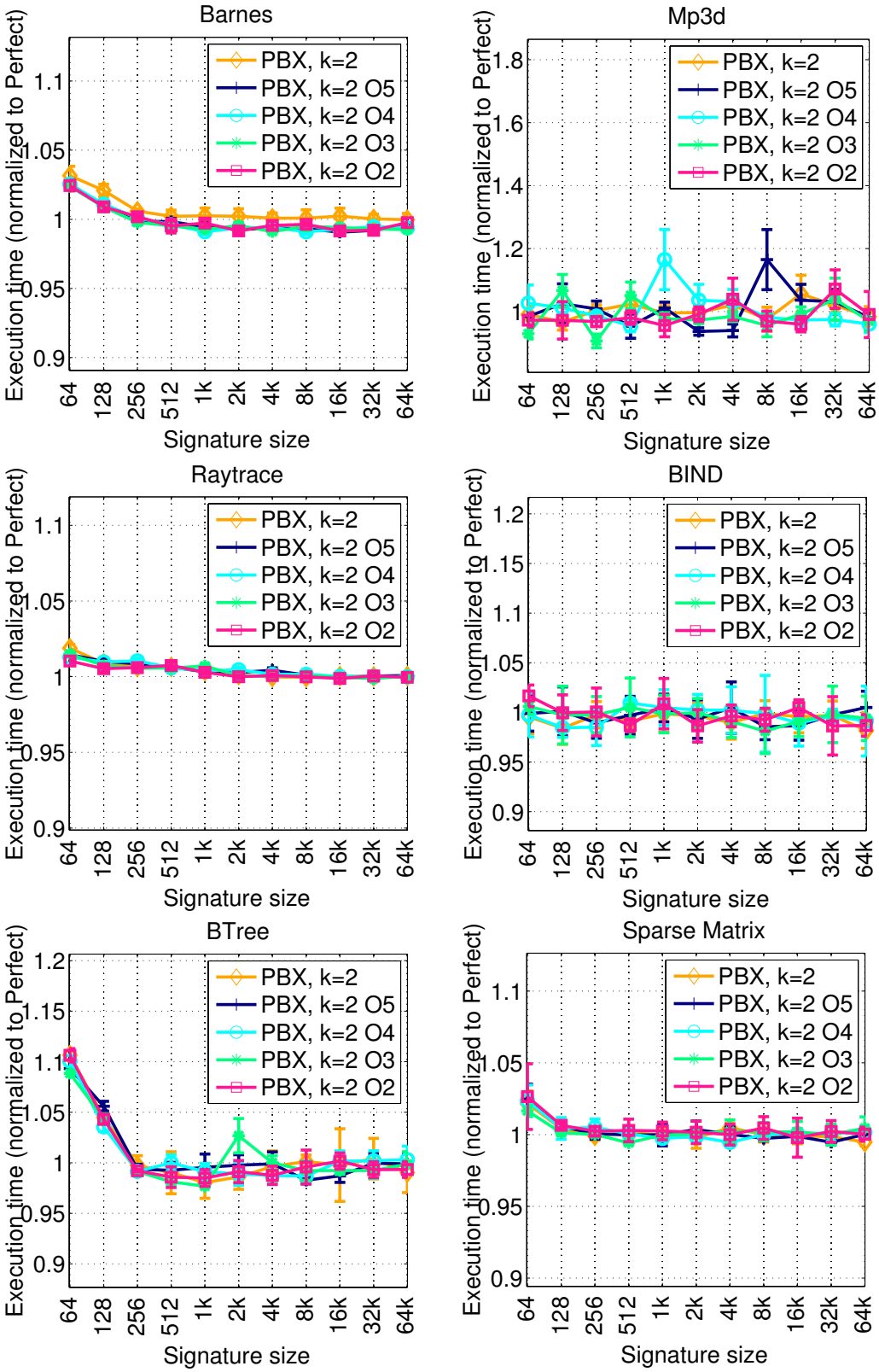


FIGURE 5-10. Execution times assuming PPN and cache-index bit-fields overlap.

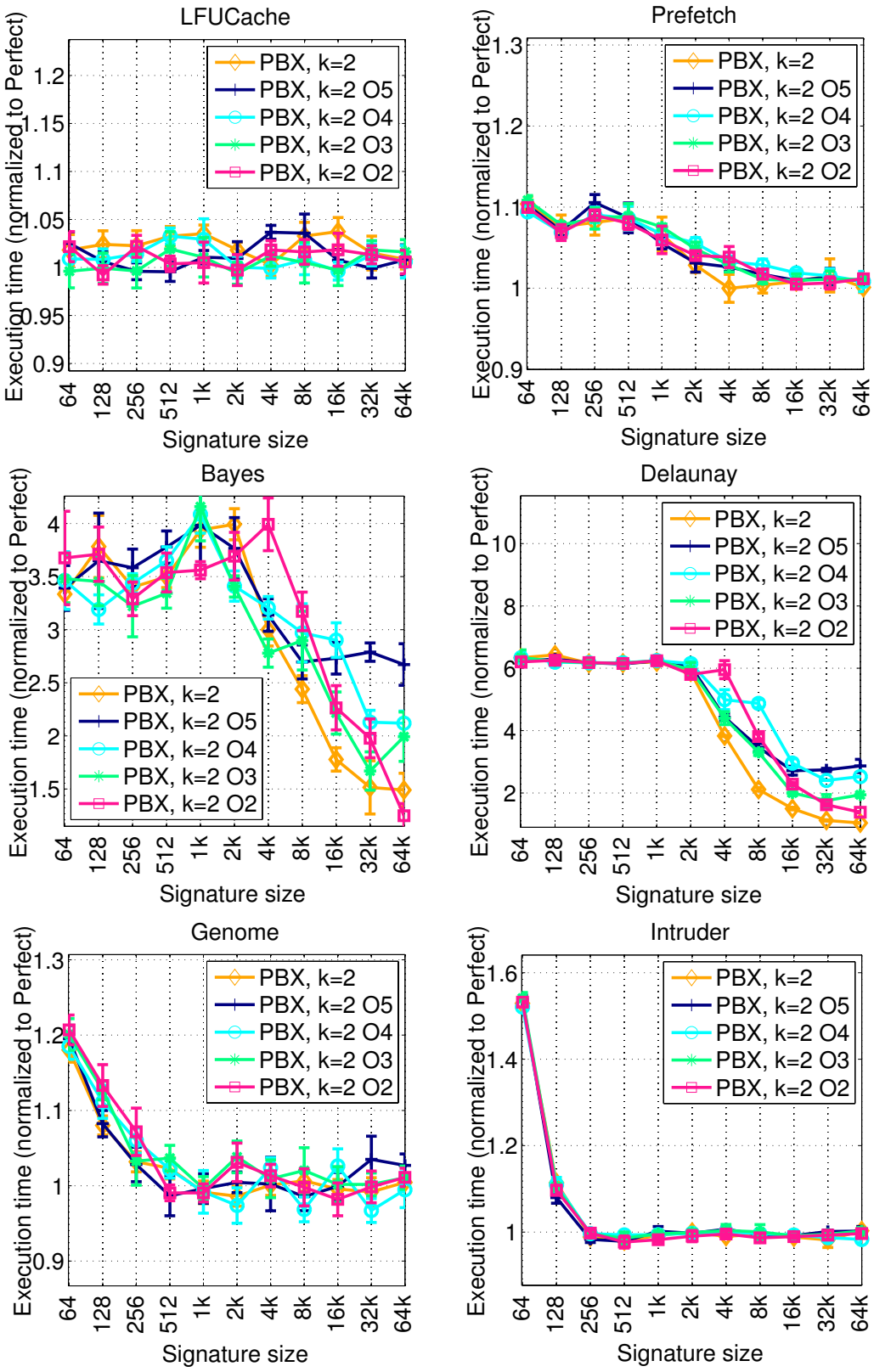


FIGURE 5-10. Execution times assuming PPN and cache-index bit-fields overlap.

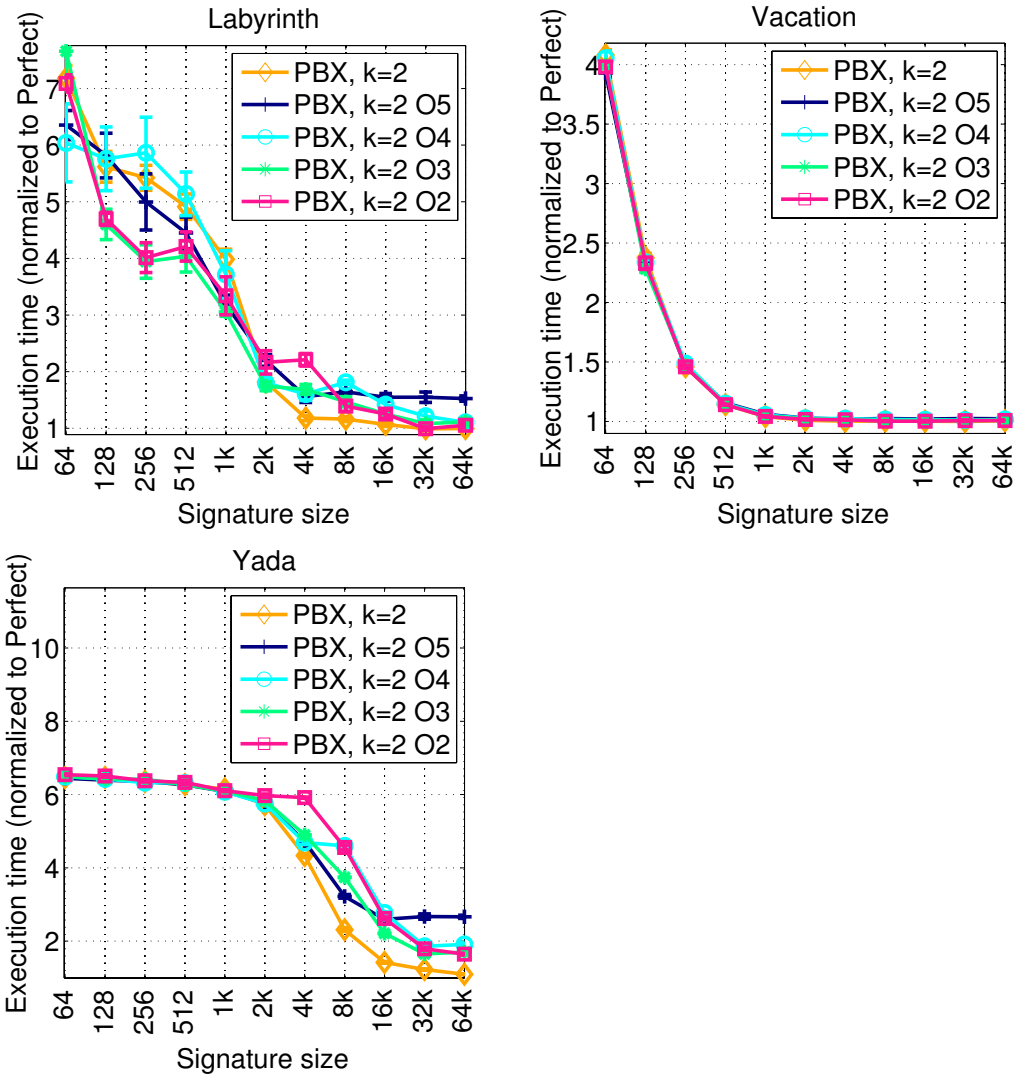


FIGURE 5-10. Execution times assuming PPN and cache-index bit-fields overlap.

5.7 Sensitivity Analysis

5.7.1 Sensitivity to Overlaps Between PPN and Cache-index Bit-fields

The results presented in this chapter assume that the cache-index and PPN bit-fields used in PBX do not overlap. That is, these bit-fields do not use any of the same input address bits. If the two bit-fields overlap, the resultant PBX hash values may experience correlation. This correlation may skew the distribution of hash values from the ideal uniform distribution. Hash values with a uniform distribution can index all signature bit indices. A non-uniform distribution can prevent some signature bit indices from being accessed. This reduction in the range of hash values harms performance in larger signatures, because the non-uniform distribution of hash values can make larger signatures perform like smaller signatures.

In order to illustrate the negative results of bit-field overlaps in PBX hashing, we also run experiments in which we vary the amount of overlap from 2-5 bits. This overlap occurs in the bits which denote the end of the cache-index bit-field and the start of the PPN bit-field. Figure 5-10 shows our results. The line denoting 2 bits of overlap is labeled “PBX, k=2 O2”, and 5 bits of overlap is the line labeled “PBX, k=2 O5”. We include the results of no overlap (“PBX, k=2”) as reference.

Overall, bit-field overlaps have a negative performance impact on large signatures that operate on large transactions. These conditions stress the non-uniform hash value distribution created by overlapping bit-fields. Examples of workloads exhibiting negative performance impact include Bayes, Delaunay, Labyrinth, and Yada. For large (i.e., >= 4kb) signature sizes, overlaps lead to execution times that are worse than the base no-overlap PBX signature. Moreover, the execution times get progressively worse as larger signature sizes are used. These effects are most prominent

in the results with 5 bits of overlap. We also separately verified that overlaps lead to increased signature false positives and reduce that range of signature bit indices that are accessed.

In summary, we find that bit-field overlaps between the cache-index and PPN bit-fields can lead to detrimental performance for PBX signatures. We recommend PBX designs which avoid overlaps in its two bit-fields.

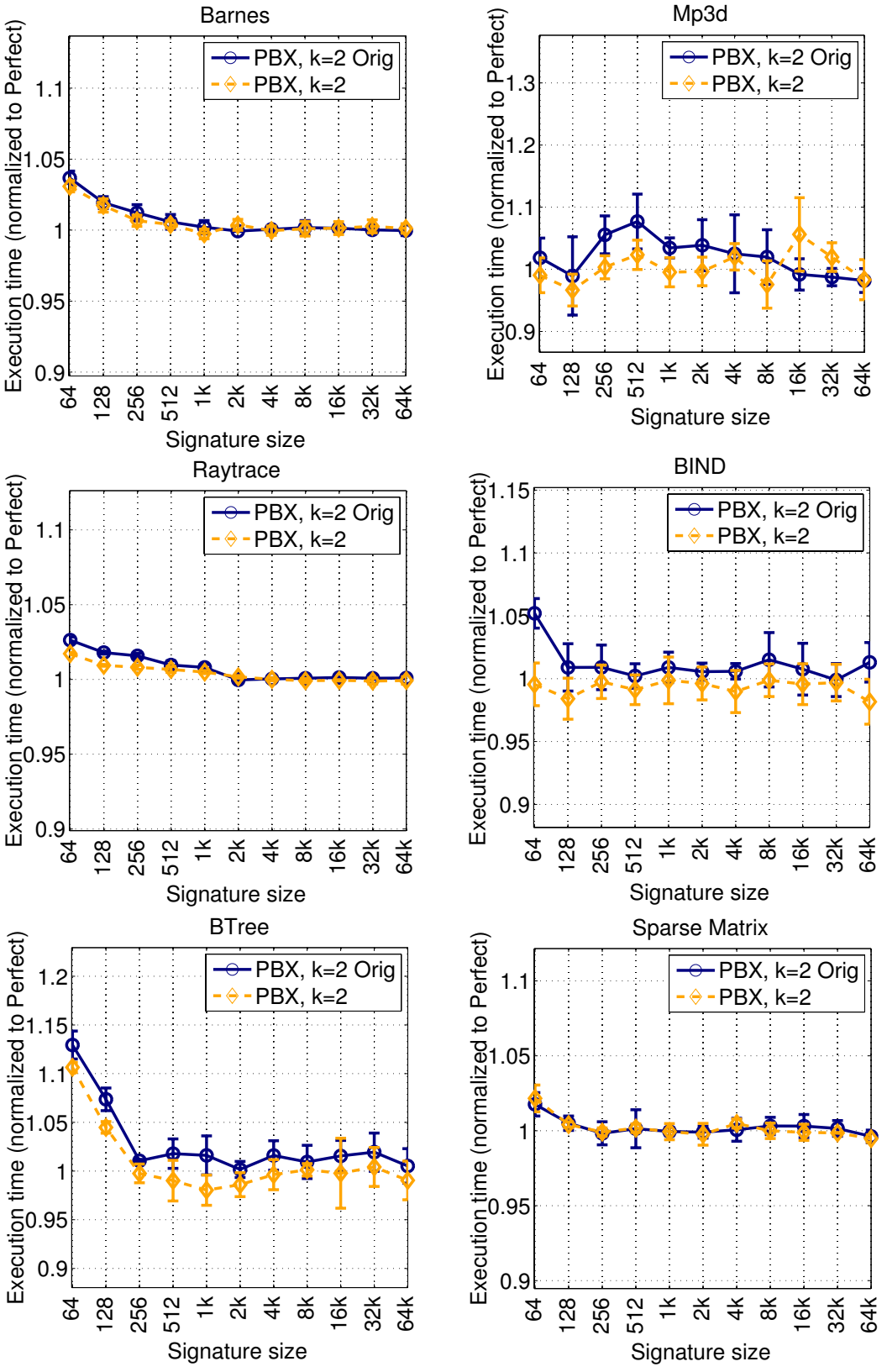


FIGURE 5-11. Normalized execution times before and after stack references are removed.

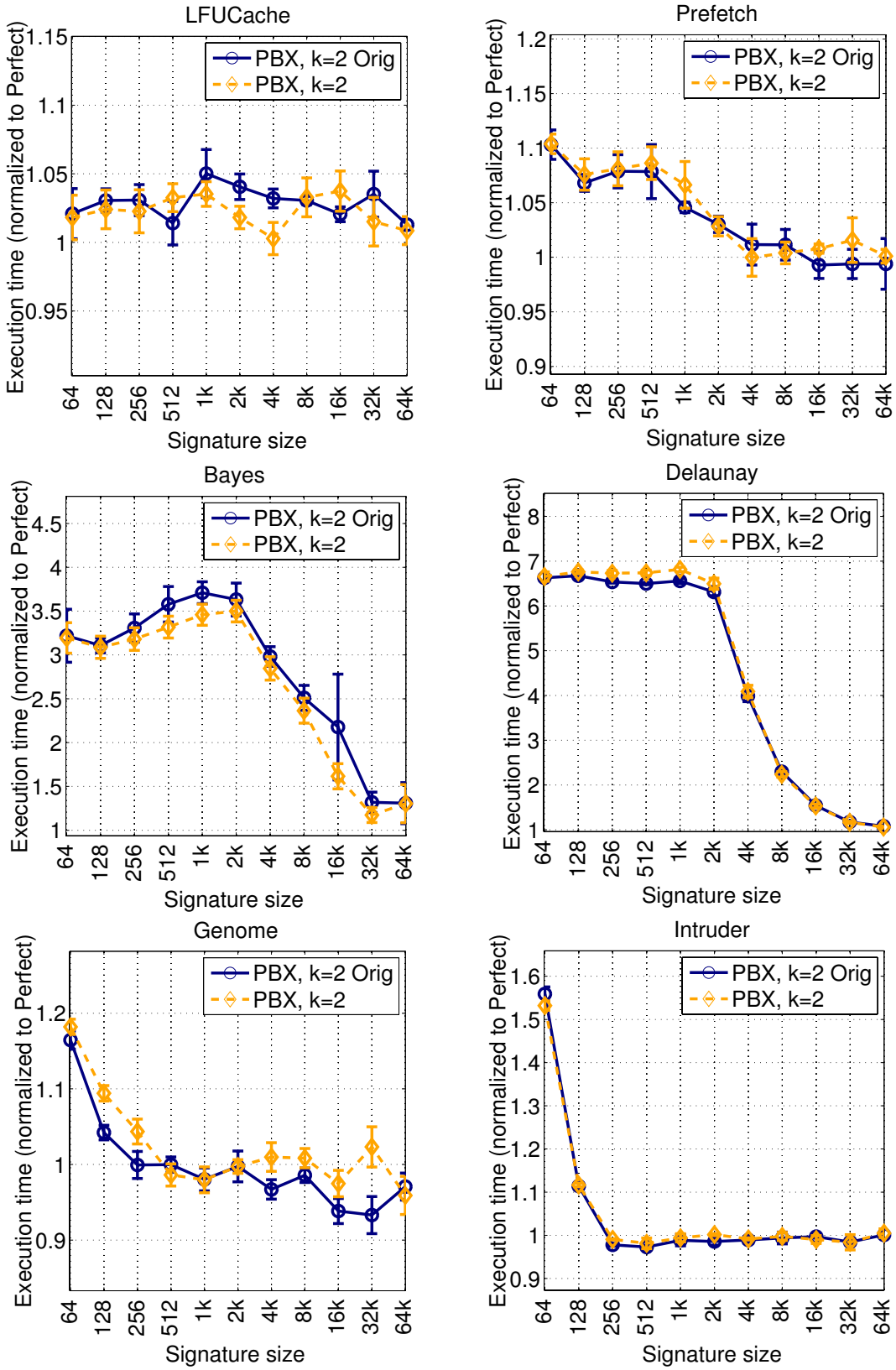


FIGURE 5-11. Normalized execution times before and after stack references are removed.

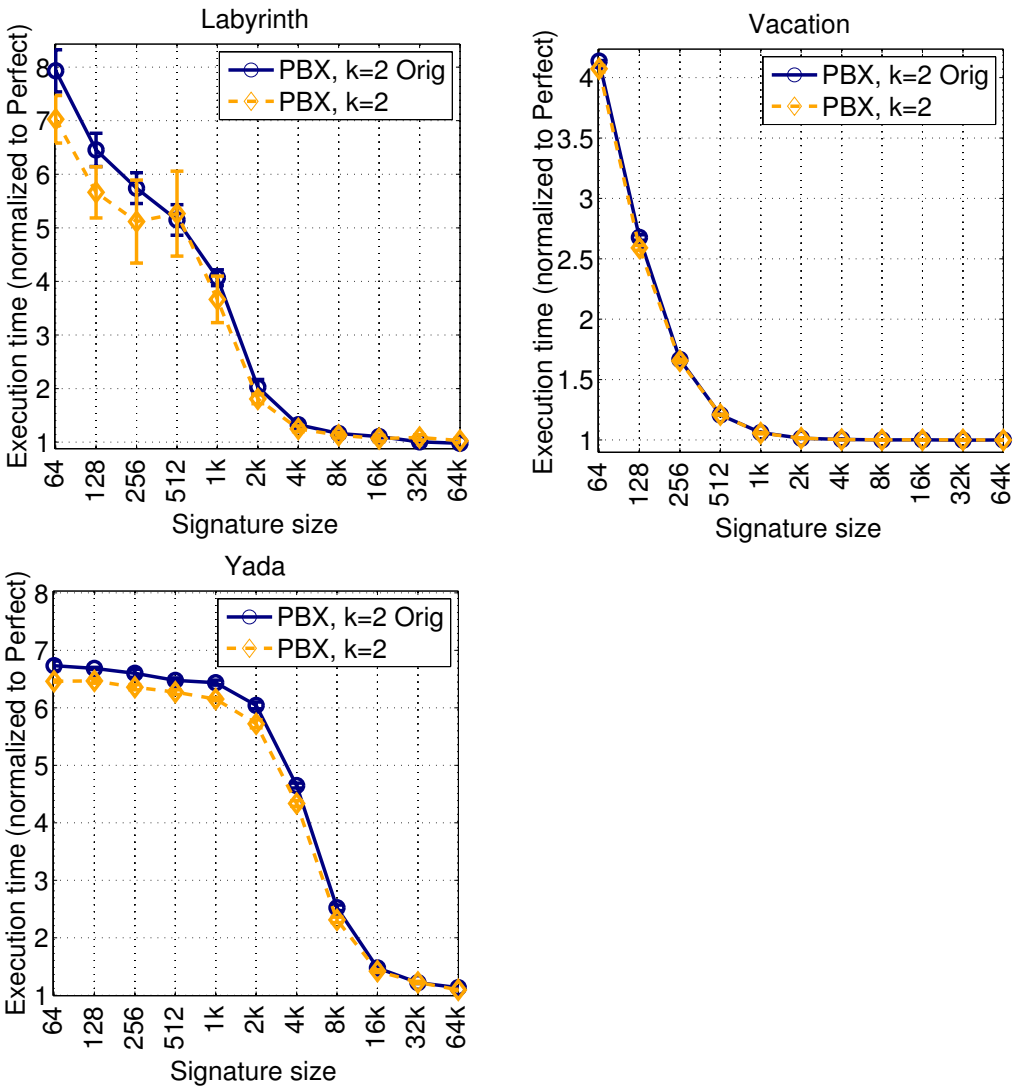


FIGURE 5-11. Normalized execution times before and after stack references are removed.

5.7.2 Sensitivity to Removing Stack References

Figure 5-11 shows the normalized execution time results before and after stack references are removed from signatures. The results for the original program (with stack references) is the line labeled “PBX, k=2 Orig”. As the figure shows, for the majority of workloads there are no significant differences between the execution times before and after stack references are removed. Exceptions occur in BIND and Labyrinth. In BIND, removing stack references causes the 64b signature size to perform equivalent to perfect signatures. In Labyrinth, removing stack references reduces the false positive rates in both signatures, and sometimes improves execution time in the 64b and 128b signatures.

The lack of execution time differences is likely due to the instruction-set-architecture (ISA) used in our experiments, which is SPARC. Our in-depth analysis of transactional addresses reveals that most of the addresses are not stack references. This is likely due to the number of architectural registers in SPARC that removes most of the stack activity (i.e., most of the operands are passed via architectural registers and not the stack). Other ISAs such as x86 may experience a larger fraction of addresses to the stack (due to fewer architectural registers than SPARC). These ISAs may have a significant improvement in execution time after stack addresses are removed from signatures.

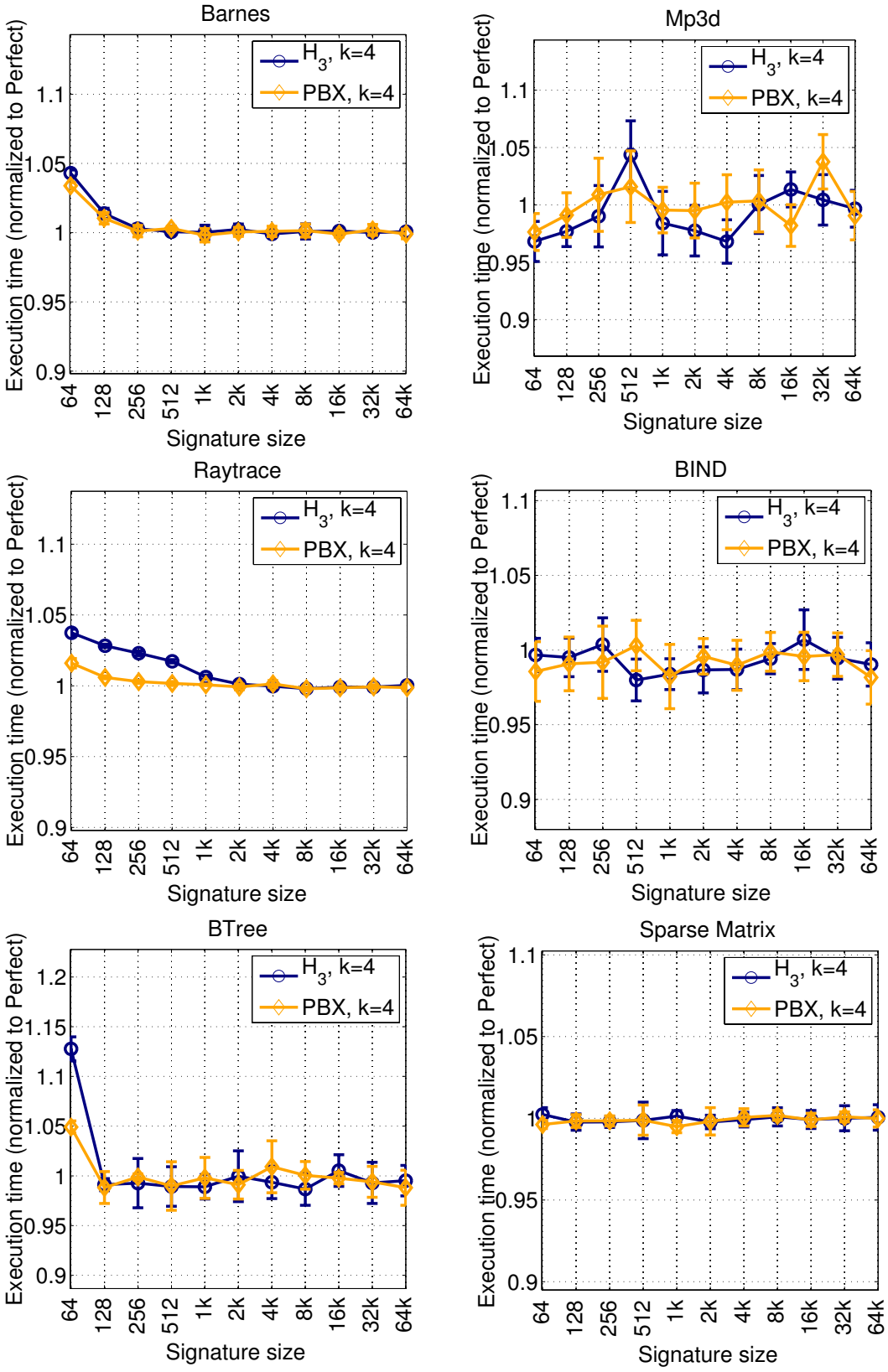


FIGURE 5-12. Normalized execution times of H_3 and PBX signatures with four hashes.

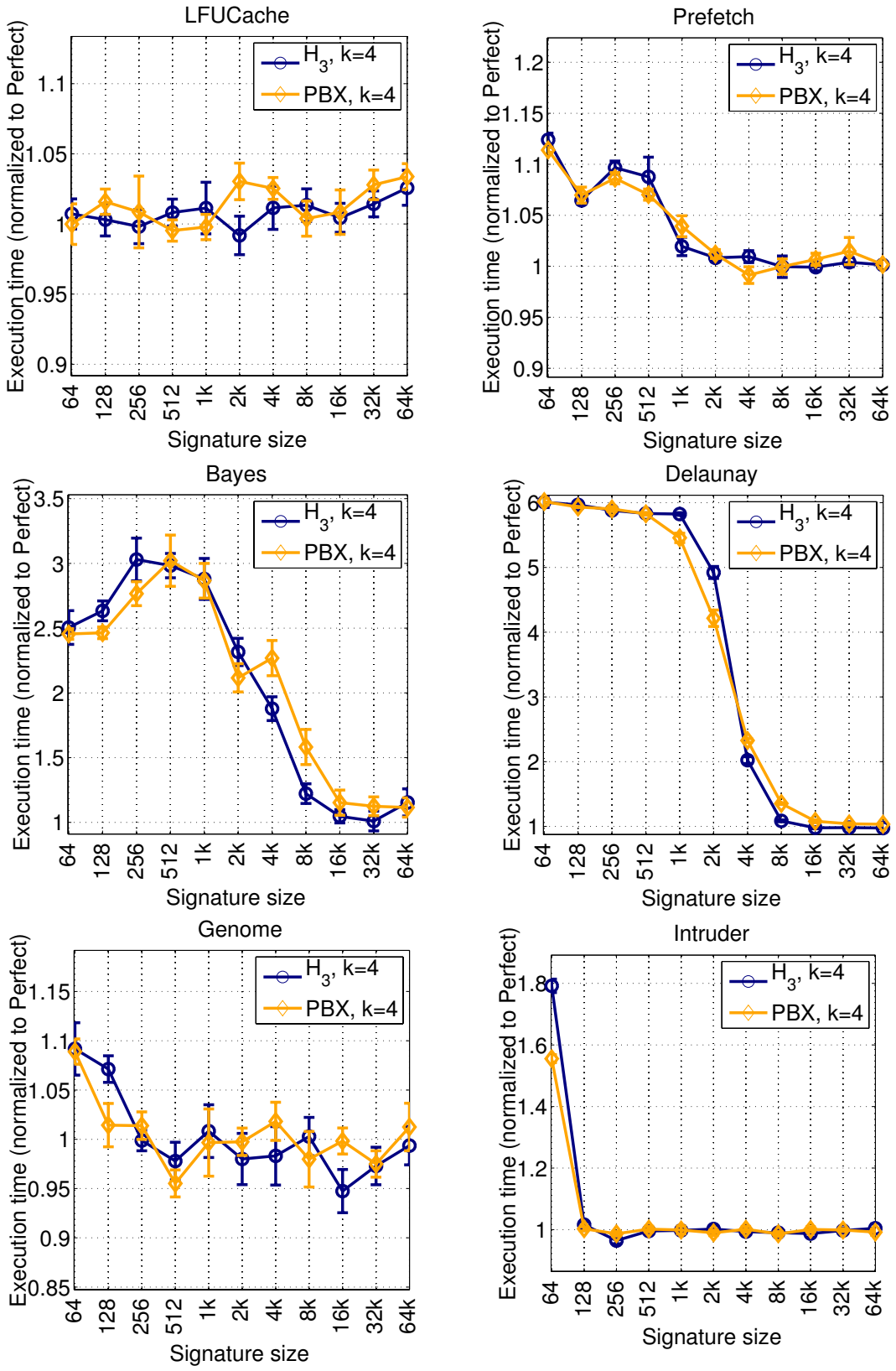


FIGURE 5-12. Normalized execution times of H_3 and PBX signatures with four hashes.

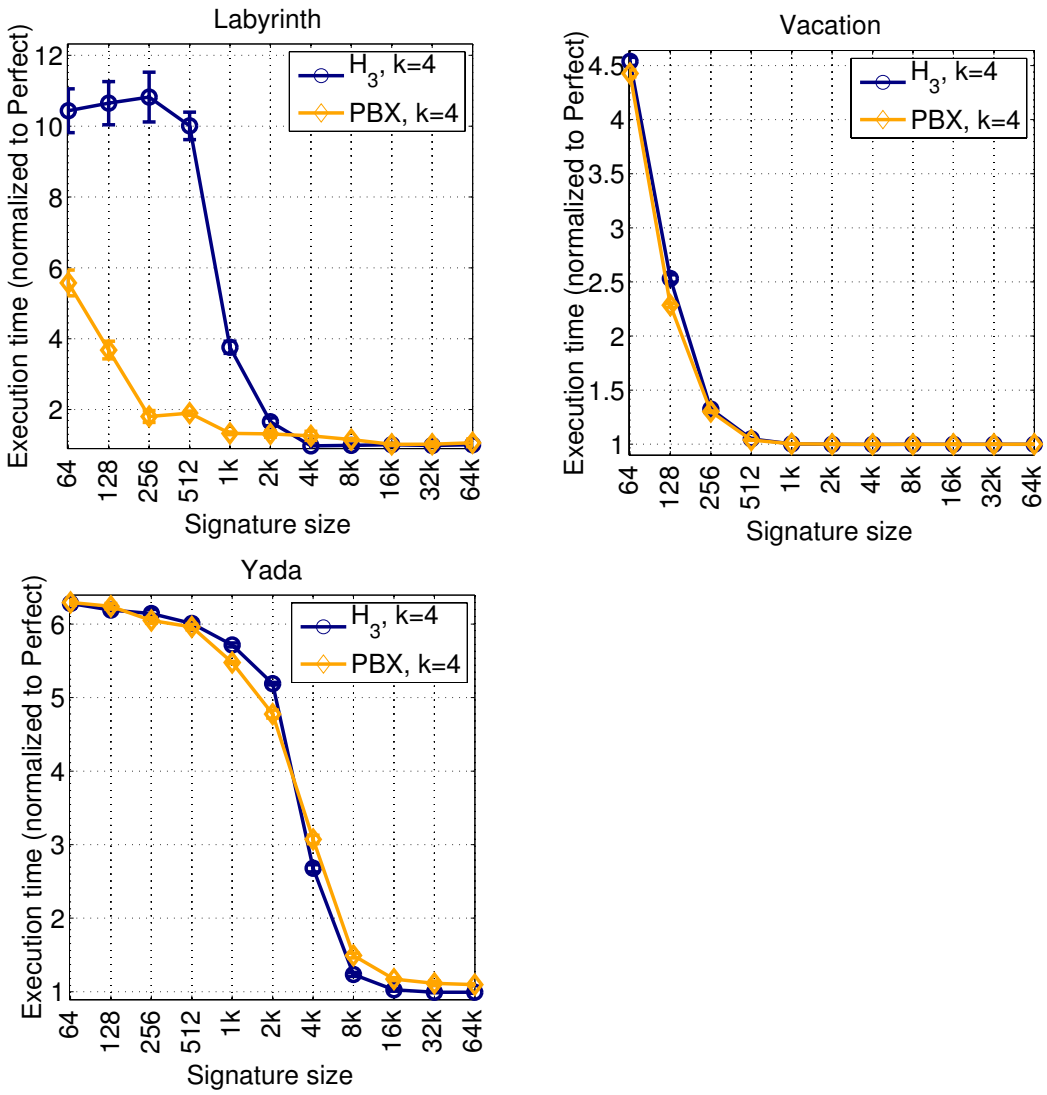


FIGURE 5-12. Normalized execution times of H_3 and PBX signatures with four hashes.

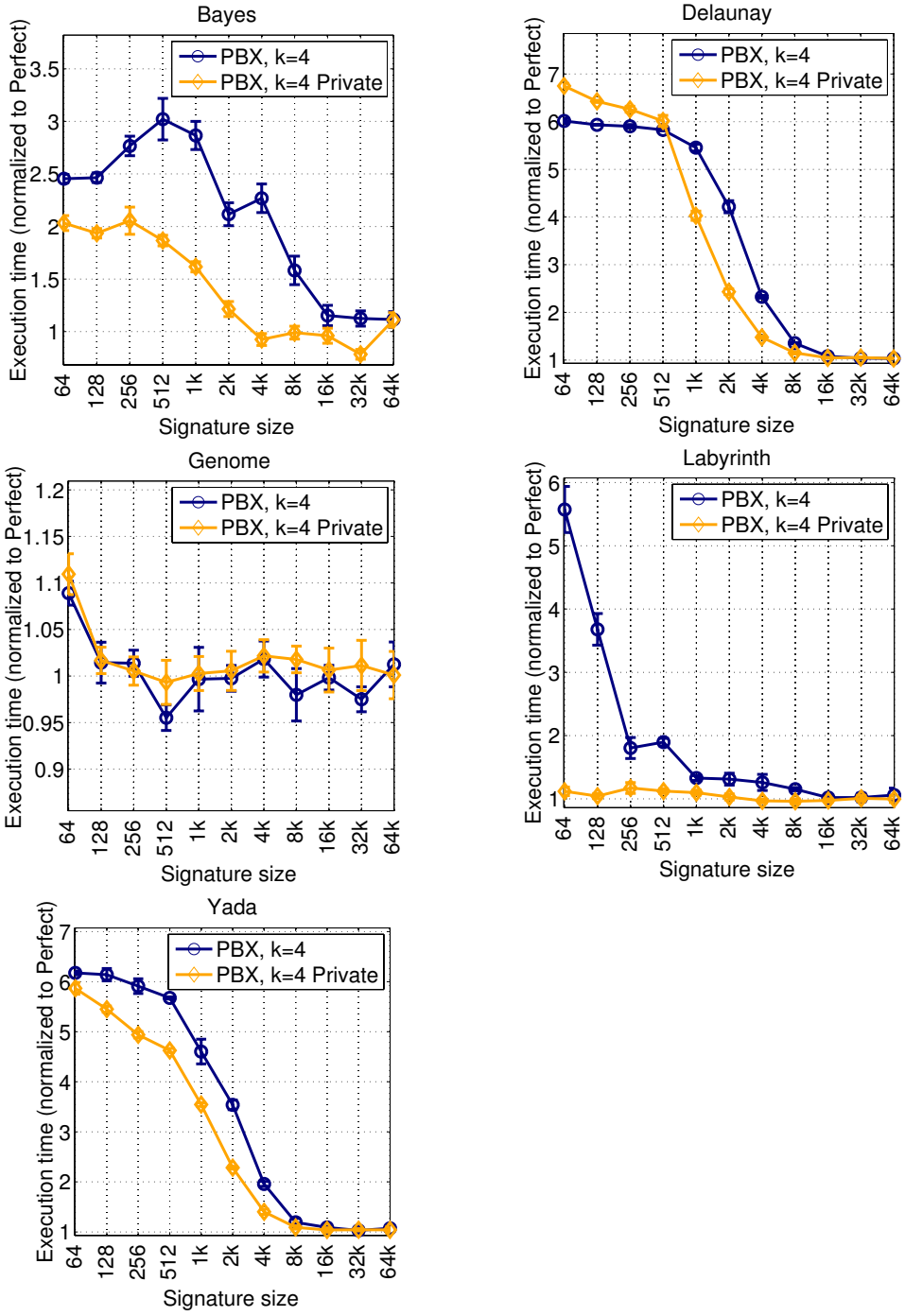


FIGURE 5-13. Normalized execution times before and after privatization, with four hash functions.

5.7.3 Sensitivity to Four Hash Functions

PBX versus H_3 hashing. We first analyze the execution times of PBX and H_3 signatures which use four hash functions. Figure 5-12 shows these results. Overall, for the majority of workloads, the execution times of PBX and H_3 are similar. These results also substantiate our finding for signatures with two hash functions. Our discussion focuses on the exceptional workloads (Raytrace, BTree, Intruder, Labyrinth). In these workloads the H_3 signature sets more bits on average versus the PBX signature, and can lead to worse execution times. In Raytrace, this leads to more non-transactional conflicts and more aborts. In BTree and Intruder this leads to more aborts and more conflicts (both transactional and non-transactional). Finally, in Labyrinth, this difference leads to substantially more transactional stall cycles and more exceptions (both inside and outside of transactions). Examples of these exceptions include clean window and data TLB traps.

Privatization. Figure 5-13 shows the normalized execution times for select STAMP workloads before and after privatization is used. The lines labeled "PBX, k=4 Private" use privatization. Overall, these results are similar to the privatization results for signatures which use two hash functions (see Section 5.6.2). There are workloads which benefit from privatization (Bayes, Delaunay, Labyrinth, Yada). There are also two workloads which sometimes degrade from privatization (Delaunay, Genome). Delaunay, Genome, and Labyrinth's execution times can be explained using the same reasoning that we described in Section 5.6.2.

5.8 Conclusions and Future Work

This chapter developed Notary, a coupling of two enhancements to hardware signature designs to tackle two problems: the high implementation overheads of H_3 hash functions, and false positives due to signature bits set by private memory accesses. We introduced PBX hashing,

which uses bit-entropy to carefully select random bit-fields and performs similar to H_3, but is implemented with much less hardware. Second, we proposed a privatization programming interface giving programmers the ability to declare which accesses cannot cause conflicts, and this can improve program execution time. Finally, Notary can be implemented in signature-based non-TM systems.

We think an interesting future research direction is to explore how to dynamically select the bit-fields for PBX based on phase changes in entropy. This will allow signature-based TM and non-TM systems to have robust performance. Additionally, it would be beneficial for researchers to characterize how often dynamic privatization occurs in TM programs. This should help HTM designers decide whether the HTM support for dynamic privatization is needed.

Chapter 6

TMProf: A Lightweight Profiling Framework for Performance Debugging in Hardware Transactional Memory Systems

Transactional memory (TM) removes many of the correctness problems associated with lock-based programming, but can suffer from hard-to-diagnose performance problems. Moreover, TM programs may execute with different performance on different TM systems. These performance problems stem from the additional critical-section parallelism enabled by TM.

This chapter proposes TMProf as a lightweight profiling framework that provides the foundation for performance debugging hardware TM systems. TMProf focuses on system-level and not application-level performance. The base TMProf implementation consists of a set of hardware performance counters that are implemented in each processor core. These counters track events that break down execution time according to stall cycles, useful transactional cycles, wasted transactional cycles, non-transactional cycles, abort cycles, committing cycles, and other implementation-specific cycles. We also discuss extended implementations of TMProf that profile additional fine-grain information.

We evaluate the utility of TMProf through several case studies involving two different hardware TM systems (HTMs) — Wisconsin's LogTM-SE [127] and an approximation of Stanford's TCC [43] — that use eager and lazy conflict detection, respectively. We vary key design parameters in

each HTM, and use TMProf to gleam key insights into the performance of each HTM running a large suite of TM programs.

We have two main findings:

- TMProf is effective at helping to understand the performance implications of the majority of our HTM parameter changes.
- We find that dynamically tracking a program's critical-path is important, and may warrant hardware support. This support may drive future implementations of TMProf.

6.1 Introduction

TM programming semantics remove many of the correctness problems in lock-based code (e.g., deadlock, composability). However, programming with TM can also create potential performance problems. First, the TM programmer may not have mastered the characteristics of the underlying TM system's implementation. For example, the TM programmer may develop a program using coarse-grained transactions. The program's performance degrades if it is run on top of a TM system that (1) can only support bounded transactions of various sizes (e.g., Herlihy & Moss HTM [48], TCC [43], LTM [7]) or (2) is designed for some transactional behavior that is rare in dynamic executions of the program (e.g., assuming few aborts).

Second, the use of transactions can mask performance problems in programs. It is straightforward to analyze lock-based programs because lock acquires and releases are explicitly defined and locks are associated with memory locations. Lock contention can be profiled by monitoring the memory activity to a lock's memory location. Furthermore, locks serialize the entrance into critical sections. In contrast, TM transactions allow concurrent executions of different dynamic

instances of the same static transaction. This can greatly increase the number of possible memory interleavings within dynamic instances of a static transaction. It is harder to diagnose and treat performance problems in transactions, since both the order and the set of memory requests affect a thread's execution. Performance problems in TM systems can lead to varying performance of different workloads on a particular TM system, or a better performance for a workload on one TM system over another.

A profiling infrastructure is needed to diagnose performance problems, both in prototype and production TM systems. This chapter proposes ***TMProf***, a lightweight hardware profiler that tracks common hardware TM (HTM) events using performance counters. TMProf allows HTM designers the ability to assess designs on prototype or production systems. In addition, TMProf may convey useful information to TM programmers on how their programs perform on specific HTM systems.

TMProf measures events that directly account for overall execution time (stalls, aborts, wasted transactions, useful transactions, commit, non-transactional, and implementation-specific). Furthermore, TMProf breaks down stalls into specific conflict event types (RW, WR, or WW), and also breaks down aborts (into read-dependent or write-dependent). Software can be built on top of TMProf to help program analysis. Furthermore, TMProf can be extended to profile finer-grain (e.g., transaction-level) events. This can help narrow down performance problems within specific regions of program code.

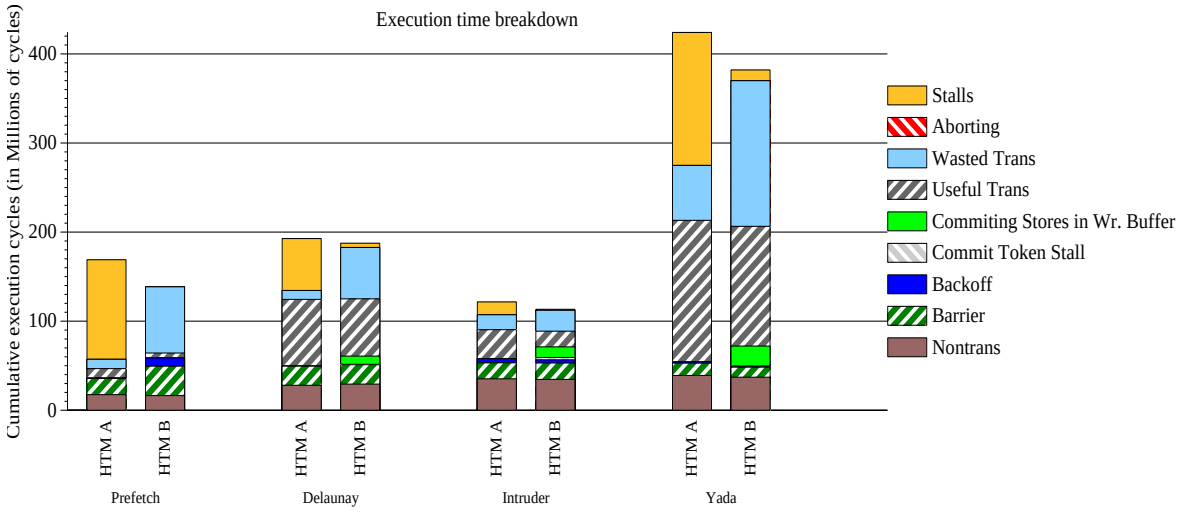


FIGURE 6-1. Execution time breakdowns of select workloads running on two different HTMs

Figure 6-1 motivates the importance of a performance debugging framework for HTMs (we describe the HTM systems and workloads in Section 6.4). This figure shows the execution time results of four TM workloads running on top of two different HTM systems ("HTM A" and "HTM B" in the figure). In general these workloads perform better on HTM B, and TMProf's execution time breakdown reveals why this occurs. For example, "Stalls" (conflicts) are a significant fraction of execution time for HTM A, while "Wasted Trans" (aborted transactions) are more significant for HTM B. Section 6.6.2 details how HTM designers can use TMProf's profiling to explain these performance differences.

We focus on HTM systems. Several production HTMs exist or have been announced [39,112]. We choose performance counters because they are lightweight, do not perturb the original execution, and can capture both coarse-grain and fine-grain events. Similar profiling could be achieved by software simulators or hardware traces, but these methods incur high latency (simulator) or stor-

age (traces) overheads. These overheads increase when examining bigger systems or longer workloads.

We evaluate the utility of TMProf using case studies involving two HTM systems. One is Wisconsin's LogTM-SE [127], and the other is an approximation of Stanford's TCC [43]. Since neither of these HTMs exist in hardware our evaluations use software simulations of TMProf running on the simulated HTMs. However TMProf does not rely on any additional features of software simulation (e.g., deterministic execution). We have two main findings. First, TMProf is effective at helping to understand the performance implications of the majority of our HTM parameter changes. Second, we find that dynamically tracking a program's critical-path is important.

6.2 Background and Related Work

6.2.1 Understanding Conflicts and Aborts

In order to debug the performance of TM systems it is important to first understand some important factors that affect TM system performance. Many of the interesting behaviors in different TM systems come from the frequency and management of transactional conflicts and aborts.

Conflicts. There are three types of conflicts: *Read-Write (RW)*, *Write-Read (WR)*, and *Write-Write (WW)*[1]. All conflict types are uni-directional. RW conflicts arise when a thread has a memory location in its read-set (i.e., locations that are transactionally read), and this memory location is subsequently requested by another thread in exclusive write mode. WR conflicts arise when a thread has a memory location in its write-set (i.e., locations that are transactionally written), and

1. These names correspond to the WAR, RAW, and WAW uniprocessor data dependencies. We use different names to distinguish between conflicts in uniprocessors and transactional conflicts.

the memory location is subsequently requested by another thread in read-only mode. Finally, WW conflicts arise when one thread has a memory location in its write-set, and another thread subsequently requests the memory location in exclusive write mode.

Conflicts can be detected as a memory request occurs (i.e., *eager* [80]), or at a transaction's commit (i.e., *lazy* [80]). TM implementations which detect conflicts lazily (e.g., TCC [43]) may not incur RW conflicts if the reader transaction commits before the writing transaction. At least one proposal from Shiraman et al. [103] combines the two conflict detection modes into a mixed conflict detection policy. The policy of handling conflicts and deciding how to resolve them is called *conflict resolution* (or contention management). There are two main actions a conflict resolution policy can take to resolve conflicts: stall and abort. A stall causes a thread to block until the conflict clears (i.e., the conflicting thread aborts or commits). An abort terminates the current transaction and rolls back the memory and register state to values prior to the transaction.

Conflict resolution policies differ primarily on the frequency of resolution (i.e., at every conflict, after some number of conflicts, or after a deadlock has occurred) and the set of threads involved in the action (i.e., which threads should abort or stall). Scherer et al. [98] propose several different policies and evaluate them in their software transactional memory (STM) framework.

For eager conflict detection, stall and abort actions may be taken for all conflict types. Abort actions are only strictly necessary to maintain transaction serializability (i.e., the appearance of a total order in transactional execution) whenever deadlocks occur. In lazy conflict detection, both actions can only be used for the WW conflict type, as this is the only conflict type in which these actions guarantee serializable transactions. In the remaining conflict types (WR and RW), lazy conflict detection needs to abort threads. Otherwise, transaction serializability is violated.

Aborts. Aborts can occur as the action on a singleton conflict instance or whenever a deadlock between threads occurs (i.e., in eager conflict detection due to stalling requestors). They can be categorized based on conflicts between the set of threads performing the aborts (we name this the aborter) and the set of surviving threads (abortee). There are at least two categories of aborts: *read-dependent* and *write-dependent*. Read-dependent aborts occur when the abortee depends only on the aborter releasing read-isolation (i.e., the conflict is RW). Write-dependent aborts occur when the abortee depends on the aborter releasing write-isolation (i.e., the conflict is WR or WW). Distinguishing between aborts is helpful if the underlying TM system can optimize for read-dependent aborts (e.g., by releasing read-isolation early). If only flat nesting (i.e., no partial rollback or open nesting support [81,73]) is implemented by the TM system, read-isolation can be safely released at the beginning of abort processing. However, support for partial rollback and open nesting in TM systems may need to retain read-isolation for the entire duration of an abort. In order to figure out if it is safe to release read-isolation early, the TM system needs mechanisms to track exactly (at each nesting level) which memory addresses are part of conflicts with individual processors.

6.2.2 Related Work

The initial proposals for profiling TM systems [43,74,29] focus on tracking transaction characteristics such as the read- and write-set sizes and the depth of transaction nesting. They also profile implementation-specific characteristics such as the commit bandwidth used for lazy version management. In addition, there are some breakdowns of useful, conflicting, and commit overhead cycles (without additional breakdowns in conflicts). TMProf focuses on common important sources of overhead across all TM systems. Transaction-level profiling is complementary to our

infrastructure and reveals additional useful information about the program. Furthermore, TMProf can be extended to profile on finer-granularity (like transactions) through additional profiling events.

Chafi et al. [26] propose TAPE (Transactional Application Profiling Environment), a hardware-enabled profiling environment for TCC. They measure specific overheads such as buffer overflows, commit conflicts, workload imbalances, and commit overheads. Their implementation involves hardware support to track and filter this performance data, and a user-readable summary report that details the worst overheads for each type, coupled with the location in the code that causes the overhead. Although TAPE is a comprehensive profiling environment, the overheads they profile are specific to the TCC system and may need to be tailored to profile other TM systems. In contrast, TMProf focuses on the set of basic overheads (such as conflicts and aborts), which fundamentally exist across all TM systems.

Bobba et al. [17] identify several pathological behaviors that can affect both eager and lazy conflict detection systems that use either eager or lazy version management. Their characterization of TM pathologies requires detailed offline analysis of TM execution traces. TMProf enables detailed online breakdowns of common TM system overheads. TMProf does not produce traces but rather event counts and overheads. Although simple, our approach allows TM designers and programmers to gain insight into additional important performance factors that affect both eager and lazy TM systems. Furthermore, some of our results validate their findings, specifically the importance of a good hardware conflict resolution policy and the bottlenecks of serial commit.

Perfumo et al. [88] profile their Haskell STM while running a variety of micro-benchmarks. They measure useful overhead metrics such as read-to-write ratio, abort rate, and cache accesses per

transactional access, and propose to use these metrics to measure the overheads of other STMs. Some of the metrics they propose can also be integrated into TMProf. Finally, Porter et al. [89] present Syncchar, a software-based profiling tool that analyzes the number of independent and conflicting threads in a fixed-size buffer containing the address sets of critical sections. They use these results to predict the speedup of programs if they are run under optimistic concurrency (e.g., transactional memory), as well as to highlight hot-spots in the code which might be highly-contended. The software using TMProf can also perform this analysis, provided the hardware implementation includes support for exposing transactional addresses.

Lastly, it is worthwhile to highlight several success stories using performance counters in performance analysis in academia and industry. First, Singhal et al. [104] describe how they use the performance counters of the SPARCcenter 2000 to build a performance tuning framework using different pieces of software. Second, Keeton et al. [58] use the Pentium Pro's performance counters to characterize the performance implications of online transaction processing workloads on a modern out-of-order processor. Lastly, Zagha et al. [129] show how they perform detailed performance analysis of different metrics using the performance counters of the MIPS R10000 coupled with software tools.

6.3 TMProf

Description. TMProf consists of a set of hardware performance counters for each processor core in HTMs that count cumulative event frequencies and cumulative event overheads. The base TMProf (we name this BaseTMProf) implementation consists of performance counters that count cumulative event frequencies and cumulative event time overheads. BaseTMProf seeks to account for all the cycles in a HTM system through the use of performance counters and various events that can be profiled. More specifically, BaseTMProf breaks down the cumulative cycles across all processor cores in a HTM system into the following components:

Total cycles = stalls + aborts + wasted_trans + useful_trans + committing + nontrans + implementation_specific

TABLE 6-1. Events profiled in base and extended TMProf implementations

	Event	Description	Interval of time for individual event	
			Begins	**Ends**
BaseTMProf, ExtTMProf	Stall	Cycles lost due to waiting for conflict to resolve. Separate overheads for RW, WR, and WW conflicts	Cycle memory request sent to memory system	Cycle conflict resolves
	Aborts	Cycles lost due to processing aborts. Separate overheads for read-dependent and write-dependent aborts	Cycle abort processing initiates	Cycle abort processing completes
	Wasted_trans	Cycles lost executing transactions that eventually abort	Cycle transaction begins	Cycle abort processing initiates
	Useful_trans	Cycles incurred by transactions that commit	Cycle transaction begins	Cycle commit initiates
	Committing	Cycles to commit transactions (e.g., get commit & write permissions)	Cycle commit initiates	Cycle commit completes
	Nontrans	Cycles spent outside of transactions	Cycle after last commit	Cycle transaction begins
	Implementation_ specific	Cycles spent in implementation-specific activities (e.g., backoff after aborts, barriers)	*For backoff:* Cycle abort processing completes	*For backoff:* Cycle transaction retry begins
			For barriers: Cycle processor enters barrier	*For barriers:* Cycle processor exits barrier
	Event	**Description**	**Calculation of overhead**	
ExtTMProf	Aborted_xact_ sizes	Read- and write-set sizes of aborted transaction	If transaction aborts, record its read- and write-set sizes (in number of cache lines)	
	Xact_work_ remaining	Amount of transactional work until transaction commit following a write-set prediction	Record current total transaction size on a prediction. Subtract these from total transaction size on commit. Record differences (in number of cache lines)	

This is a simple model that assumes all events on each processor core can be assigned to a cycle. However this model can be further refined to take into account the event overlaps that occur in modern out-of-order cores. Finally, BaseTMProf represents the most general set of profiling that is common across all HTMs.

The extended TMProf implementation (we name this ExtTMProf) starts with a BaseTMProf implementation and adds fine-grain (e.g., transaction-level) profiling of specific events (such as the size of aborted transactions) in each HTM. Our definition of the additional profiling within ExtTMProf is broad, and it generally consists of fine-grain, HTM-specific profiling that is not captured by BaseTMProf. Other implementations of ExtTMProf for specific HTMs may choose to profile additional fine-grain events and overheads, similar to how modern processors may add performance counters to profile fine-grain events (e.g., cache and pipeline) in different implementations. We show the utility of each TMProf implementation in Section 6.5 and Section 6.6. Table 6-1 summarizes the events and the calculation of an individual event's overheads (in cycles or size) in BaseTMProf and ExtTMProf.

Like existing performance counters, TMProf's counters are virtualized (and multiplexed) using high-level software or operating system support. The high-level software may also combine counter values to calculate other metrics (such as fraction of time lost to wasted transactions) or histograms. In addition, new events and counters can be exposed to newer versions of the high-level software.

We describe implementations of TMProf for two HTM systems: Wisconsin's LogTM-SE [127] and an approximation of Stanford's TCC [43]. However since most of the profiled overheads are common across TM systems, TMProf can be implemented for other HTM systems.

Implementation for LogTM-SE. BaseTMProf starts with the hardware required to implement LogTM-SE, and adds additional meta-data on NACK coherence messages and per-processor tables to record conflicts in order to differentiate abort types. Additional per-processor registers may be needed to help calculate the overheads in Table 6-1. The overhead of conflicts is the elapsed time between the cycle the memory request was sent to the memory subsystem and the cycle the conflict is detected (i.e., the cycle the NACK message arrives). Different conflict types can be distinguished by the responder as they occur by piggy-backing conflict type meta-data on NACK messages (e.g., 3 additional bits to represent all conflict types plus 1 bit to indicate whether the responder is older).

The cycle overhead of aborts is the elapsed time between when abort processing initiates and when it completes. Abort types can be characterized by the aborter's conflicts with other processors. Per-processor conflict tables maintain a mapping of conflict type to the processor causing the conflict. On aborts, the aborter queries its local conflict table for information associated with the abortee. If the only conflict is RW, the abort type is read-dependent.

Whenever a transaction begins the current cycle value is stored in a separate per-processor register. If the transaction aborts, the elapsed time between the saved transaction begin cycle and the start of the abort processing gets accumulated in the counter storing the wasted transactional time. If the transaction commits, the elapsed time between the saved transaction begin cycle and the cycle when commit initiates gets added to the useful transactional time.

In the absence of commit actions [81], there are no committing cycles in LogTM-SE. This is because all commits are zero-cycle, local operations. Backoff is the elapsed time between the completion of the abort and the retry of the aborted transaction. Barrier cycles are the elapsed time

between when a processor enters a barrier and when it exits the barrier. These barriers are program barriers, and may be detected in the HTM through special barrier instructions. Non-transactional cycles are the elapsed time between the cycle after the completion of the last transaction commit and the cycle when a processor begins another transaction. Note that the cycle denoting the completion of the last transaction commit is stored in a separate per-processor register.

ExtTMProf needs additional counters to profile the xact_work_remaining event and overhead. When a write-set prediction (discussed in detail in Section 6.5.1) occurs, the current transaction size (read+write-set size) is recorded in a register, and lazily copied to ExtTMProf's software data structures. Alternatively, a set of *n* registers can be used to store the transaction sizes of the last *n* predictions. These sizes can be exact or approximate (since hardware signatures [23,97,128] are used in LogTM-SE). When the transaction commits, the saved transaction sizes in ExtTMProf's software (or hardware registers) are subtracted from the total transaction size at commit. The differences are then stored in performance counters belonging to this event, and can be processed by ExtTMProf's software to produce histograms. In order to profile the aborted_xact_sizes event and overheads, the read- and write-set sizes of aborted transactions need to be recorded in the associated counters.

Implementation for an approximation of TCC. BaseTMProf starts with the hardware for TCC, and adds performance counters to track event frequencies and overheads. We describe how TCC's BaseTMProf implementation differs from LogTM-SE. Unlike LogTM-SE, conflicts are detected lazily in TCC. This occurs whenever write-sets are broadcast to processors [43], or when commit requests are sent to the directory [25]. In both cases the abort and wasted transactional performance counters for the processors involved in conflicts (e.g., the aborter) need to be updated.

Since aborts do not impede thread progress in TCC, BaseTMProf does not distinguish between read-dependent and write-dependent abort events and overheads.

Stall events (from WR conflicts) can occur between committers and concurrent transactional threads (e.g., a transactional thread wants to read from a line that exists in the write buffer of a committer). When the WR conflict occurs, the reading processor will update its local stall performance counters (both frequency and overhead).

Committing cycles represent the elapsed time between the start of the commit phase (i.e., broadcasting the write-set) and the completion of the commit phase (i.e., all speculative stores have been flushed from the write buffer). BaseTMProf profiles the overhead of obtaining commit permissions separately from the overhead of committing the stores in the write buffer. Our approximation of TCC serializes WW conflicts by stalling conflicting processors, and these conflicts only occur between concurrent committing transactions. We lump these serialization overheads with the overheads of obtaining commit permissions. ExtTMProf's profiling of aborted_xact_size is implemented like in LogTM-SE.

6.4 Platform and Methodology

TMProf is implemented in the Wisconsin GEMS multiprocessor simulator [70]. We build on its TM profiling infrastructure, which breaks down the execution time of HTM systems as described by Bobba et al. [17]. We implement BaseTMProf by refining and validating the existing profiling categories, and adding detailed breakdowns of the frequencies and cycle overheads of each conflict type. We also add support to profile the frequencies and cycle overheads of each abort type. Finally, we implement ExtTMProf by adding the appropriate transaction-level profiling. We assume our performance counters can be accessed and updated with no overheads. We validate

the correctness of our performance counters by comparing the overheads reported by TMProf with the same overheads calculated using offline analysis of detailed TM execution traces.

Our TM workload suite of fifteen benchmarks consists of four micro-benchmarks (BTree, Sparse Matrix, LFUCache, and Prefetch), three applications from the SPLASH-2 suite [124] (Raytrace, Barnes, and Mp3d), seven applications from Stanford's STAMP suite [77] (Delaunay, Genome, Vacation, Intruder, Yada, Labyrinth, and Bayes), and one lock-based DNS server converted to use transactions (BIND 9.4.1 [55]). Chapter 3 describes our workloads as well as their characteristics.

All HTMs are built using a base 16-core CMP system. Due to better scalability [56], BIND runs on top of a 32-core CMP. In addition, we also execute Prefetch with a 32-core CMP because it does not exhibit enough transactional conflicts for interesting behaviors with 16 cores. Chapter 3 describes our system model parameters.

We evaluate TMProf in the context of two HTM systems, Wisconsin's LogTM-SE [127] and an approximation of Stanford's TCC [43]. LogTM-SE uses eager conflict detection and eager version management (i.e., new values are written in place). We abbreviate the eager conflict detection and eager version management system as *EE*. The approximation of TCC uses lazy conflict detection and lazy version management (i.e., new values are written into a private buffer). We abbreviate the lazy conflict detection and lazy version management system as *LL*. To reduce the state space of TM systems and experiments we focus on these two HTM systems and no other combinations of conflict detection and version management. All HTMs use perfect conflict detection [97,23,128] on cache blocks, such that no false conflicts arise. In addition, we assume our HTMs only support flat nesting.

The following sections describe how we use TMProf to performance debug our EE and LL systems. We first describe the characteristics of each HTM system, and then show how we use TMProf to debug the performance of the HTM system. Our goal is to show how TMProf enables better understanding of the performance of each HTM system when its parameters change. Finally, Appendix C presents raw statistics gathered from the simulated TMProf framework detailing execution time, execution time breakdowns, and stall breakdowns for each system configuration.

6.5 Using TMProf to Performance Debug the EE System

6.5.1 Characteristics of the EE System

Idealizations. We focus on the fundamental trade-offs in eager conflict detection. Therefore we choose to idealize the overheads of eager version management. Its main overhead comes from unrolling the undo log for aborts. This action may be on the critical-path for other threads trying to access memory addresses stored in the logs. We idealize log rollback by assuming it takes zero cycles. We evaluate the sensitivity of this idealization in Section 6.7.

We vary two key parameters affecting program execution time in an EE system. These parameters are different conflict resolution policies and the use of write-set prediction. We first describe them in detail and then present our results with performance debugging the EE system with TMProf.

TABLE 6-2. Conflict resolution policies for eager conflict detection

Policy	Description
Base	Requestor stalls until potential deadlock occurs (i.e., requestor has previously stalled an older transaction and is currently stalled by an older transaction).
Timestamp	Older requestor transactions always abort younger transactions. Younger requestor transactions are stalled by older transactions.
Hybrid	Behaves like Base except when RW conflict from older requestor occurs. Hybrid aborts the younger reader in favor of the older writer requestor.

Conflict resolution policies. The three hardware conflict resolution policies we evaluate are named Base, Timestamp, and Hybrid, and they are summarized in Table 6-2. All three policies use timestamps to determine transaction age. In Base, requestors stall on conflicts and repeatedly retry the conflicting memory access. A retry can succeed when the responder commits or aborts its transaction. Otherwise the requestor may abort due to a possible deadlock with another processor. A possible deadlock occurs whenever the requestor has previously stalled an older transaction and is currently stalled by an older transaction.

TABLE 6-3. Requestor's conflict actions under different conflict resolution policies

Operation done first (requestee)	Operation requested second (requestor)	Requestor younger?	Requestor previously stalled older xact during this xact?	Requestor's action (Base)	Requestor's action (Timestamp)	Requestor's action (Hybrid)
W	R	N	Y	Stall	N/A	Stall
W	R	N	N	Stall	Abort requestee	Stall
W	R	Y	Y	Abort	N/A	Abort
W	R	Y	N	Stall	Stall	Stall
R	W	N	Y	Stall	N/A	Abort requestee
R	W	N	N	Stall	Abort requestee	Abort requestee
R	W	Y	Y	Abort	N/A	Abort
R	W	Y	N	Stall	Stall	Stall
W	W	N	Y	Stall	N/A	Stall
W	W	N	N	Stall	Abort requestee	Stall
W	W	Y	Y	Abort	N/A	Abort
W	W	Y	N	Stall	Stall	Stall

The Timestamp policy avoids stalling older requestor transactional threads by always aborting younger conflicting transactional threads. Furthermore, younger requestor transactional threads are stalled by older responding transactional threads.

Finally, the Hybrid policy is a variant of the Base policy. It differs from Base only in its handling of RW conflicts involving older requestors trying to write. In this case the Hybrid policy aborts the younger transactional reader, and the older writer may proceed with its store. This policy is beneficial for threads with latency-critical stores which need to be made public to other readers (e.g., flags, work queue updates). Table 6-3 summarizes the combination of possible conflicts between a requestor and requestee with the requestor's action on each conflict instance. Some of Timestamp's actions may be marked as "N/A" because it is not possible for the requestor to have stalled an older transaction in this policy.

Write-set prediction. Another factor that impacts the performance of eager conflict detection is write-set prediction. Predictors try to avoid costly aborts due to a pattern of read followed by write access to a memory location by multiple threads executing the same static transaction. Write-set predictors detect this pattern and eagerly acquire exclusive access on the first load to memory locations that will be modified later in the transaction. This effectively reduces conflicts (and concurrency) by serializing accesses to those memory locations.

We examine Moore's LOAD-PC predictor [79] under our TMProf framework. The LOAD-PC predictor works as follows. During transaction execution LOAD-PC maintains a mapping of load block addresses with the PC of the corresponding load instructions. If any memory location in which a mapping exists is later modified in the same transaction (by a store instruction), LOAD-PC inserts its load address-to-PC mapping into a predictor table. Subsequent transactional load

instructions might find a matching PC entry in the predictor. A match triggers an upgrade that issues an exclusive memory request for the target memory address associated with that load instruction. We evaluate LOAD-PC with infinite predictive history and zero-cycle access latency. We also use the default threshold (four load/store instances for each load PC) before initiating predictions.

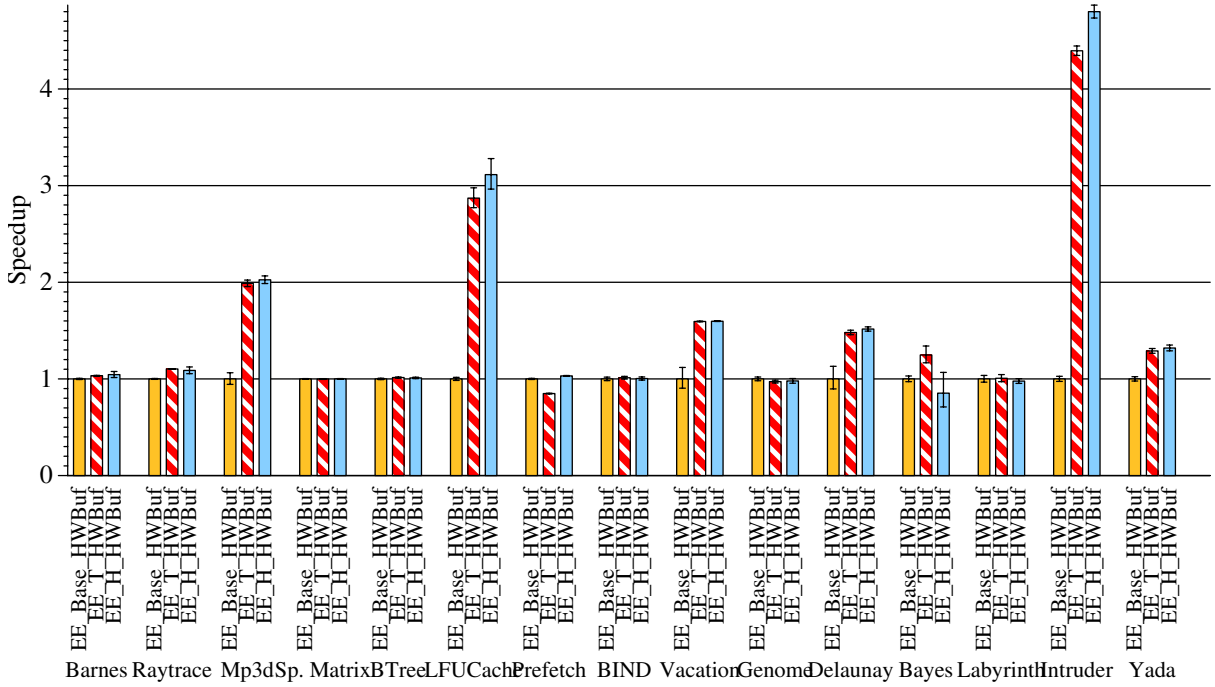


FIGURE 6-2. Program speedups normalized to Base conflict resolution policy

6.5.2 Results

Conflict resolution policies. Figure 6-2 shows the normalized speedups of different conflict resolution policies. The "EE_Base_HWBuf" bar represents Base, "EE_T_HWBuf" represents Timestamp, and "EE_H_HWBuf" represents Hybrid. All speedups are normalized to the Base policy, and higher bars are better.

Three major trends that arise from these workloads. First, Timestamp and Hybrid perform similar or better than Base for the majority of the workloads. Second, Hybrid sometimes performs better than Timestamp (e.g., LFUCache and Intruder). Third, Timestamp can be worse than Base (e.g., Prefetch). Figure 6-2 does not explain why the performance differences occur, so we use the BaseTMProf implementation to help explain the results. We illustrate these results using six rep-

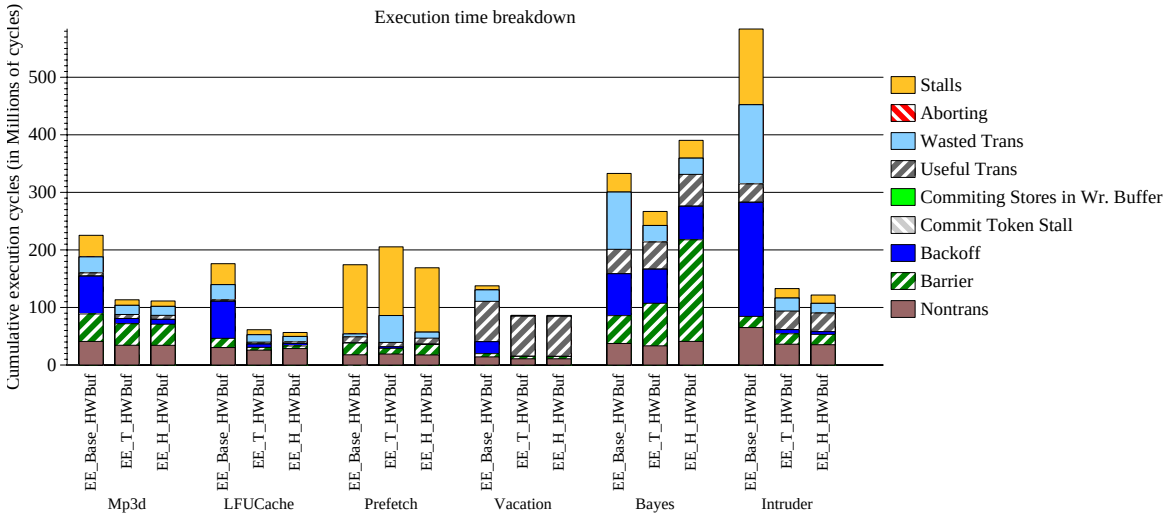


FIGURE 6-3. Execution time breakdown of selected workloads

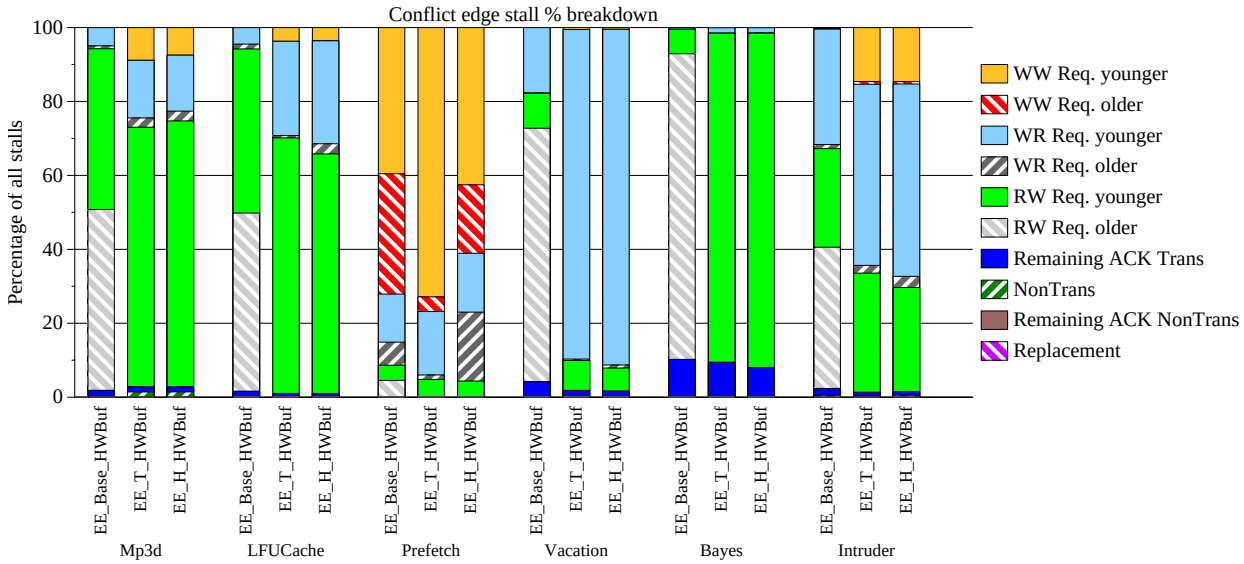


FIGURE 6-4. Stall breakdown by conflict type for selected workloads

resentative workloads (Mp3d, LFUCache, Prefetch, Vacation, Bayes, and Intruder). Our discussions reference the figures illustrating execution time breakdown (Figure 6-3), stall breakdown (Figure 6-4), and abort frequency breakdown (Figure 6-5). Note that due to our idealizations all aborts take zero cycles. Figure 6-5 also shows implementation-specific aborts. “Requestor no-stall” aborts are an optimized version of read-dependent aborts in which the requestor does not have to wait for the responder to release read-isolation. There are also two HTM-specific aborts

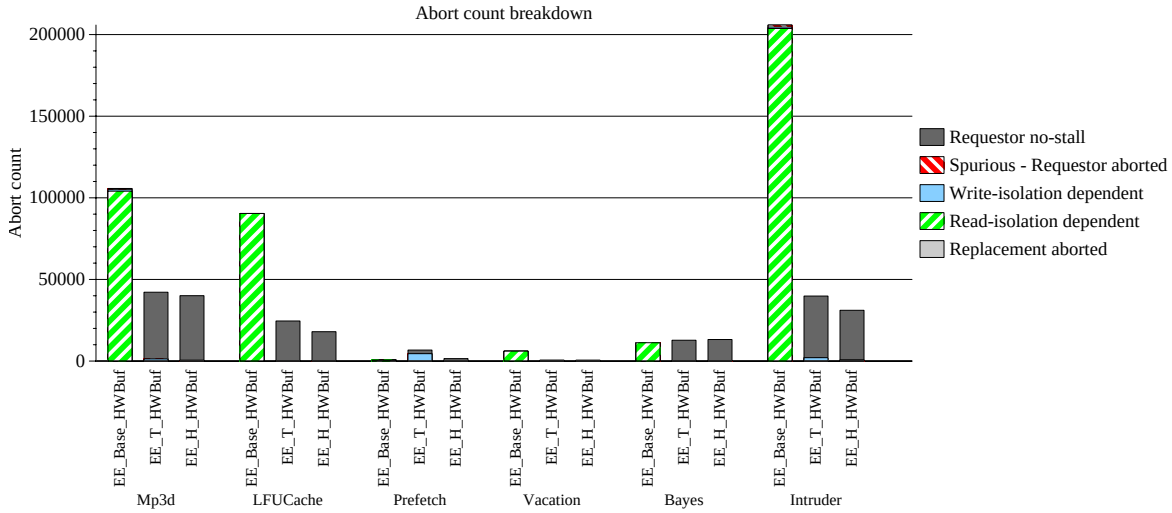


FIGURE 6-5. Abort frequency breakdown for selected workloads

("Spurious" and "Replacement"). These two abort types occur rarely if at all in our workloads. We now analyze our three trends in detail.

First, for Mp3d, LFUCache, Vacation, and Intruder, there is a noticeable reduction in stall cycles as a percentage of total execution time (Figure 6-3). For example, in Mp3d for the Base policy, stalls account for approximately 15% of execution time cycles. This decreases to less than 10% of execution time when Timestamp or Hybrid is used. BaseTMProf is also able to quantify the policy's impact on various conflict types and correlate its impact on overall execution time. As Figure 6-4 shows, the reduction in stall overhead for Mp3d comes from the elimination of all RW stall cycles to older requestors. This is because both Timestamp and Hybrid are policies which optimize away these RW stalls. For the Base policy, RW conflicts account for approximately 50% of all stall cycles. Finally, as BaseTMProf shows, the majority of aborts in the Base policy is read-dependent. Timestamp and Hybrid have fewer overall aborts, and most of those read-dependent aborts do not stall the requestor. BaseTMProf can similarly be used to explain the better performance of Timestamp and Hybrid for LFUCache, Vacation, and Intruder.

Second, in LFUCache and Intruder the Hybrid policy out-performs Timestamp. Again, BaseTMProf reveals why this occurs. In LFUCache, the Timestamp policy yields more stalls and more transactional work that get aborted (the "Wasted Trans" category). Intruder also has a higher percentage of aborted transactional work in Timestamp than Hybrid. For these workloads it is more beneficial to stall whenever possible rather than to aggressively abort younger transactions to resolve conflicts.

Third, Timestamp can sometimes perform worse than the Base policy. BaseTMProf reveals that Prefetch exhibits this behavior (see Figure 6-3 and Figure 6-4). Timestamp results in a large

increase in aborted transactional work compared to Base. About 30% of all conflicts in the Base policy are WW to older transactions, and Timestamp always aborts the younger transaction in these conflicts. The conflicts occur midway through a transaction, and stalling the older transaction is more beneficial than throwing away the younger transaction's work.

Implication 1: BaseTMProf reveals the performance implications of different hardware conflict resolution policies, by drilling down into the execution time breakdowns and coupling this with detailed stall and abort breakdowns. BaseTMProf enables fast, lightweight, online performance debugging for HTM designers.

Write-set prediction. Figure 6-6 shows the normalized speedups of the eager system with and without write-set prediction enabled. All systems use Hybrid conflict resolution. The results for write-set prediction are the rightmost bars in each set labeled "EE_HP_HWBuf". Results are normalized to the eager system that does not use write-set prediction, so higher bars are better.

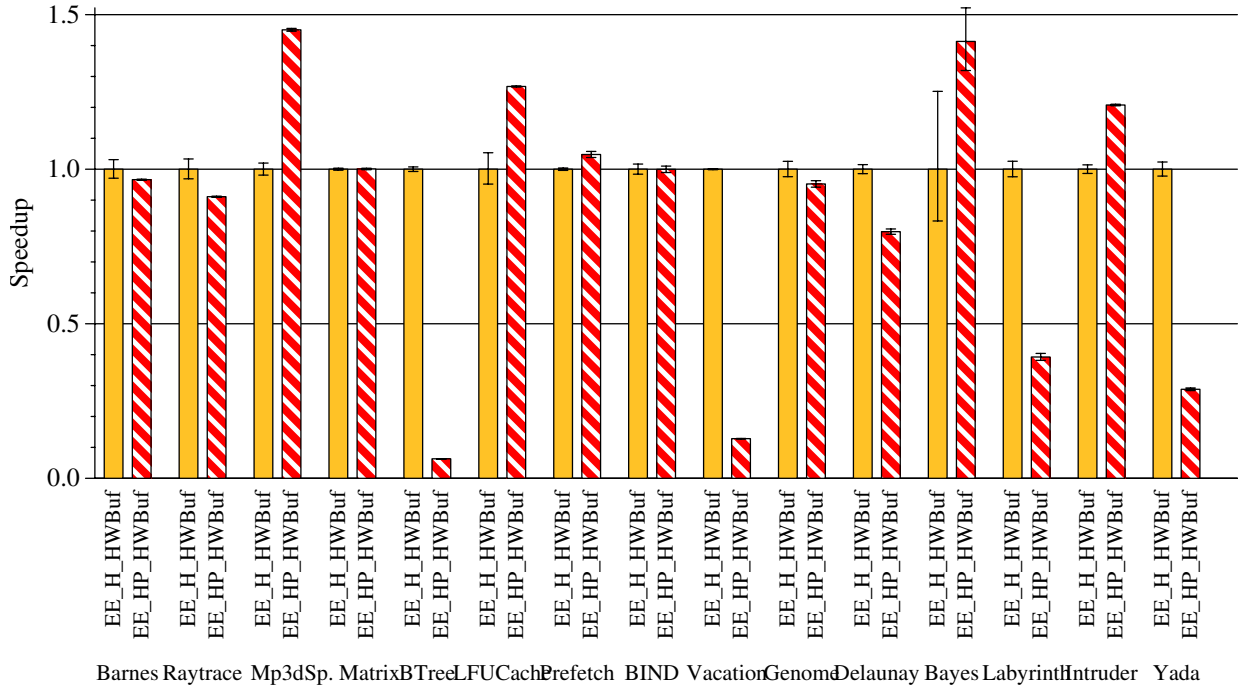


FIGURE 6-6. Speedups of systems with write-set prediction normalized to systems without write-set prediction

Overall, the results indicate workloads fall into three categories. First there are workloads which benefit from write-set prediction. These include Mp3d, LFUCache, Prefetch, Bayes, and Intruder. Second, there are workloads which degrade from write-set prediction. These include Barnes, Ray-trace, BTree, Vacation, Genome, Delaunay, Labyrinth, and Yada. The remaining workloads show little difference with write-set prediction.

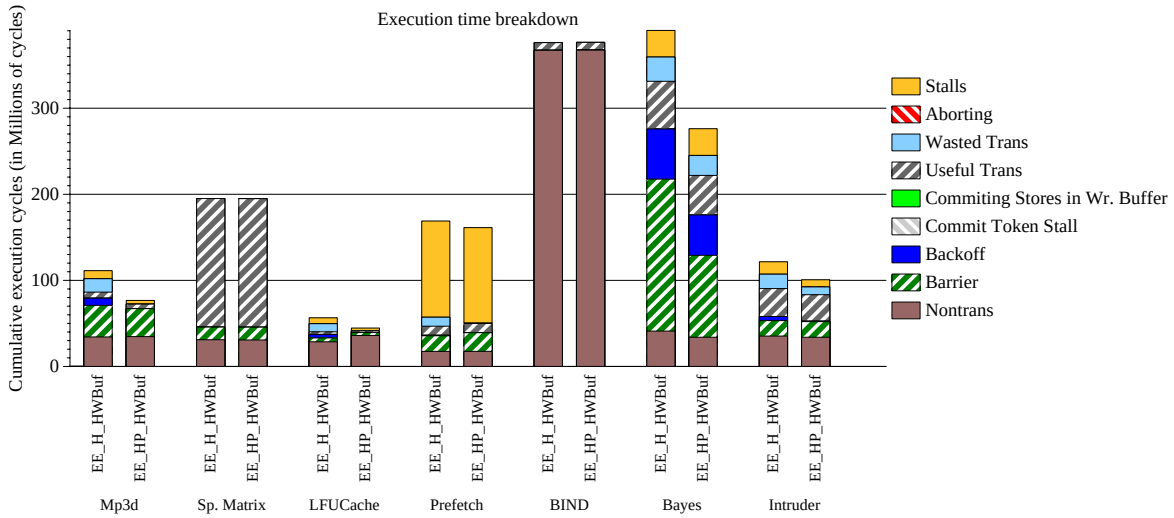


FIGURE 6-7. Execution time breakdown of workloads that have similar or better execution times from write-set prediction

For brevity, we use ExtTMProf to examine the execution time breakdown (in cycles) of the first group (workloads that benefit) and third group (workloads that show no difference) together. These results are shown in Figure 6-7. Overall, for the majority of workloads there is a decrease in the amount of stall and wasted transactional cycles in the overall execution time. This is because

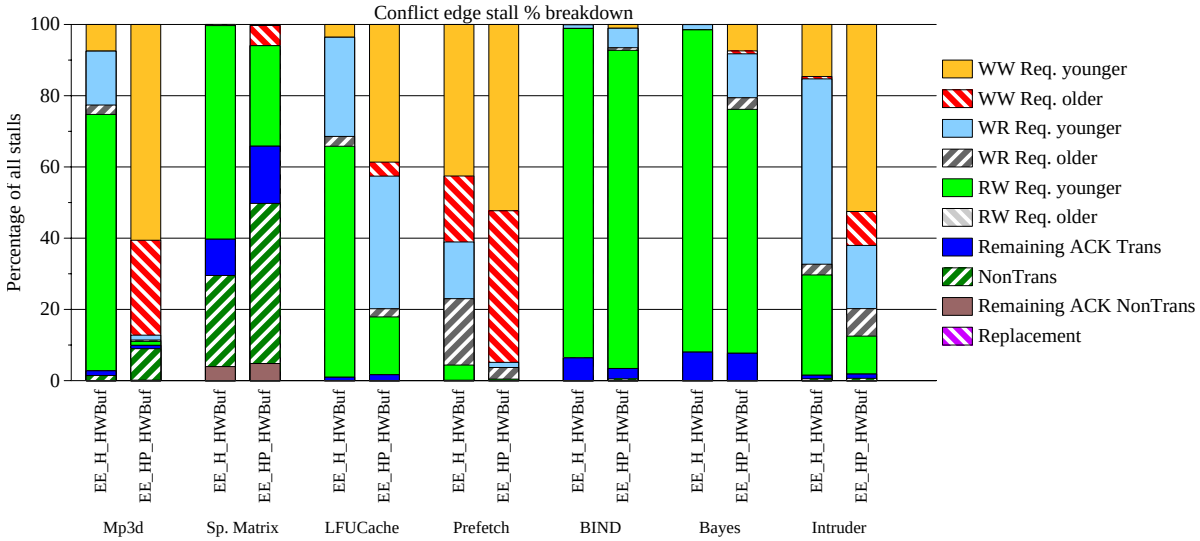


FIGURE 6-8. Stall breakdowns for workloads that have similar or better execution times from write-set prediction

write-set prediction avoids costly aborts (including wasted transactional work) by serializing accesses to conflicting memory addresses. Fewer aborts result in fewer re-executions of transactions and thus fewer wasted transactional cycles. Furthermore, serialization of conflicts reduces contention in the system and therefore leads to fewer total stall cycles.

We examine ExtTMProf's stall category breakdowns in more detail, and the results are shown in Figure 6-8. For the Mp3d, LFUCache, Bayes, and Intruder workloads, the percentage of stalls due to RW requestor younger decreases. Write-set prediction causes this by allowing latency-critical write requests from younger transactions to proceed without stalling. Furthermore, predictions from the younger writers stall other readers (increasing the fraction of WR conflicts), or cause additional exclusive permission requests from multiple processors (increasing the fraction of WW conflicts). For Prefetch, the WR older requestor stall category is a smaller fraction of overall stalls. Fortunately, write-set prediction allows the older reader to gain exclusive access to the block before younger writers, which is crucial to improving execution time in this workload. A side effect of prediction is that there are now more exclusive permission requests in the system

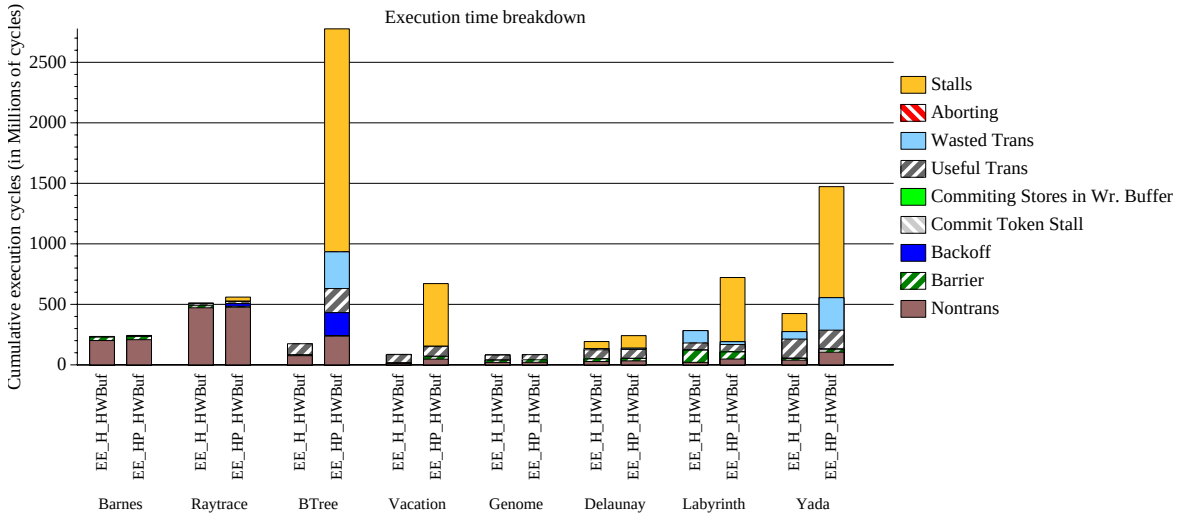


FIGURE 6-9. Execution time breakdown of workloads which degrade from write-set prediction

(original plus additional ones caused by predictions), and this increases the fraction of conflicts due to WW conflicts. However, this effect is less critical to system performance than the positive effect of streamlining the execution of older transactions at the expense of younger transactions.

Next, we focus on the workloads which degrade when write-set prediction is used. Again, we use the ExtTMProf implementation for our detailed analysis. Figure 6-9 illustrates the execution time breakdown for these workloads, and Figure 6-10 shows detailed stall breakdowns. Write-set prediction increases the number of stalls for the majority of workloads. For Raytrace, BTree, and Labyrinth, write-set prediction increases the percentage of WR requestor older stalls. Write-set prediction increases the probability that younger readers gain exclusive permission to cache blocks, and this is bad for performance when these younger readers lock out latency-critical older transactions. This phenomenon is illustrated in Figure 6-10, because prior to prediction younger transactions encounter RW or WR conflicts, and after prediction younger transactions prevail in conflicts while a larger fraction of older transactions get stalled (by WW or WR conflicts). Additionally, some of these workloads access a significant amount of transactional cache lines after the

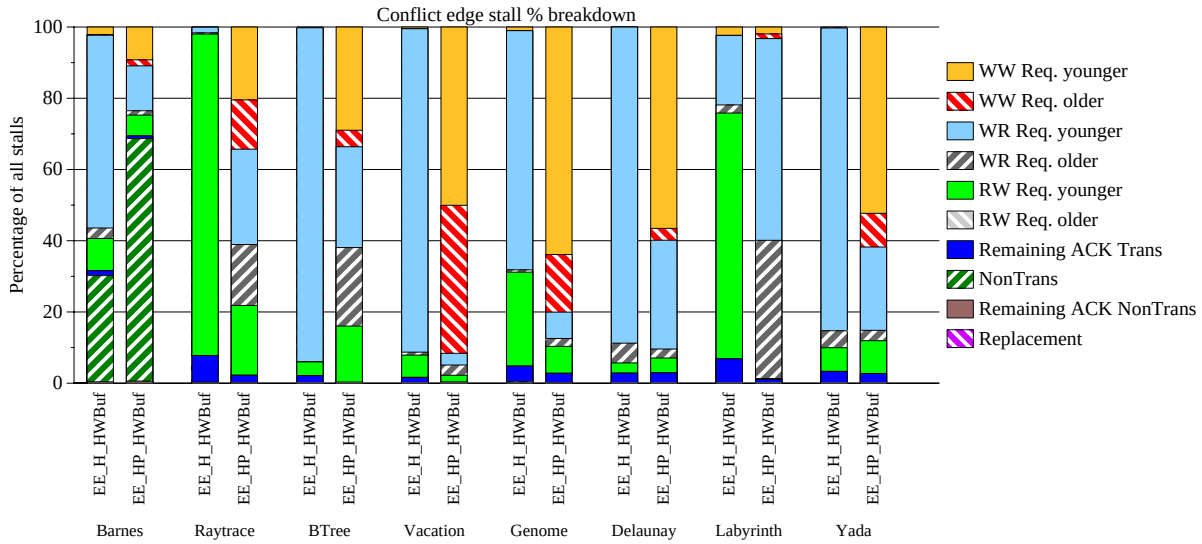


FIGURE 6-10. Stall breakdown of workloads which degrade from write-set prediction

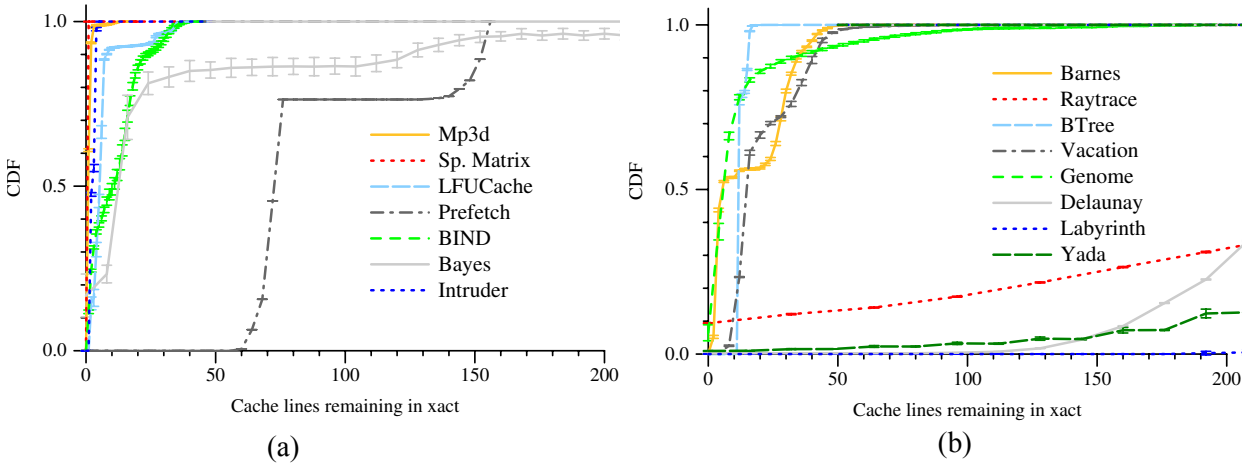


FIGURE 6-11. Cumulative distribution functions for workloads with execution times that (a) are similar or better and (b) worse with write-set prediction

prediction is made. This increases the duration in which readers and writers are stalled behind the exclusive owner of a block.

We use the fine-grain profiling of ExtTMProf to measure the remaining amount of transactional work (in number of transactional read and write cache lines) until commit after a write-set prediction is made. From these measurements we plot the cumulative distribution function (CDF) for these results across all predictions made in each workload.

These CDF results are plotted in Figure 6-11, with (a) representing workloads with similar or better execution times and (b) representing workloads that perform worse with write-set prediction. A CDF line that has a slowly-increasing slope implies a higher probability that more transactional work needs to be performed after a prediction is made. Additional work may reduce concurrency to other readers and writers.

ExtTMProf shows predictions have very different effects in the two groups. In (a), predictions enable latency-critical writers to serialize potentially conflicting readers behind them, and improve execution times. In (b), readers are more latency-critical and predictions hurt reader con-

currency. There are four workloads in (b) that have a significant amount of transactional work remaining after a prediction is made. These workloads are Raytrace, Delaunay, Labyrinth, and Yada. A CDF value of 0.5 occurs around 350 cache lines for Raytrace, 200 cache lines for Delaunay, 400 cache lines for Labyrinth, and 300 cache lines for Yada. These CDFs are the worst, and concurrency degrades in the TM system.

Implication 2: ExtTMProf helps HTM designers understand when write-set predictions are helpful for EE systems. The fine-grain profiling in ExtTMProf clearly distinguishes the characteristics in which write-set predictions degrade HTM performance. An offline profiling technique (e.g., traces) cannot easily determine whether write-set predictions will be beneficial for a workload, since the full effects of conflicts and aborts are dependent on dynamic behavior.

6.6 Using TMProf to Performance Debug the LL System

6.6.1 Characteristics of the LL System

Idealizations. Our system with lazy conflict detection uses lazy version management, in which all speculative transactional stores are buffered in an infinite-sized write buffer that can be accessed (and flushed) in zero cycles. We also idealize commit. For serial commit (i.e., only one thread can commit at a time) we assume no overhead is incurred to grab the global commit token. For parallel commit we assume no commit token latencies and also no overheads to check against the write-sets of all concurrent committers.

We examine two key characteristics affecting program execution time in a LL system. These characteristics are parallel commit and the prefetching effects of lazy conflict detection. We first

describe them in detail and then present our results with performance debugging the LL system with TMProf.

Parallel commit. Our default LL system uses a single global commit token for commit arbitration. A token holder blocks other potentially non-conflicting threads from committing. This policy can severely hamper TM system scalability as the number of processor cores on a CMP increases. In contrast, eager conflict detection allows concurrent, local transaction commits.

In order to evaluate these commit token overheads we also implement parallel commit for our lazy system. This is done by doing the following. First, the parallel commit system has as many commit tokens as processors in the system. Second, a token requestor checks the write-sets of all current commit token holders for conflicts against its own read- and write-sets. If there are conflicts, the requestor stalls until a token holder finishes its commit. Then it retries its commit request (and re-stalls if necessary). We idealize both of these overheads.

Our idealized parallel commit algorithm is more aggressive than the directory-based algorithms from Chafi et al. [25] and Pugsley et al. [90]. Because both schemes operate at the granularity of directories and not at individual cache blocks they may unnecessarily restrict commit concurrency. For example, both schemes will treat two distinct transactional stores whose block addresses map to the same directory as conflicting during commit, while our scheme will not. Our implementation allows us to explore the upper-bound performance of parallel commits.

Prefetching effects of lazy conflict detection. A transaction that executes with lazy conflict detection and reaches its commit point has the opportunity to issue all memory requests within a transaction at least once (even if conflicts occur early in the transaction). Load requests are issued to the memory system, and store updates are re-directed to a separate buffer. If a transaction

instance is aborted, subsequent retries that follow similar execution paths are likely to access those lines (with cache hits instead of misses). Thus the same program executing under lazy conflict detection could finish faster than with eager conflict detection. In contrast, eager conflict detection may abort transactions before the end of a transaction is reached. These early aborts limit eager conflict detection's prefetching effects.

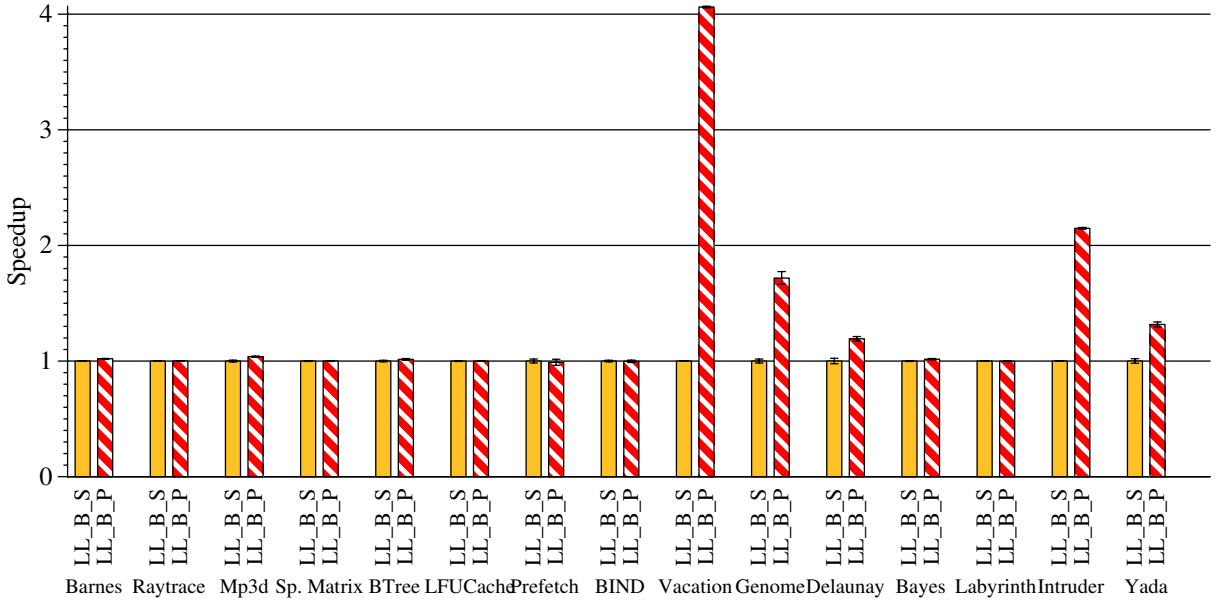


FIGURE 6-12. Normalized speedups using serial and parallel commits

6.6.2 Results

Parallel commit. Figure 6-12 shows the normalized execution time results of the LL system using serial and parallel commits. Serial commit are the bars labeled "LL_B_S", and parallel commit are the bars denoted by "LL_B_P". There are five workloads which benefit significantly from parallel commit (Vacation, Genome, Delaunay, Intruder, and Yada). The speedups of parallel

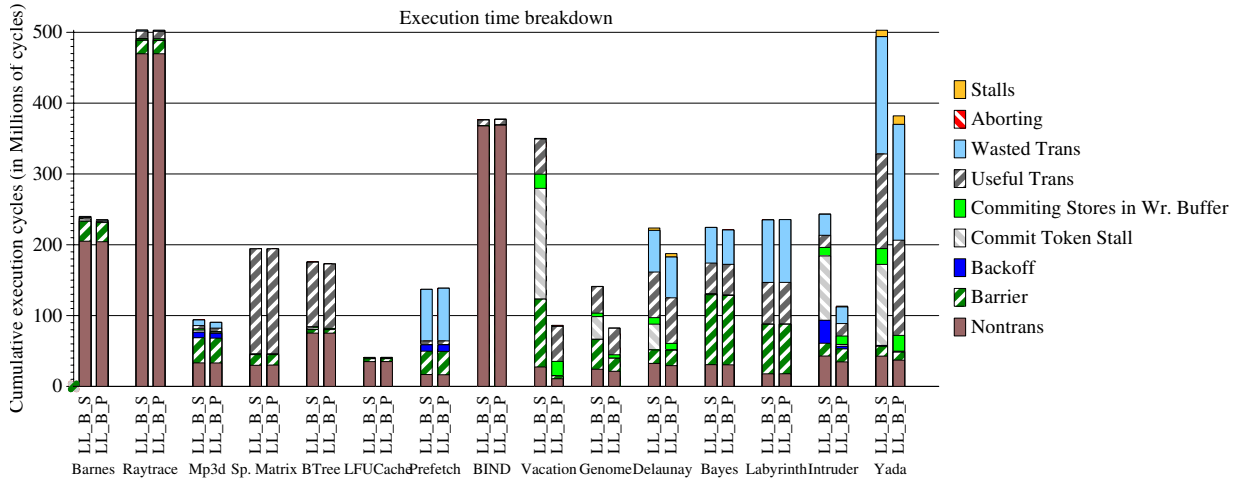


FIGURE 6-13. Execution time breakdown for serial and parallel commits

commit range from 30% to more than four times over serial commit. This figure does not reveal the whole story, so we use BaseTMProf to profile the execution time breakdowns. These results are shown in Figure 6-13.

The most noticeable result is that for these five workloads commit token overhead ("Commit Token Stall" in the figure) account for a significant fraction of the serial commit system's execution time. For example, in Vacation, commit token stalls account for over 40% of all execution time (the largest of all workloads). This workload also has the largest speedup when parallel commit is used. Assuming these workloads scale to larger number of cores (>16 cores), these commit overheads would undoubtedly consume an even larger fraction of execution time. Finally, BaseTMProf's detailed stall breakdowns (Figure 6-14) reveal that threads are not stalled by WW conflicts prior to their commit point. Most of the conflicts are WR conflicts caused by committers stalling other threads attempting to load addresses belonging to the committer's write-set. Note that Prefetch has no stall cycles.

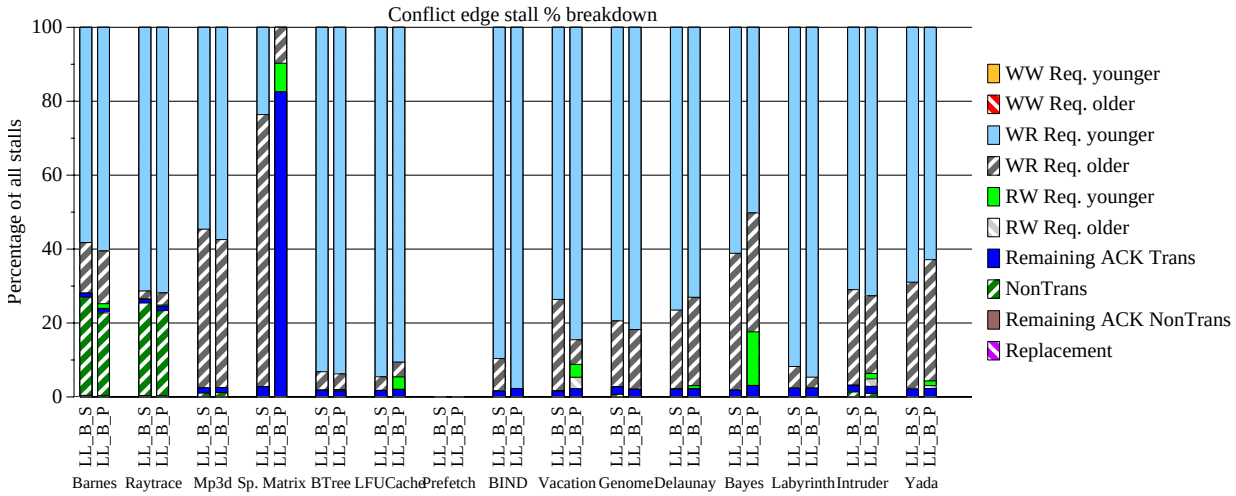


FIGURE 6-14. Stall breakdown of systems which use serial and parallel commits

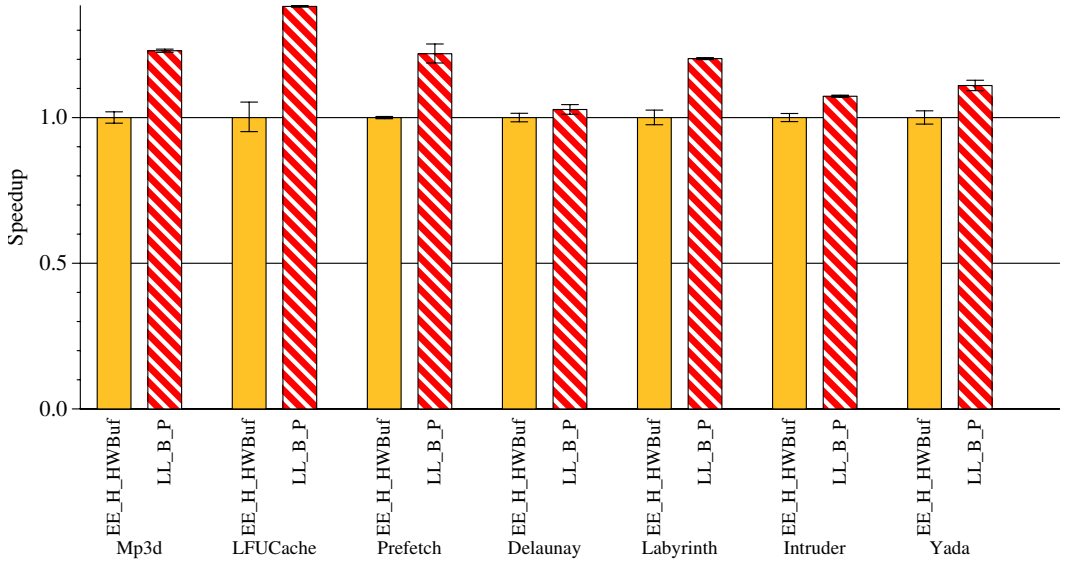


FIGURE 6-15. Normalized speedup of selected workloads with eager and lazy conflict detection

Implication 3: BaseTMProf reveals when serial commit is a large overhead, and why parallel commit helps. This profiling can help HTM designers decide whether future implementations of the HTM may need to implement the additional hardware complexity of parallel commits.

Prefetching effects of lazy conflict detection. We illustrate the prefetching effect by comparing a workload's execution times under EE (with Hybrid conflict resolution) and under LL (with par-

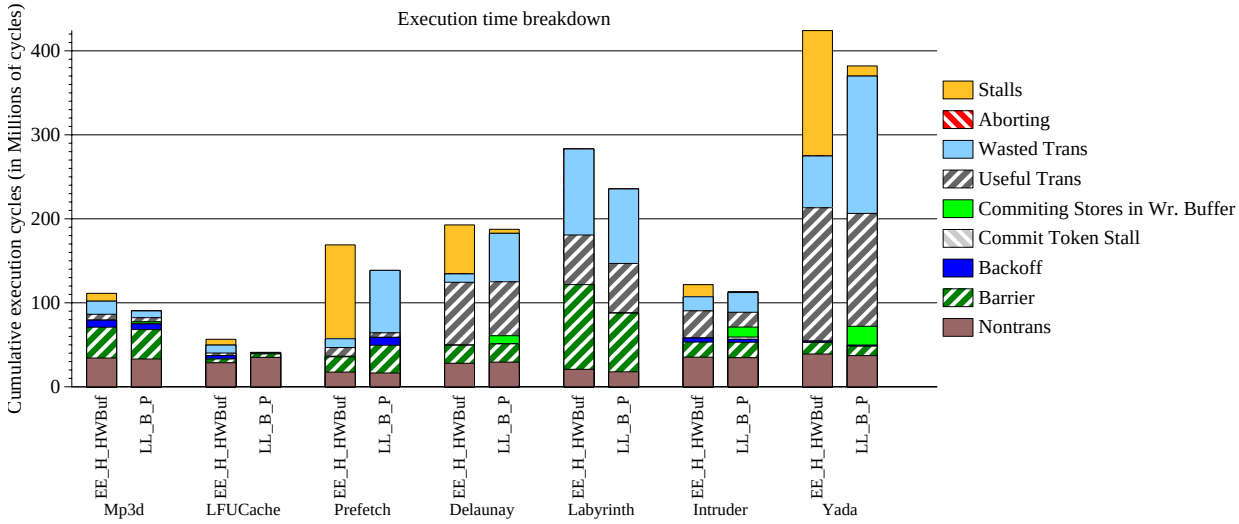


FIGURE 6-16. Execution time breakdown of eager and lazy conflict detection

allel commits). Of all workloads, seven perform better with LL (Mp3d, LFUCache, Prefetch, Delaunay, Labyrinth, Intruder, and Yada). Figure 6-15 shows their normalized speedups, normalized to the EE system (higher is better). Overall, lazy conflict detection leads to speedups of 4-40%. ExtTMProf provides the necessary fine-grain profiling that explains these performance differences.

Figure 6-16 shows the execution time breakdown for these workloads. Our discussion focuses on the category labeled “Useful Trans.” This category represents the total time spent inside committed transactions. If average memory latencies are similar then this category should have the same overhead value across both HTM systems. This is because both systems execute approximately the same number of committed transactions. However, the figure shows this overhead is smaller in the LL system for the majority of workloads. ExtTMProf provides the additional profiling needed to explain these results.

Specifically, we leverage ExtTMProf’s ability to profile the read- and write-set sizes of aborted transactions. Because conflicts are detected at commit in the LL system, there is a higher proba-

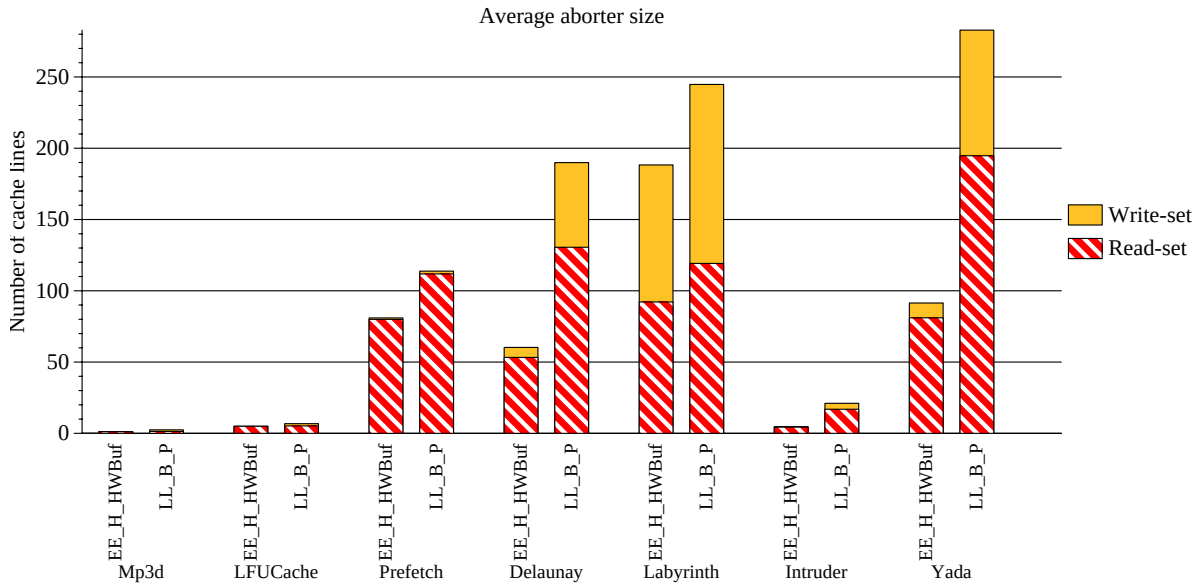


FIGURE 6-17. Read- and write-set sizes of aborted transactions

bility that transactions issue more memory requests (and proceed farther) than transactions executed in the EE system. Thus, the sizes of aborted transactions should be greater in the lazy system. Figure 6-17 confirms our hypothesis. For example, in Delaunay the transactions in the EE system access fewer than 60 cache lines in the read-set, compared to about 130 cache lines in the LL system. For some workloads (Delaunay, Labyrinth, Intruder, and Yada), it is evident that the conflicts in EE occur early in the transaction.

Implication 4: ExtTMProf explains why some workloads execute faster on the LL system than on the EE system. If ExtTMProf is implemented in prototype HTM EE and LL systems, it may influence the design decision to implement the LL system for production. As a HTM programmer, ExtTMProf may be used to provide feedback on why particular workloads may execute faster on the LL system rather than the EE system.

6.7 Future Directions for TMProf

In previous sections we show how BaseTMProf or ExtTMProf implementations can be used to performance debug our EE and LL HTM systems. However, we found at least one case which cannot be explained by our proposed counter-based TMProf implementations. We discovered this while trying to understand the performance of non-idealized abort rollback for our EE system. Figure 6-18 shows a subset of these execution time results. The results for non-idealized software rollback are the bars labeled "EE_H", and lower bars are better. The most surprising result is that for these workloads software unroll performs *better* than with no unroll overheads. This is contrary to our expectations of worse performance with software unroll. Other workloads (Vacation, Intruder) behave as expected, with worse performance with software unroll. We hypothesize that the latency of software abort may actually reduce contention, and help the remainder of the program.

This diversity of these results reveals a weakness in counter-based profiling infrastructures such as TMProf. It is well known that many factors affect uniprocessor performance [38], and these

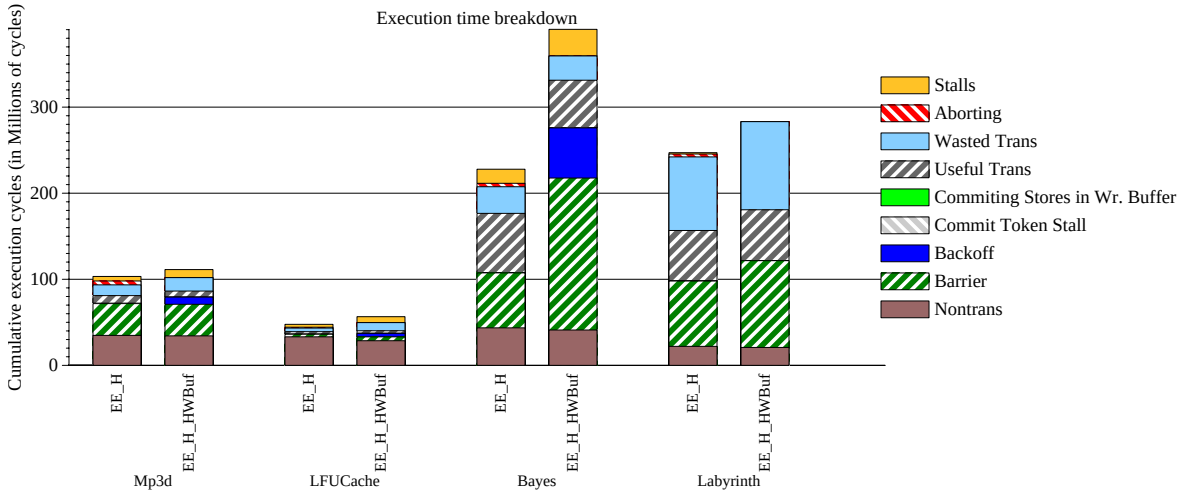


FIGURE 6-18. Execution time breakdown of workloads for which software rollback performs better than no rollback overheads

factors multiply in multiprocessor systems. The control flow in multi-threaded programs is dependent on the timing of events, which can change from one execution of a program to another. These changes can alter the critical-path of the program, which is the longest control-flow in time that determines the program's execution time. There have been numerous proposals to compute this critical-path in multi-threaded programs ([66,125,49]). For example, Yang et al. [125] propose a software-based critical-path analysis based on the Program Activity Graph. However, these proposals are all characterized by offline analysis in software. We feel some hardware support may be warranted in future implementations of TMProf. This will enable faster, detailed online profiling (and feedback) that will allow deeper understanding of TM systems and workloads.

Furthermore, our current implementations of TMProf do not provide sufficient high-level profiling feedback to the TM programmer. The TM programmer would benefit most from profiling that can be tied back to specific lines in the source code. Some of this profiling can be achieved if TMProf is extended to associate events and their overheads to specific addresses and program counters.

6.8 Conclusions

We present TMProf, a hardware profiling infrastructure consisting of a set of per-processor performance counters that measure the cumulative frequencies and overheads of events in a HTM system. The BaseTMProf implementation accounts for all execution cycles of a TM system. BaseTMProf counts cycles based on common events across all HTM systems and also implementation-specific events such as backoff after aborts. Furthermore, BaseTMProf provides detailed breakdowns of stalls and aborts. We also describe ExtTMProf, an extended implementation of BaseTMProf that provides HTM-specific fine-grain profiling of the transaction sizes of aborted

transactions, and the remaining transactional work after a write-set prediction. Similar to how performance counters in a processor's pipeline adapt to changes in the pipeline's implementation, changes in an HTM's implementation may require low-level profiling be added or removed in ExtTMProf.

We evaluate TMProf using case studies involving two different HTM systems (LogTM-SE and an approximation of TCC). We have two major findings. First, TMProf is effective at helping the HTM designer understand the performance of the majority of the parameter changes in the two HTMs. Second, we find that dynamically tracking the critical-path of TM programs may reveal important profiling information, and hardware support may be warranted. This support may drive future implementations of TMProf.

Chapter 7

Extensions to Signatures

Signature false conflicts can cause significant performance degradations when transactional memory (TM) systems use signatures for conflict detection. As Chapter 5 details, removing private stack references from signatures or using a heap-based privatization interface can help alleviate some performance penalties. In this chapter we discuss six extensions to signatures which can be used in conjunction with earlier privatization mechanisms to further reduce signature false conflicts.

The extensions stem from the following motivations. First, if the programmer has high-level knowledge about the program that may be helpful for conflict detection, then an opportunity exists for signatures to exploit that knowledge. Second, programs exhibit spatial locality. Signature hashes can be tailored to the amount of spatial locality within programs. The goal is to reduce the probability of false conflicts for objects accessed from distinct memory regions, at the expense of a slight increase in false conflicts from objects accessed within the same memory region. For example, for a spatially local region of size R, objects within R may experience additional signature false conflicts with a locality-aware hash function. However, objects outside of R may actually experience fewer signature false conflicts. This decrease may lead to fewer total false conflicts overall. Third, signature hashes may be altered to avoid false conflicts arising from bad hash values. The following list summarizes the six extensions and describes how signatures can use them. We also relate these extensions to a high-level model of signatures, which are composed

of inputs to signatures, hash functions, and hash values. Our descriptions indicate how each extension modifies one or more of the signature's components.

- Static transaction identifier (XID) independence — The programmer/runtime annotates the set of static transactions that cannot ever conflict, based on knowledge of the program. Each static transaction is assigned to a unique XID. Signatures can use this information passed down by software by specializing when the conflict detection mechanisms are invoked for a specific XID. This reduces the frequency of conflict detection checks and thereby the probability of false conflicts. XIDs are provided as additional inputs to signatures, and additional hash functions are needed to check whether the input XIDs match the set of conflicting XIDs stored at each processor. The XIDs increase the total amount of information contained in the signature's input values.
- Object identifiers (IDs) — The programmer declares a set of high-level identifiers denoting which data structures or parts of data structures are accessed inside of static transactions. Each processor can locally store the set of object IDs accessed within transactions and check them against another set of object IDs from a different processor. If these object ID sets are checked along with signature lookups during conflict detection, signature false conflicts could be reduced if object ID sets rarely overlap. Object IDs provide another set of signature inputs, and additional hash functions are needed to check whether object IDs overlap with the set of object IDs accessed within a transaction. If object IDs are encoded using addresses, there may be some overlap between addresses from memory requests and object IDs. Thus, object IDs may not add much additional information to the input values.

- Spatial locality using static signatures — Many programs display the characteristic of spatial locality, where cache lines that are close together in physical memory are likely to be accessed together within some interval of time. This concept applies to the read- and write-sets of transactions, where its cache lines can be captured by memory regions of various sizes (e.g., 64B and larger). The same concept can be applied to address hashing for signature insertion operations, where hashing on larger granularities reduces the number of regions encompassing the inserted addresses. Fewer regions directly translate to fewer signature bits set, and reduces the utilization of the signature. Lower utilization reduces the probability of false conflicts on subsequent lookup operations. This extension applies the signature hashing granularity statically using a fixed granularity. With respect to our high-level model of signatures, this extension alters the signature hash functions, but does not increase information contained in the signature's input values.
- Spatial locality using dynamic signatures — Dynamic signatures adapt to the best hashing granularity for each workload. Instead of using static hashing granularities for signatures as described previously, dynamic signatures are used. Dynamic signatures maintain a set of signatures, each hashing on a different granularity, along with hit counters associated with each signature. We describe an algorithm that dynamically adapts signatures to query the signature that hashes with the best predicted granularity. With respect to our high-level model of signatures, this extension adds additional signatures (each consisting of its own hash functions and hash values). The information contained in the signature inputs remain unchanged.

- Coarse-fine hashing — This is an alternative implementation of optimizing for spatial locality. Signature hashes operate on coarse or fine address bits. Fine bits consists of some number of low-order address bits (e.g., multiple cache blocks), and coarse bits are the remaining high-order bits (e.g., denoting memory regions). Dedicating signature hashes to operate on each region reduces the probability of false conflicts due to full signatures (by hashing on coarse bits) or false conflicts due to sparse signatures (by hashing on fine bits). This extension changes the signature hash functions, but does not increase information contained in the input values.
- Dynamic re-hashing — Re-hashing aims to minimize the negative performance effects of bad signature hashing. Bad signature hashing can cause false conflicts, leading to more conflicts and aborts. Signature re-hashing can be performed by rotating the input address bits before using them as inputs to the hash functions. Therefore re-hashing can potentially turn false conflicts into *transient* false conflicts. This extension conceptually changes the signature hash functions by changing the order of bits from the input values. It does not increase the information contained in the input values.

We idealize the hardware implementation overheads of these extensions in order to estimate the upper-bound performance of these extensions. Even with these idealizations we find that most of these extensions do not lead to significant performance improvements for signatures. Also, we assume our hardware transactional memory (HTM) system uses Hybrid conflict resolution with "magic" waiting. This means that a processor with an aborted transaction (the aborter) retries the transaction exactly after the processor that aborted it (the abortee) commits or aborts its own transaction. Magic waiting idealizes the backoff overheads after aborts and enables the best exe-

cution times for the STAMP workloads. Except for BIND and Prefetch, our TM workloads (described in Chapter 3) run on top of a 16-core CMP. BIND and Prefetch run on top of a 32-core CMP. All system parameters are the same as in Chapters 5-6, and their details are described in Chapter 3.

Overall, we find that certain workloads may be more affected by scheduling effects (e.g., from the operating system or from different conflict orders) than others. This limits the effectiveness of our extensions on those workload's performance. However, other workloads are less affected by scheduling effects, and thus show performance improvements. Our analyses highlight these workloads and offer insights into their performance.

We now describe each extension and its results on signature performance in turn. We normalize all execution time results to perfect signatures, and Appendix D presents the raw execution time cycles for perfect signatures for all our workloads.

7.1 Static Transaction Identifier (XID) Independence

Description. The main idea behind static transaction identifier (XID) independence is to enable the programmer to declare the set of static transactions which cannot possibly conflict in dynamic executions of the program. Each static transaction is associated with a unique XID. The TM system can store this information, and then use it to filter out instances in which signature conflict detection can be turned off for certain XIDs. At a high level, the XID information is stored and checked independently of signatures, and could be implemented using a programmable lookup table. Every memory request is associated with the XID from the requesting thread (with a default XID for non-transactional requests), and requestors participate in conflict detection only if its XID is pre-determined to conflict with the responder's XID.

Current TM interfaces export language-level constructs such as `tx_begin` and `tx_end`, or `atomic{}`, and programmers use these to declare language-level transactions. Based on our own experiences with programming and debugging TM programs, we also find it useful to associate XIDs with each statically-defined transaction.

We define XID values to be non-negative integers, and each static program transaction gets assigned to a unique XID in a TM program. XIDs not only facilitate debugging but can also be used to convey the notion of *XID independence* — two XIDs are independent if they are statically found to never conflict in any dynamic execution of the TM program. Otherwise we say the XIDs are dependent. Additionally, we define dynamic instances of the same static XID as dependent. This definition makes sense, because the declaration of a single static transaction signifies the programmer's intent that dynamic instances of that static transaction might conflict with each other.

Distinguishing between dependent and independent XIDs is helpful for conflict detection because independent XIDs implies conflict detection does not have to be performed between those XIDs. In eager conflict detection, responders of memory requests for independent XIDs can bypass the full conflict detection algorithm (in hardware or software). In lazy conflict detection, commit overhead (both latency and bandwidth) can be reduced if XID independence can filter the set of processors (or directories) that need to be involved in the commit algorithm.

XID independence can also directly reduce the additional conflict and abort overheads associated with imprecise conflict detection (e.g., with signatures). For example, assume two threads are executing XID 1 and XID 2. XID 1 always reads memory address A and XID 2 always writes memory address B. If the read- and write-sets are precisely recorded on memory addresses then

there is no conflict with XID 1 and XID 2. However, signatures which summarize the read- and write-sets of XID 1 and XID 2 might signal a false conflict due to the hashing scheme mapping addresses A and B to signature bits. False conflicts can be avoided if signatures have the knowledge that these XIDs are independent and avoid conflict detection between XIDs 1 and 2.

There are at least three hurdles that impede the utility of XID independence. First, a programmer may not know or be able to categorize all XIDs as dependent or independent. This is particularly true for large TM programs or one that incorporates any closed-source library code. Second, even with complete programmer knowledge the underlying TM system may unknowingly violate XID independence. For example, false sharing on memory addresses can occur between independent XIDs, making them actually dependent in dynamic executions. Third, mistakes by the programmer in categorizing XIDs can lead to program errors. For example, marking a set of XIDs as independent when they are actually dependent can lead to unexpected results or program crashes.

Although these restrictions may ultimately outweigh the performance benefits of XID independence, we feel compelled to explore its performance under ideal conditions (i.e., none of the above restrictions are violated). This allows us to focus on the implementation and performance costs separate from the semantic issues.

Implementation. We idealize the implementation overheads of XID independence. This consists mainly of extra meta-data on coherence request messages (to indicate the XID of the requestor), and the programmable hardware state to track which XIDs conflict. The latter can be initialized by using a program-level interface that lets the programmer pass this information to the hardware. These idealizations allow us to focus on the best possible performance improvements of XID independence and examine the implementation overheads separately.

TABLE 7-1. Description of how XID independence applies to select STAMP workloads

Workload	Description of XID independence
Bayes	XID 0 is independent of all other XIDs because of a barrier in the code and non-conflicting operations. XIDs 1,2,4,6,11,12,13,14 access a shared tasklist, but are independent of remaining XIDs which operate on other data structures.
Delaunay	XID 0 and XID 2 are dependent, but are independent of the other XIDs. XIDs 0,2 operate on a shared worklist. XID 1 refines a cavity, and is independent of XID 3, which updates global variables.
Genome	All the XIDs are independent. This is because the XIDs either operate on different data structures (e.g., constructEntries and startHashToConstructEntryTable) or reside in different code regions due to barriers.
Intruder	XID 0 is independent of XIDs 1,2. XID 0 retrieves a data packet, and XIDs 1 and 2 perform the decoding processes.
Labyrinth	All the XIDs are independent. This is due to non-conflicting sets of operations. XID 0 dequeues work from a shared queue, XID 1 computes a route locally and copies it back to the global grid, and XID 2 inserts the route's coordinates into a shared pathVectorList.
Vacation	XID 0 inserts a customer into a tree. This XID is independent of XIDs that query and modify other trees (e.g., car, flight, room).
Yada	XIDs 0 and 4 are dependent, but are independent of the other XIDs. XIDs 0,4 operate on a shared work heap. XIDs 1,2,3 are dependent, but are independent of other XIDs. XIDs 1,2,3 operate on a shared mesh. XID 5 operates on global variables and is independent of other XIDs.

Methodology. We focus on seven STAMP workloads (Bayes, Delaunay, Genome, Intruder, Labyrinth, Vacation, and Yada). These workloads have multiple static transactions and therefore multiple XIDs (at least three static transactions and three XIDs) and spend a significant fraction of time inside transactions. In order to simulate the programmer efforts involved in denoting XID independence, we manually examine each STAMP workload. For each XID we find the set of XIDs that could possibly conflict with it. This process was fairly straightforward, and required a couple of iterations to capture all the sufficient dependencies. Table 7-1 summarizes the XID indepen-

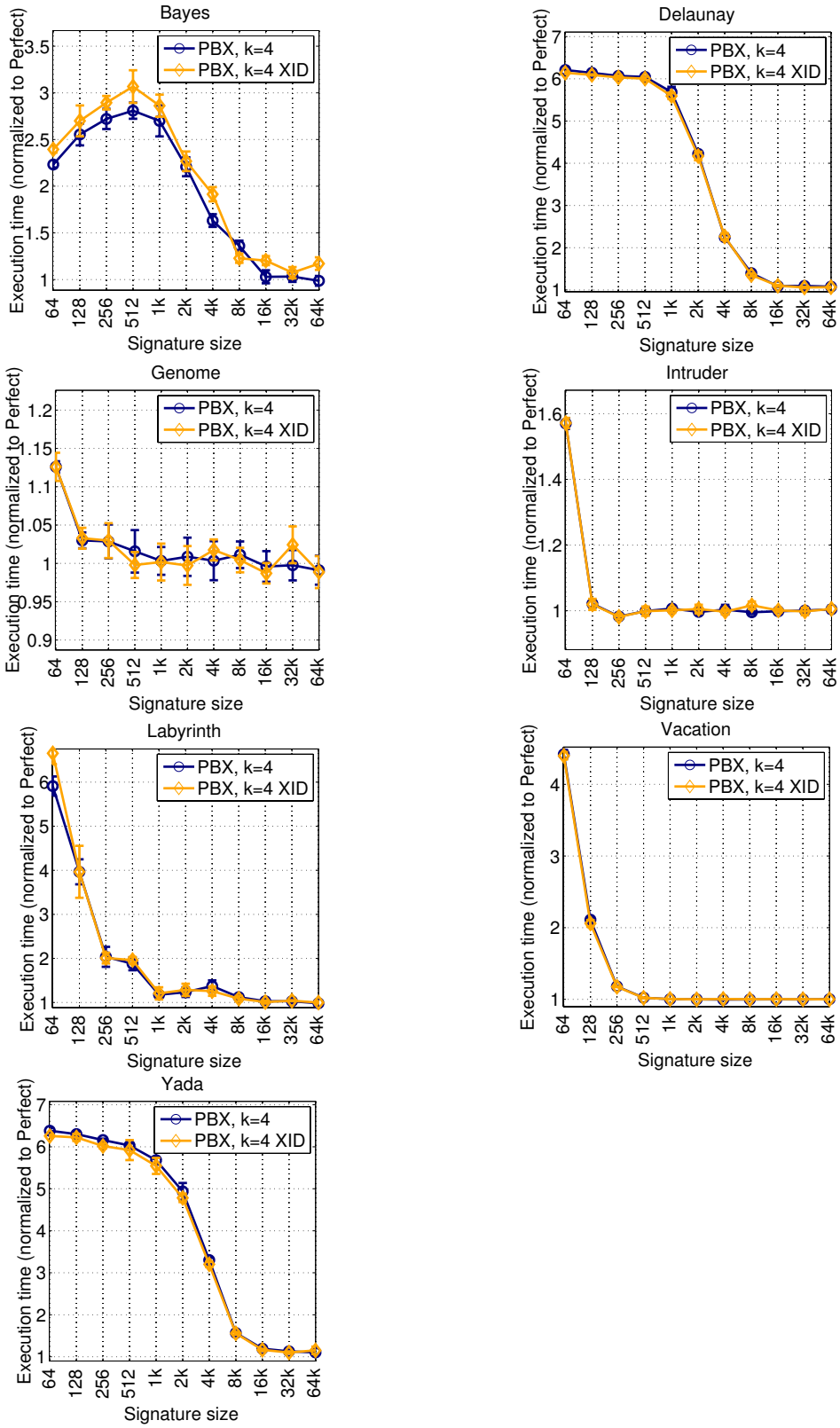


FIGURE 7-1. Normalized execution times before and after XID independence is used

dence that was found for the selected STAMP workloads. All experiments are performed with LogTM-SE with Perfect and imperfect signatures.

Results. Figure 7-1 shows the normalized execution times of our seven workloads with and without XID independence. All execution times are normalized to the execution times of Perfect signatures, and lower values are better. As the results show, XID independence does not significantly improve execution times (and is sometimes worse for Bayes and Labyrinth). In Bayes, this is due to increased cycles inside barriers and waiting to re-execute aborted transactions. In Labyrinth, for 64b signatures, XID independence increases the cycles spent waiting to re-execute aborted transactions. The behaviors for these two workloads arise due to non-deterministic thread interleavings that perturb the order of memory requests. This effect can lead to different execution paths in both the user and supervisor code.

Conflict profiling in the original workload indicates that most of the stall cycles are between XIDs that actually dynamically conflict with each other. Furthermore, since this is a static optimization, the programmer needs to conservatively mark XIDs as dependent even though they might only be dependent with small probability. However, it is worthwhile to note that our workloads do not contain a large number of XIDs. Bayes has the largest number of static transactions (and XIDs), with thirteen in the program. Future transactional programs may contain a much larger number of static transactions (and XIDs), in which case it may be prudent to revisit the performance implications of XID independence.

In summary, XID independence does not yield significant execution time benefits for any of our seven STAMP workloads. This is maybe due to our conservative estimation of potential con-

flicts. Furthermore, semantic costs associated with correct usage may ultimately outweigh any performance benefits.

We now describe an alternative optimization (object IDs) that can leverage the dynamic interleavings (and conflicts) of transactions, and can detect fine-grain conflicts. Unlike XID independence, there are instances in which object IDs help performance. But there are also hardware costs and semantic hurdles associated with this optimization.

7.2 Object Identifiers (IDs)

Description. The main idea behind object identifiers (IDs) is to use a smaller set of addresses (named the object IDs) than the read- or write-set to help signatures reduce the probability of false conflicts. The set of object IDs is declared by the programmer and passed to the signature hardware. We call our scheme object IDs because the scheme uses identifiers to denote shared objects that are accessed within transactions. This idea is similar to how language-level objects are used for software transactional memory systems, and we discuss how our ideas are related in Section 7.6.3.

Our definition of an object is a loose one, and is specific to the data structures that are accessed within transactions. For example, if a workload can access multiple red-black trees within transactions, we can assign separate object IDs to uniquely identify each tree. These IDs can be as simple as the addresses of the roots of the trees. If a program uses a hash table, object IDs can be assigned to denote the address of each hash bucket.

We first present the high-level picture of object ID usage, and follow with code examples denoting correct and incorrect usages. Our object ID scheme works as follows. The first step is for

the programmer to use a program-level interface to statically declare the shared objects that can be accessed within each static transaction. The invariant is that object IDs need to be declared before those objects are accessed in the transaction. Furthermore, an object ID declaration and the first access to the corresponding object in the transaction must not be re-ordered by the compiler or the hardware. If re-ordering occurs, the hardware will not be aware of which shared object has been accessed. Conservatively, these declarations can be inserted at the earliest point in the transaction in which the object is valid (i.e., during initialization or when it contains valid state). This is the most time-intensive step (and could be aided by compiler heuristics) as it requires knowledge about the program's algorithms and data access behaviors. Most importantly, the failure to declare object IDs between two static transactions that can concurrently access the same objects dynamically can lead to program errors.

The second step is performed by the library or runtime system. This software transforms the high-level object ID declarations into the appropriate instructions that transfer this information down to the hardware TM system. These transformations apply to object ID values that are constant or dynamic (e.g., heap-based addresses).

The final step is for the HTM to use this information during conflict detection. We describe the high-level hardware support, and leave the details for a subsequent section. Conceptually, the TM system stores all the object IDs accessed in each transaction instance. This can be done in hardware, in software, or a hybrid of the two. For eager conflict detection, this information is also passed on each coherence request (as additional meta-data in the messages). For lazy conflict detection this information is sent to each processor (or directory) that participates in the commit algorithm. The conflict detection algorithm uses object IDs as supplemental information to detect

conflicts, along with the original conflict detection state (RW bits or read- and write- signatures in each processor).

The definition of a conflict is augmented as follows. Two processors have a conflict if (1) there is at least one RW, WR, or WW conflict and (2) there is a non-null intersection between the object IDs of the requestor (or commiter) and the responder (or other processors involved in the commit algorithm). Note this definition applies to conflicts that were previously categorized as false conflicts under our original definition. True conflicts will remain conflicts under the revised definition. The correctness of this definition relies on a sufficiently large set of object IDs that encompasses at least all the shared objects between two potentially conflicting transactions. If this set is too small, program errors can arise.

```
XID 1                                   XID 2
// A is shared between XID 1 & 2        // A is shared between XID 1 & 2
tx_begin();                             tx_begin();
 objectid_ptr(&A);                       objectid_ptr(&A);
 A.shared_data = 5;                      A.shared_data = 15;
tx_end();                               tx_end();
```

(a) Correct object ID

```
XID 1                                   XID 2
// A is shared between XID 1 & 2        // A is shared between XID 1 & 2
tx_begin();                             tx_begin();
 objectid_ptr(&A.shared_data);           objectid_ptr(&A.shared_data);
 A.shared_data = 5;                      A.shared_data = 15;
tx_end();                               tx_end();
```

(b) Correct object ID

```
XID 1                                   XID 2
// A is shared between XID 1 & 2        // A is shared between XID 1 & 2
tx_begin();                             tx_begin();
 objectid_ptr(&A);                       A.shared_data = 15;
 A.shared_data = 5;                     tx_end();
tx_end();
```

(c) Incorrect object ID

```
XID 1                                   XID 2
// A is shared between XID 1 & 2        // A is shared between XID 1 & 2
tx_begin();                             tx_begin();
 objectid_ptr(&A);                       objectid_ptr(&A.shared_data);
 A.shared_data = 5;                      A.shared_data = 15;
tx_end();                               tx_end();
```

(d) Incorrect object ID

```
XID 1                                   XID 2
// A is shared between XID 1 & 2        // A is shared between XID 1 & 2
tx_begin();                             tx_begin();
 objectid_ptr(&A);                       objectid_ptr(&B);
 A.shared_data = 5;                      A.shared_data = 15;
tx_end();                               tx_end();
```

(e) Incorrect object ID

FIGURE 7-2. Examples illustrating correct and incorrect object ID declarations. There is no false sharing between any address (object or field) in these examples.

Examples. We now describe several examples of correct and incorrect object ID declarations. Figure 7-2 contains five examples illustrating correct and incorrect object ID declarations for two example XIDs. In our examples, A and B represent two different objects, and A contains the sub-field `shared_data` (additional sub-fields also exist). There are no instances of false sharing between objects A and B and their field addresses in these examples.

The first declaration is (a). It is correct due to two properties. First, the object ID declaration is for the shared object A that is shared between XIDs 1 and 2. Second, the declaration occurs before its use (i.e., before the update to `A.shared_data`).

The second declaration is (b). It is also correct and obeys the two properties mentioned previously. Notice that it is sufficient that the ID be on the sub-field's address rather than on the object's address. This is because the declaration correctly identifies the shared access to `A.shared_data`, and both XIDs use the same object ID declaration.

The third declaration is (c). It is incorrect because XID 2 misses an object ID declaration, which is present in XID 1. In this case concurrent executions of XID 1 and 2 could result in missed conflicts during conflict detection.

The fourth declaration is (d). It is incorrect because the object IDs are on different granularities. The address of object A and its sub-field `A.shared_data` could reside on different addresses. Therefore concurrent updates to `A.shared_data` by XIDs 1 and 2 may proceed without conflicts.

The final declaration is (e). It is incorrect because XID 2 does not correctly identify the shared object that is accessed by the transaction (it marks object B instead of A). The shared object A is correctly identified in XID 1, but conflict detection will still miss conflicts between XID 1 and 2. This can lead to inconsistent state in object A.

In summary, these examples illustrate some correct and incorrect object ID declarations. If the programmer is methodical and is consistent with object ID declarations (i.e., avoid using different granularities for different XIDs), then these problems can be avoided. We concede that these semantics may unduly burden a TM programmer, but the performance benefits of object IDs may outweigh these costs.

Implementation. In this section we briefly sketch the hardware mechanisms required to implement object IDs. Two mechanisms need to be in place that can be used by the underlying TM system. First, each processor needs to keep a list of all the transactional object IDs that are accessed within a dynamic transaction. Second, processors need to exchange and check object ID information for conflict detection. We now describe possible implementations for both mechanisms.

There are at least three options to track per-processor object IDs. In the first option, each processor contains a fixed-size hardware table (implemented as a content-addressable memory, or CAM) that precisely tracks a bounded number of object IDs accessed in the transaction. If a transaction accesses more object IDs than entries in the table then an overflow bit can be set. If this bit is set it means that the processor has touched all possible object IDs in the system. The table's hardware overhead is calculated as follows. If the table has n entries and each entry stores a d-bit object ID, the table requires $n*d$ bits total, plus one overflow bit, for a total of $(n*d + 1)$ bits. This implementation option is attractive if the product $n*d$ is small. Additional bits are needed to store each entry's valid bits or a pointer to the next available free entry in the table.

The second option to track per-processor object IDs is imprecisely through a Bloom filter. A fixed-size Bloom filter allows an unbounded number of object IDs to be stored in hardware. For example, a 64b Bloom filter can track 64 object IDs exactly (assuming all object IDs are equally

probable). In general, the same 64b Bloom filter can track n number of d-bit object IDs imprecisely with an increasing false positive rate that varies directly with n. This option is attractive if $n > m$, where m is the number of entries in the Bloom filter (e.g., m=64 for a 64b Bloom filter). However, if $n >> m$, the false positive rate of the Bloom filter is likely close to 1, and won't be useful in distinguishing object IDs. Compared to the first option, Bloom filters are more attractive than the table if $(n*d) >> m$.

The third option is a hybrid of the two previous options. More specifically, a table tracks n number of d-bit object IDs precisely, and is backed up by a m-bit Bloom filter when the table is full. The total number of bits to implement this solution is $(n*d + m + 1)$, where the (+1) indicates the Bloom filter's valid bit. The hybrid solution works as follows. When there are $(<= n)$ object IDs the table is used to store and look up object IDs. When there are exactly $(n+1)$ object IDs the table is full. At this point, all $(n+1)$ object IDs are hashed and inserted into the Bloom filter, the Bloom filter valid bit is set, and subsequent insertions and lookups re-direct to the Bloom filter.

We now discuss the hardware state required to implement the object ID checks. The main overhead is the extra object ID meta-data that piggy-backs on top of every request message (for eager conflict detection), or the meta-data that is sent to each processor or directory involved in the commit algorithm (for lazy conflict detection). For eager conflict detection, it is sufficient to just send the object ID of the latest object that is accessed (in time). This information can be stored in an extra hardware register on each processor. This information suffices for eager conflict detection because each processor locally maintains a summary of all object IDs that it has accessed. Furthermore, all object IDs are declared prior to the uses of the corresponding object (and are not re-ordered with respect to object accesses). Therefore all accesses to shared objects

will update the object ID register in the processor. For lazy conflict detection, this meta-data summarizes the set of object IDs that have been accessed by the commiter. The size of this meta-data can add significant overhead to commit messages (e.g., a 64b object ID table in addition to a 64b commit message adds 100% overhead). Like the state for object IDs, this meta-data can be stored precisely or imprecisely. Lazy conflict detection can use the commiter's meta-data to check against the object ID meta-data of the directory (using CAM lookups or Bloom filter intersection operations).

Methodology. We evaluate the performance implications of object IDs using ideal overheads (hardware state and access latency). This allows us to get a sense of the upper-bound performance and then weigh it against the implementation and semantic costs.

We focus our evaluation on the STAMP workloads, as these workloads put high pressure on signatures and therefore would likely be the most affected by object IDs. We find that only four STAMP workloads are amenable to our object IDs (Bayes, Yada, Genome, and Vacation). For the remaining STAMP workloads the object IDs are on numerous fine-granularity objects, and would generalize to the lists maintained by Perfect signatures.

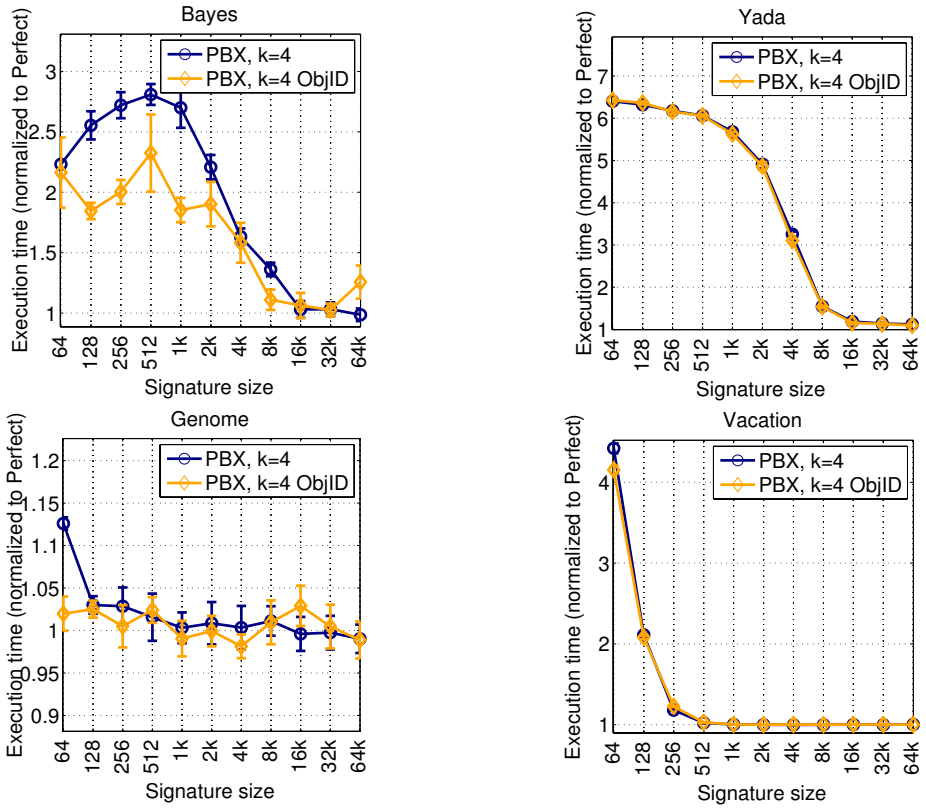


FIGURE 7-3. Normalized execution times before and after using object IDs

Results. Figure 7-3 shows the normalized execution times of workloads before and after object IDs are used. All execution times are normalized to the execution times of Perfect signatures. For each workload we first describe how we defined object IDs. Then we describe the execution time results for each workload.

In Bayes, a high-level object consists of an node, which itself consists of two linked lists, named the parent and child. We focus more fine-grain and make the parent and child pointers of each node the object IDs. For our simulations there are a total of 16 nodes, which means that there are 32 possible object IDs. Although this is a small number of object IDs, since each object ID is a pointer value (32b in our infrastructure), this represents a total of 1kb of state for each processor. This is certainly too much overhead to be passed around on request messages, so it makes more

sense to store these in small Bloom filters (<= 32b) and pass these around on each request message. Overall, object IDs improve execution time for small signature sizes (less than 4kb). More specifically, there is a 28% improvement in execution time for 128b signatures and a 32% improvement for 1kb signatures when object IDs are used. This optimization is not as effective for larger signatures because a smaller fraction of conflicts are false conflicts.

In Genome the program's algorithm operates on hash tables that use hash buckets. There are at least three hash tables, with each table using 16 or 256 buckets. We declare object IDs on the pointers to each hash bucket. Thus there are a maximum of 256 object IDs for each transaction. Again, a small Bloom filter should be used to store and check object IDs. The 64b signature sees the most execution time benefit (9.7%).

Vacation operates on four red-black tree data structures, and we declare object IDs on the pointer to the root of each red-black tree. This small number of object IDs also has a correspondingly small improvement in execution time. Specifically, for 64b signatures there is a 6.1% improvement when object IDs are used. This is because most of the false conflicts occur during intra-tree operations (insert and delete), and not on operations between different trees.

Finally, Yada operates on shared mesh data structures, which itself contains multiple elements (633 for our simulations). We put object IDs on each mesh element. However our results show that there are no performance benefits of object IDs. After further analysis we find that most of the conflicts arise from manipulating the helper data structures which modify the elements. Object IDs would not make sense here because they would operate on many fine-grain addresses.

In summary, these results do not provide strong evidence that object IDs should be implemented. Some of the improvements are significant (in Bayes) but the remaining workloads show

modest improvements. Furthermore the hardware state to store and check against the object IDs, coupled with the large semantic learning curve for object IDs, are too costly compared to the performance benefits.

XID independence and object IDs are two alternatives to reducing the performance degradations of signature false conflicts. Due to its high costs we now turn to a more promising approach, spatial locality, that improves signature performance by exploiting well-known memory reference behaviors of programs.

7.3 Spatial Locality

7.3.1 Motivation

Many programs exhibit some amount of spatial locality, in which addresses neighboring the current memory access are likely to be referenced in the near future [107]. For example, suppose cache block A (of size 64B) is accessed, and its memory location is address 0x0. If another cache block B is accessed at memory location 0x40 (i.e., 64B from A), then the accesses to A and B would be considered spatially local.

Signatures can choose to perform hashing on different granularities. By default, signatures hash on the granularities of cache blocks. Thus, if addresses A and B are inserted into a signature hashing on cache block granularity, they could potentially set two different sets of bits (one set for each address). However, if signatures hash on a larger granularity (e.g., 128B for our example), there is a possibility addresses A and B could both be represented in the signature using one set of bits. Signatures can benefit from this optimization because fewer bits mean the signature is less utilized. Lower utilization has the potential to reduce false positives for signature lookups,

depending on which hashing granularity is used. However, a negative consequence of hashing on larger granularity is the loss of information during lookups. In our example, signature lookups can not guarantee whether only A or B have been accessed, or whether both blocks were accessed. If the hashing granularity is sufficiently large, a multi-threaded program runs into problems with false sharing amongst threads [111].

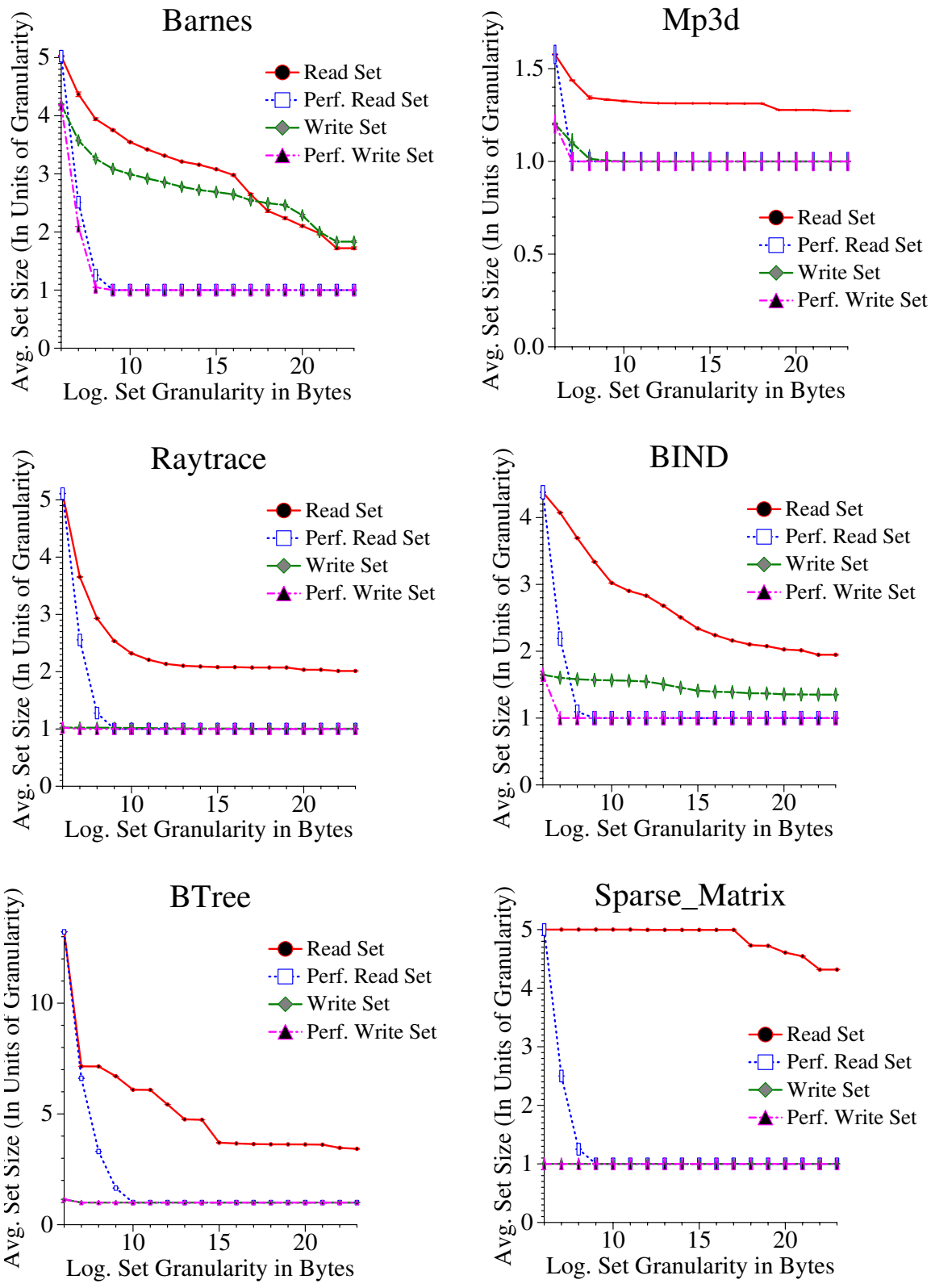


FIGURE 7-4. Read- and write-set sizes for 64B to 8MB granularities. X-axis represents the logarithm base 2 of the set granularity in bytes.

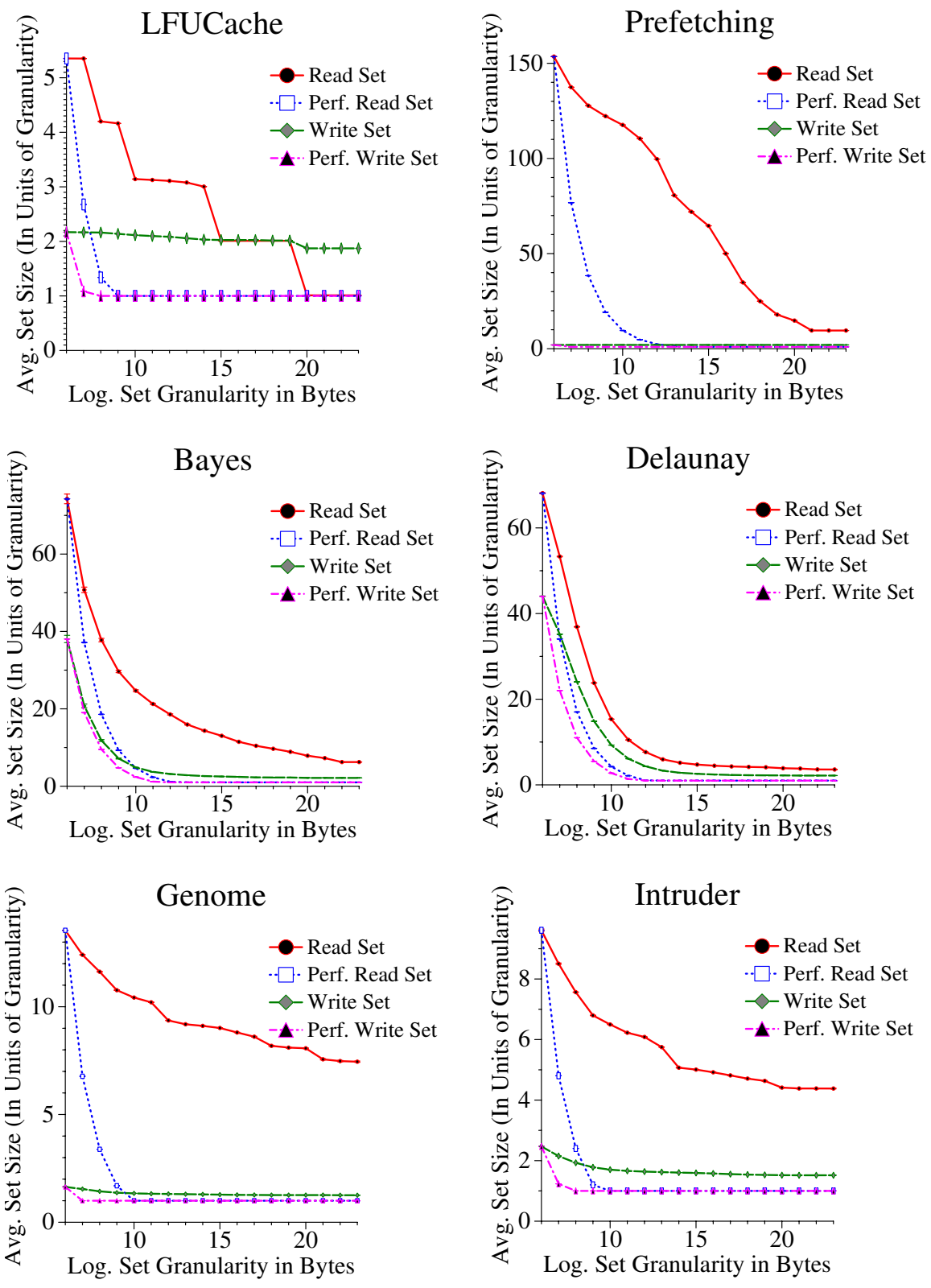


FIGURE 7-4. Read- and write-set sizes for 64B to 8MB granularities. X-axis represents the logarithm base 2 of the set granularity in bytes.

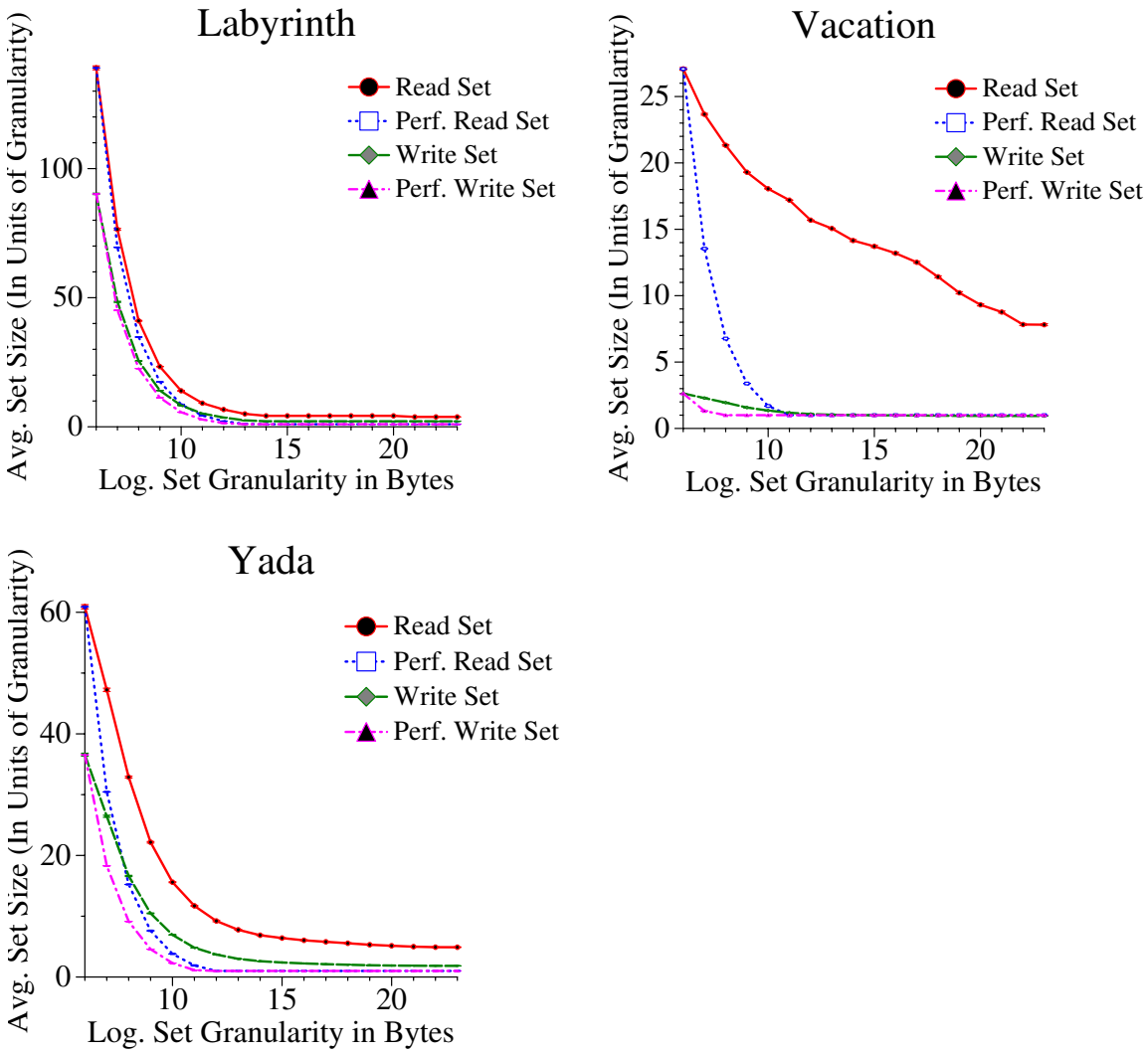


FIGURE 7-4. Read- and write-set sizes for 64B to 8MB granularities. X-axis represents the logarithm base 2 of the set granularity in bytes.

7.3.2 Measuring Spatial Locality

Profiling read- and write-sets. We attempt to capture our workload's spatial locality by profiling the sizes of the transactional read- and write-sets for various granularities of memory accesses, ranging from a cache-block (64B for our system) to 8MB. We perform these measurements on transactional accesses and provide separate measurements for the read- and write-sets. These measurements differ from the entropy results from Chapter 5 because our entropy results indirectly measure spatial locality, whereas these measurements directly measure spatial locality. Section 7.5 describes how entropy may be used in future implementations of our spatial locality extension to dynamically adapt to changes in a workload's spatial locality.

These measurements are presented in Figure 7-4. Note that the x-axis shows the log base two of the granularity unit starting at 64B (e.g., 1kB would be 10 on the x-axis). We define two metrics which show the extremes of spatial locality in relation to the axes used in Figure 7-4. Workloads with *no spatial locality* will be represented as horizontal lines in the figure. This is because the sizes of its read- and write-sets will be unaffected by changes in the granularities at which those sets are measured. For workloads with *perfect spatial locality*, doubling the granularity size will halve the set size (with a minimum set size of one which will be covered by a sufficiently large granularity). For example, if a workload's read- and write-sets contain 20 and 10 addresses, respectively, for 64B granularities, with perfect locality increasing the granularity to 128B will result in 10 and 5 addresses, respectively. Note that Figure 7-4 shows both the actual locality measurements (lines labeled "Read Set" and "Write Set") as well as the measurements assuming perfect spatial locality (lines labeled "Perf. Read Set" and "Perf. Write Set").

A quantitative measure of spatial locality. Although Figure 7-4 is the ultimate truth on a workload's spatial locality, it is useful to derive a workload-independent metric of spatial locality that can be used to compare the amount of spatial locality across workloads. We propose such a metric which we call *average spatial locality*. This metric describes, on average, how different the placement of a transaction's read- and write-set is from their placement assuming perfect spatial locality. However, the value of average spatial locality is not very accurate at representing the distances between cache blocks (because our granularities are on multiples of cache lines). Thus the best way to calculate distances between cache blocks is to measure it empirically for each workload.

We calculate average spatial locality as follows. First, we start with the spatial locality measurements as shown in Figure 7-4. This allows us to quantify the number of memory regions (for actual and perfect locality) that encompass the read- and write-sets for granularities from 64B to 8MB. Second, using these measurements we can derive the multiplicative factor between the perfect and actual spatial locality measurements, for each granularity. The arithmetic average of these multiplicative factors across all granularities is the average spatial locality.

This metric indicates how far off a workload's spatial locality is from perfect spatial locality. If this metric is 1.0, this means that the actual spatial locality of a workload is equivalent to that of perfect spatial locality. The higher the value of this metric, the worse the spatial locality of the workload is from that of perfect spatial locality.

For example, take the Delaunay workload from Figure 7-4. From granularities of 64B to 1kB the actual number of regions for the read-set is 68, 53, 36, 23, and 15. Under perfect spatial locality the number of regions for the same granularities is 68, 34, 17, 8.5, and 4.3. Thus the multipli-

TABLE 7-2. Average spatial locality values for all workloads

Workload	Average spatial locality value	
	Read-set	Write-set
Barnes	2.62	2.44
Mp3d	1.30	1.01
Raytrace	2.03	1.00
BIND	2.35	1.43
BTree	3.77	1.00
Sparse Matrix	4.40	1.00
LFUCache	2.21	1.96
Prefetch	25.80	1.94
Bayes	8.31	2.16
Delaunay	3.95	2.57
Genome	7.46	1.29
Intruder	4.60	1.59
Labyrinth	3.18	1.90
Vacation	9.95	1.14
Yada	5.04	2.29

cative factors (computed by number of actual regions divided by number of regions in perfect locality) for these granularities is 1.0, 1.37, 2.17, 2.80, and 3.61. Then we average these constant factors to obtain the average spatial locality value of 2.19 (i.e., its spatial locality is on average two times worse than perfect spatial locality).

Table 7-2 shows the average spatial locality metric for all our workloads. We now point out some interesting quantitative trends from this table. First, the STAMP workloads exhibit poor spatial locality, with average spatial locality values ranging from 3-10 for the read-set. This means that it requires large memory regions to capture it’s read- and write-sets. Second, the benchmark with the worst spatial locality is Prefetch, with a locality metric of 26 for the read-set. This is expected, as Prefetch was designed to access a large number of cache blocks across many regions of memory. It also implies that this workload will not benefit from spatial locality.

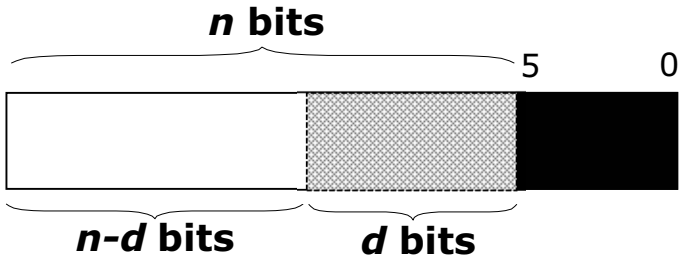


FIGURE 7-5. The bits that are used for static signatures optimizing for spatial locality. The low-order *d* bits are ignored, and the remaining *n-d* bits are used as inputs to signature hash functions.

Having quantified the amount of spatial locality in our workloads, we now turn to how signatures can take advantage of spatial locality. We focus on three different hardware schemes and quantify their impact on the workload's execution time. The three schemes are static spatial locality, coarse-fine hashing, and dynamic spatial locality. We now examine each scheme in turn.

7.3.3 Spatial Locality using Static Signatures

Description. The main idea behind spatial locality with static signatures is to increase the granularity in which we insert addresses in the signatures. This granularity is statically chosen for a signature configuration. The larger the granularity, the higher the probability that fewer bits will be set in signatures. The basic trade-off between hashing granularity and false conflicts is that hashing on larger granularities reduces the probability of false conflicts on objects residing far apart in memory, but could increase false conflicts on objects close together in memory. Figure 7-5 illustrates the bit-fields that are ignored and used by static signatures optimized for spatial locality. We refer to this figure in our discussions below.

Implementation. By default signatures operate on block addresses (64B granularities in our HTM system, and shown as the blacked out bit-field in Figure 7-5). In order to examine the effects of spatial locality we look at larger granularities — 128B, 512B, 2kB, 8kB, and 32kB. This

is implemented by skipping some number of low-order address bits before inserting (or checking) the address into the signatures. Referring to Figure 7-5, this is done by changing variable *d* from 1 to 9 bits, to represent 128B to 32kB granularities.

The remaining *n-d* bits are used as input to the signature's hash functions, modeled after PBX. We choose to use an 11b cache-index field for all PBX signature configuration. This limits the maximum granularity to 32kB (i.e., max *d* value of 9 bits). At 32kB the PPN bit-field for PBX is two bits, since we start PPN immediately following the cache-index bit-field, and we skip the highest-order four bits due to their constant values. However, this is not an unreasonable restriction since Figure 7-4 shows that many workloads exhibit characteristics of spatial locality prior to the 32kB granularity.

We restrict the spatial locality optimization to the XIDs with the largest read- and write-set sizes based on offline profiling of the read- and write-sets for each workload. This allows us to target the effectiveness of spatial locality at the XIDs which will most likely benefit from it.

Finally we choose to limit the number of hash functions (a maximum of four for our configurations) that operate with spatial locality for certain signature sizes. Larger signature sizes are more likely to degrade when all hashes operate on larger granularities, because they are able to represent the same set of addresses with fewer false positives than smaller signatures. Reducing the number of signature bits set for these signatures likely increases the false positive rate of these signatures. Therefore for the STAMP workloads we restrict 1kb signatures and larger to allow two hash functions to operate with our spatial locality scheme. For the remaining workloads, this restriction applies to signatures 512b and larger. These cutoffs are somewhat arbitrary but our results confirm these boundaries are effective.

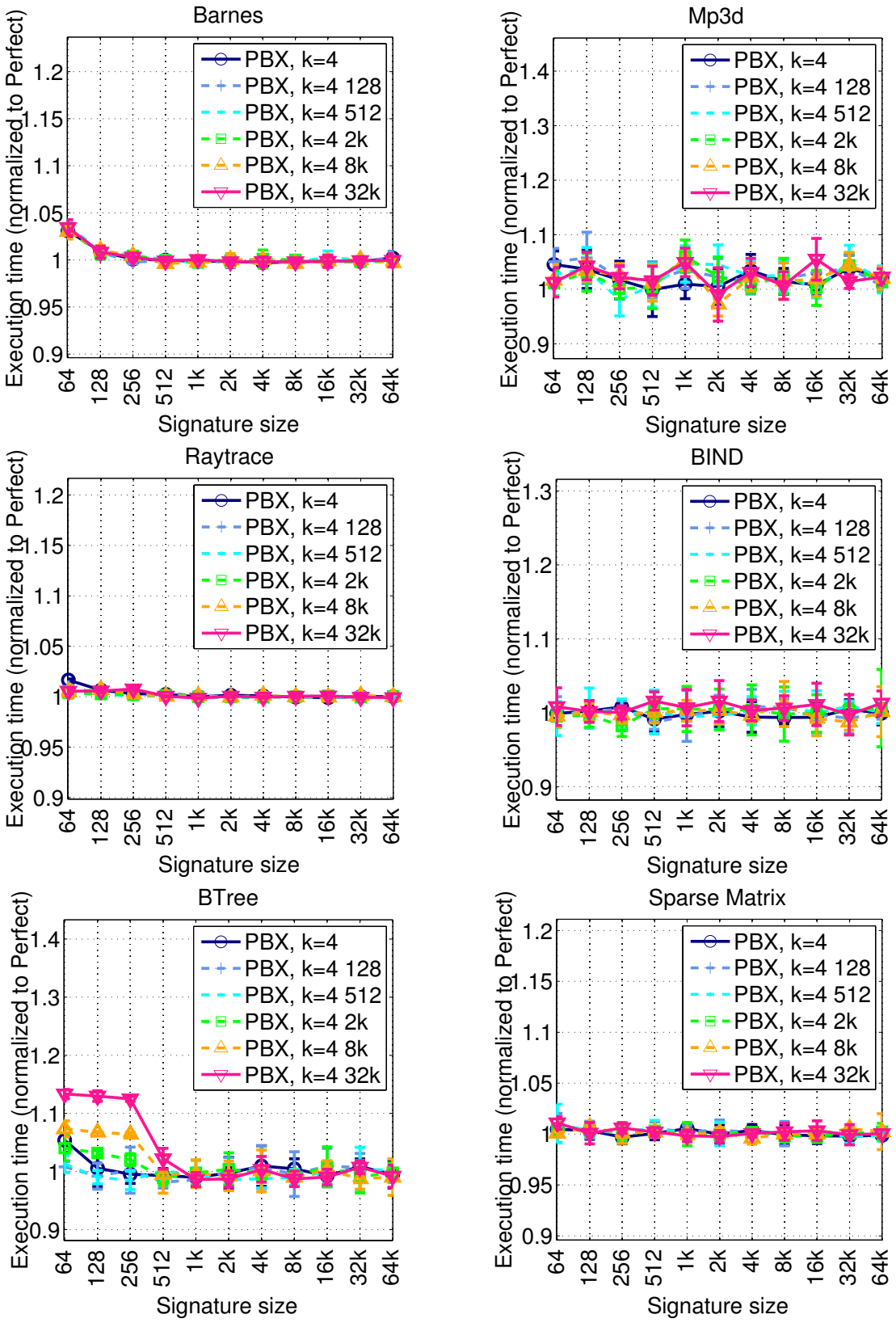


FIGURE 7-6. Normalized execution times of spatial locality for static signatures

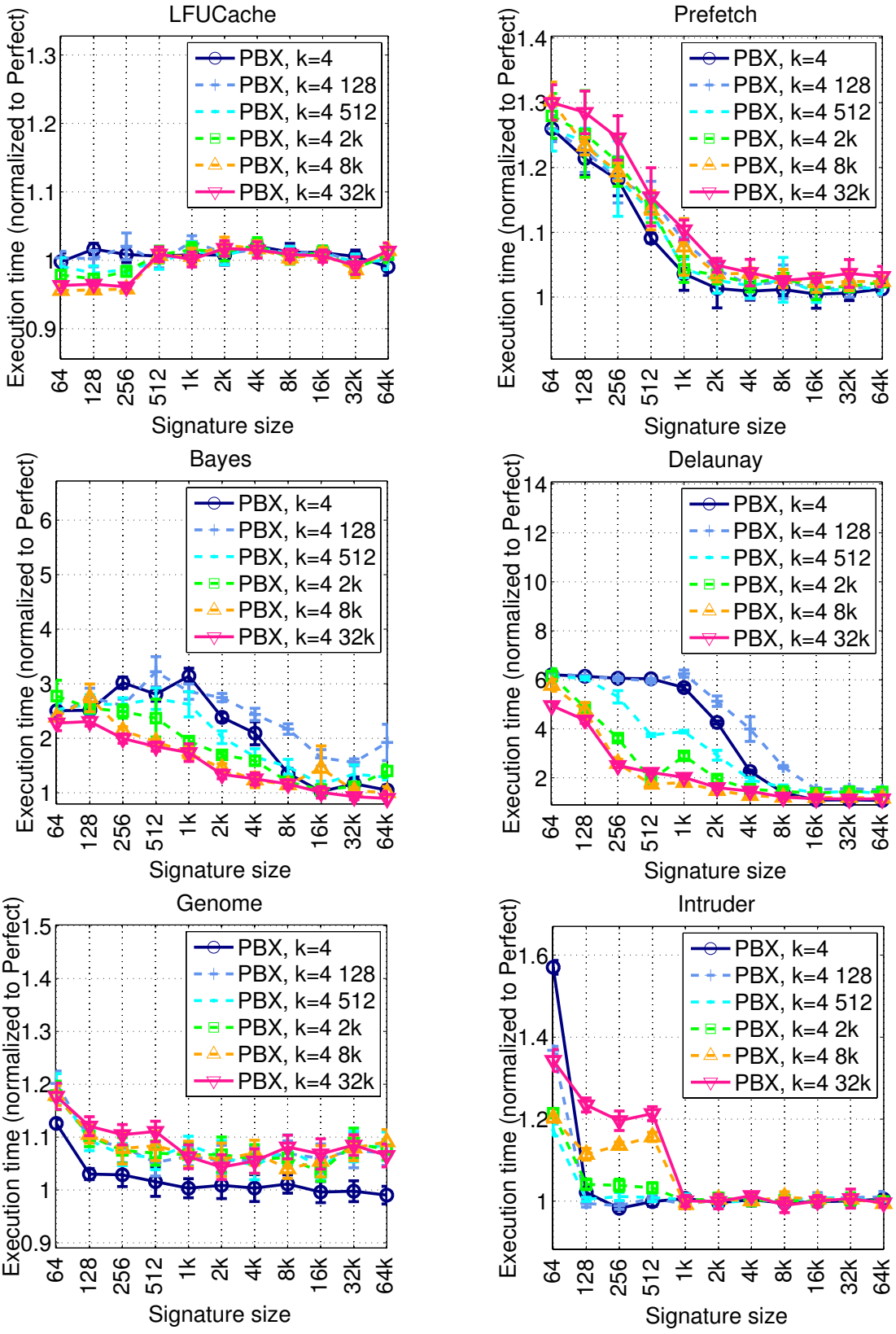


FIGURE 7-6. Normalized execution times of spatial locality for static signatures

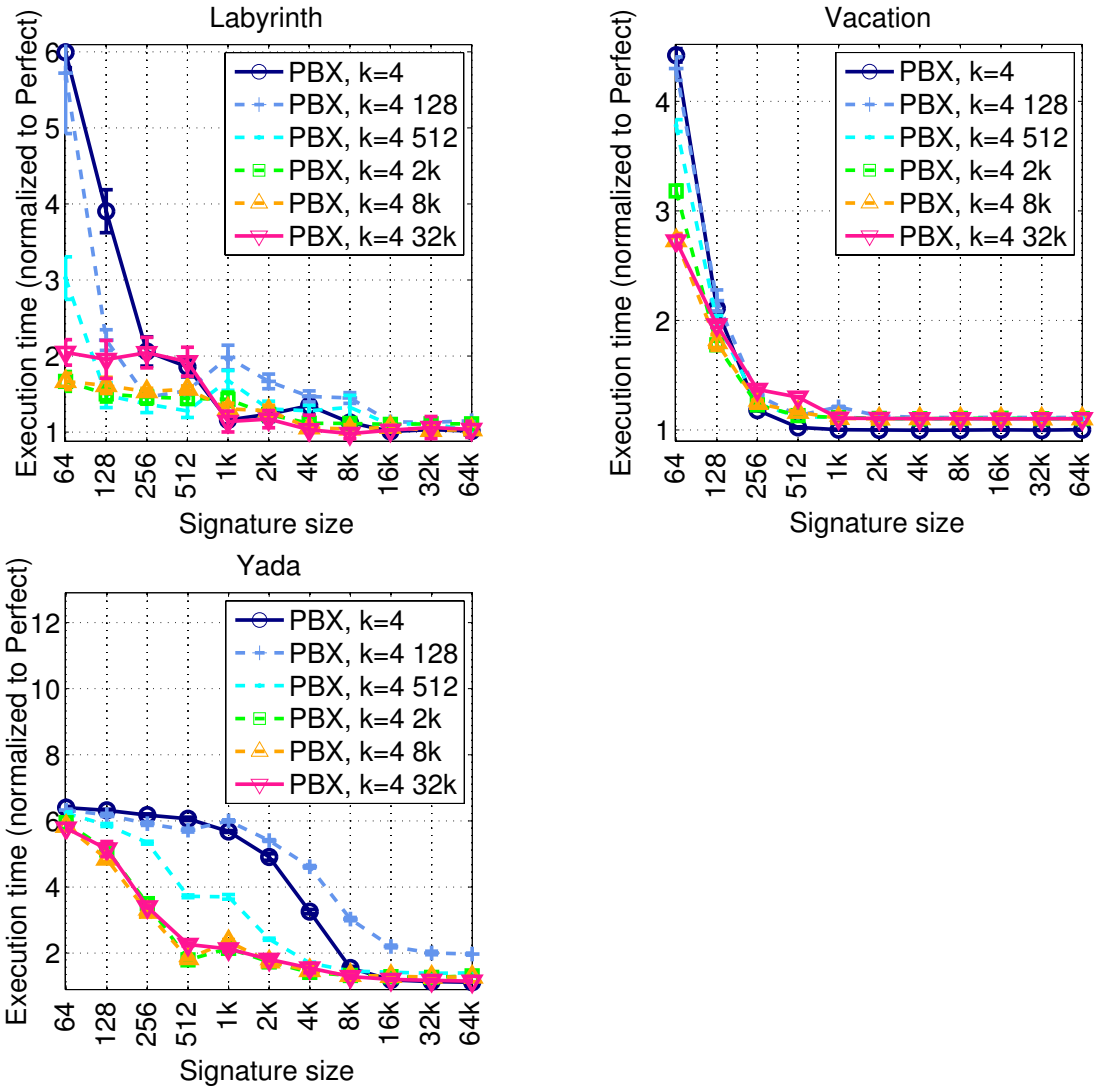


FIGURE 7-6. Normalized execution times of spatial locality for static signatures

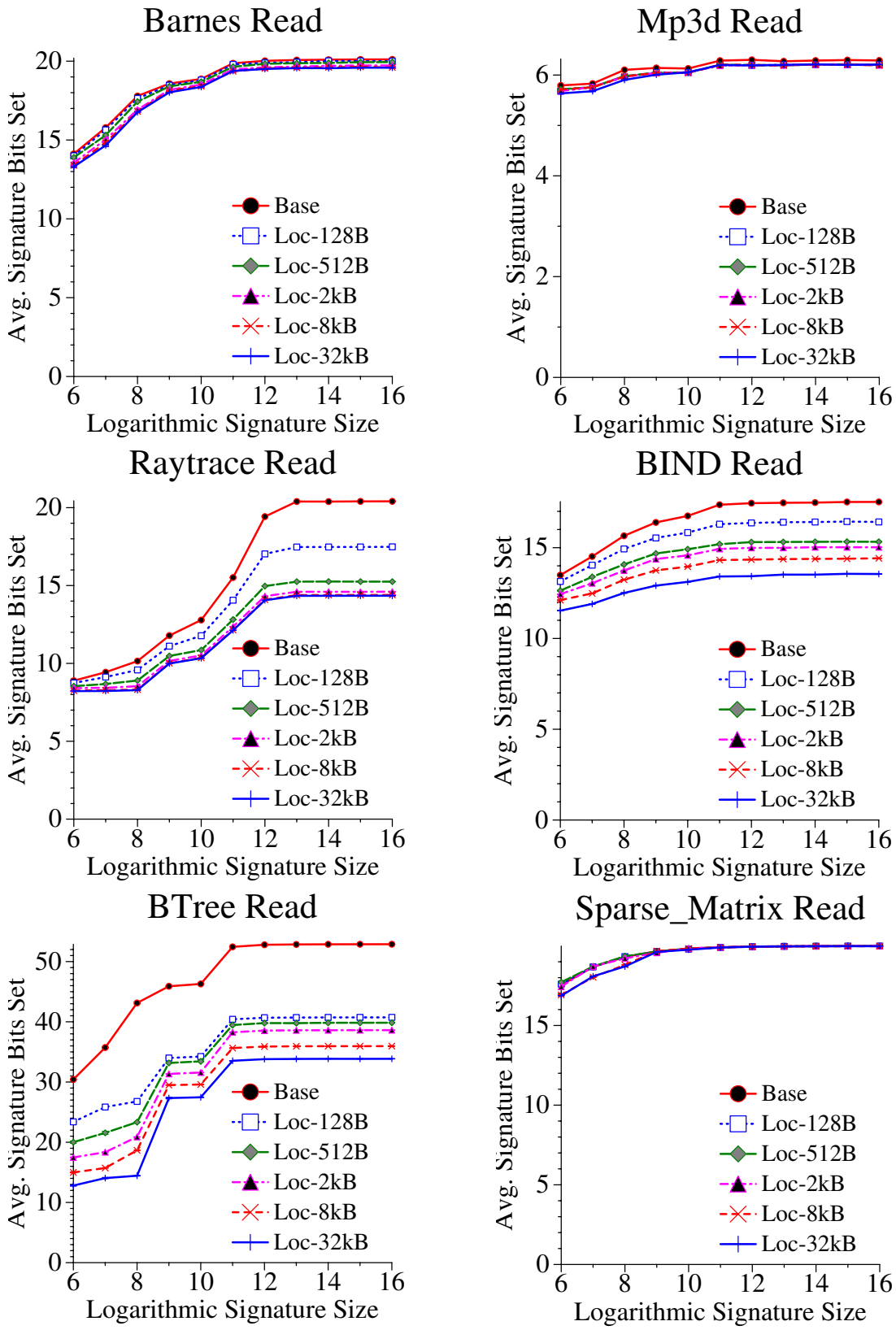


FIGURE 7-7. Average read signature bits set for static signatures. X-axis is logarithm base 2 of the signature size in bits

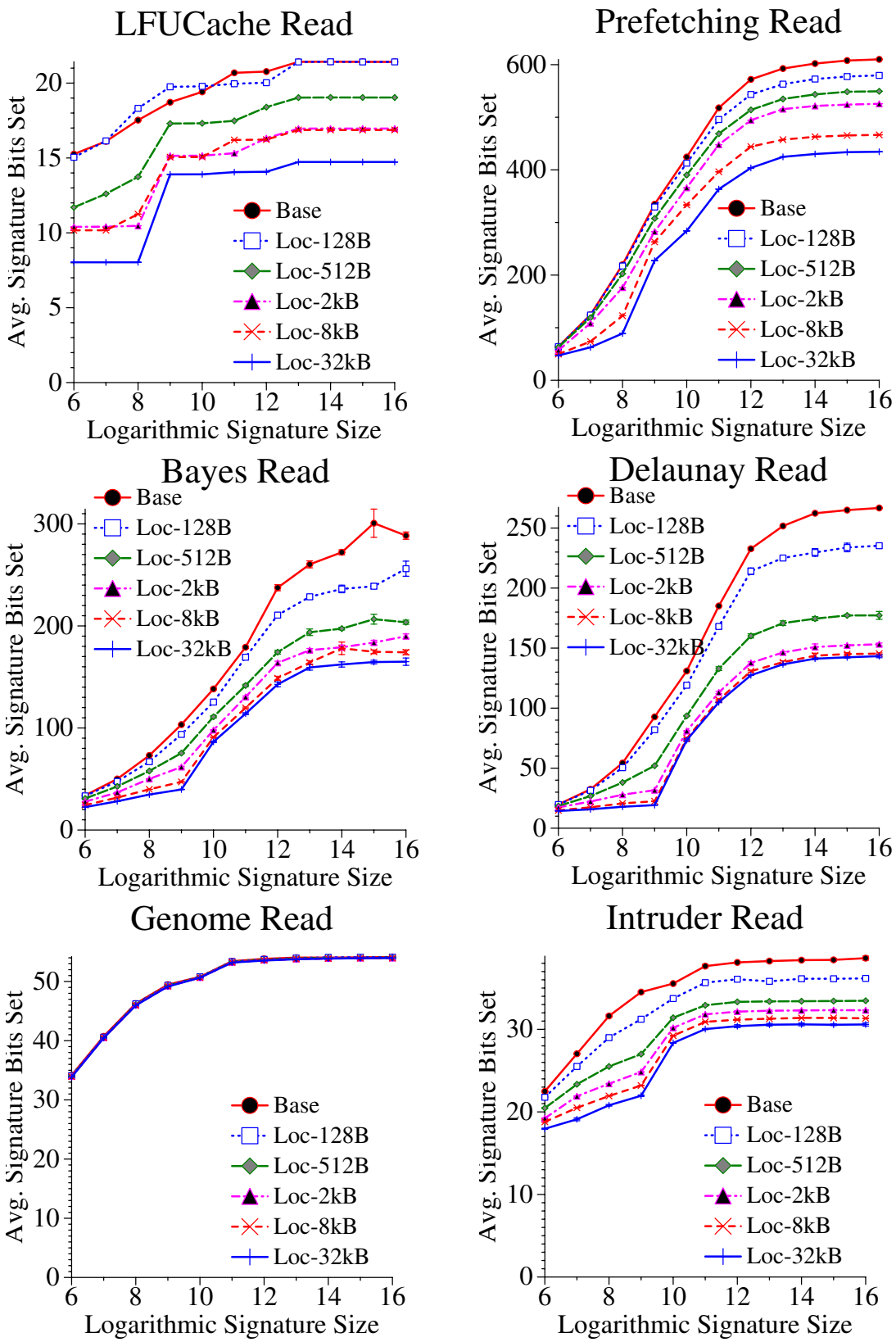


FIGURE 7-7. Average read signature bits set for static signatures. X-axis is logarithm base 2 of the signature size in bits

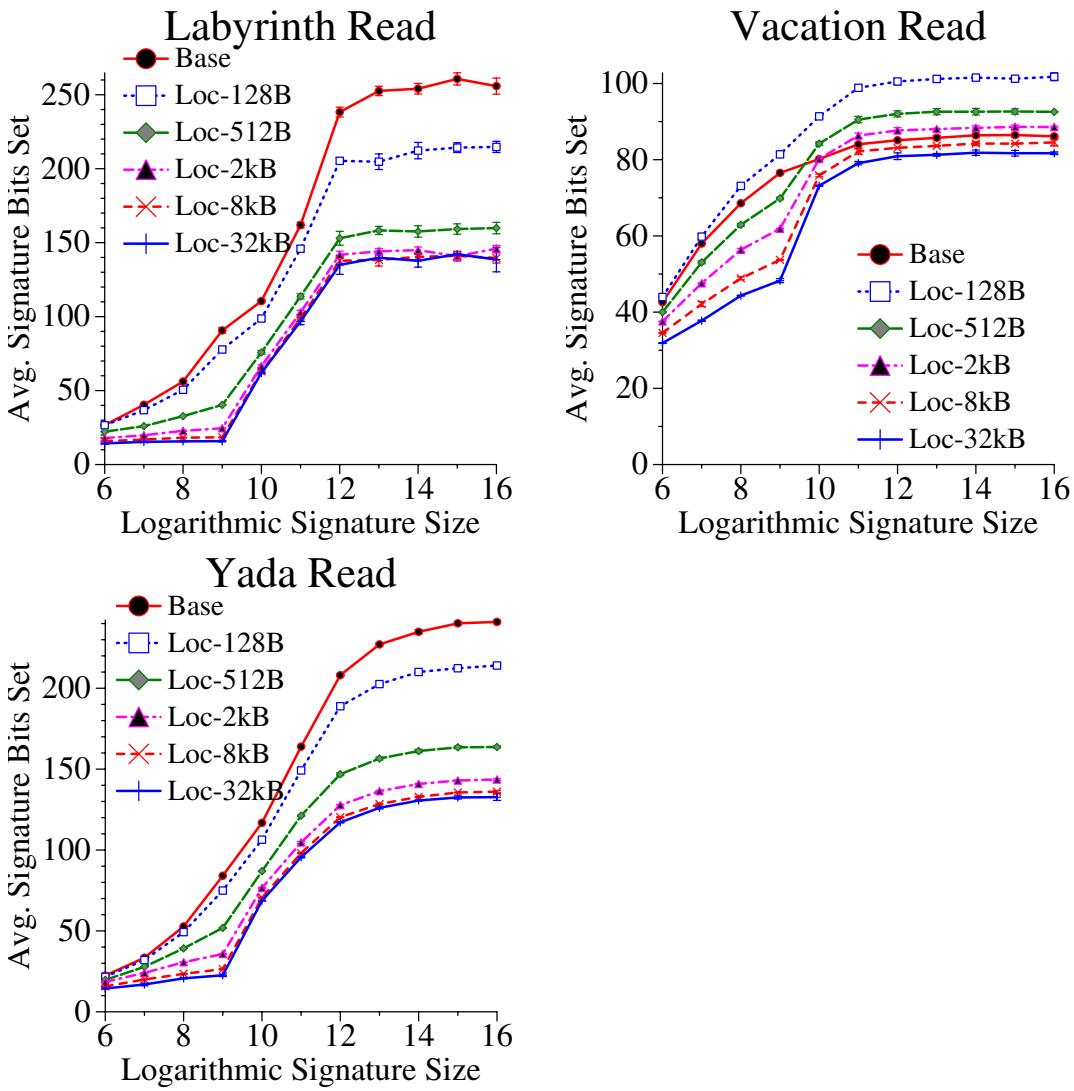


FIGURE 7-7. Average read signature bits set for static signatures. X-axis is logarithm base 2 of the signature size in bits

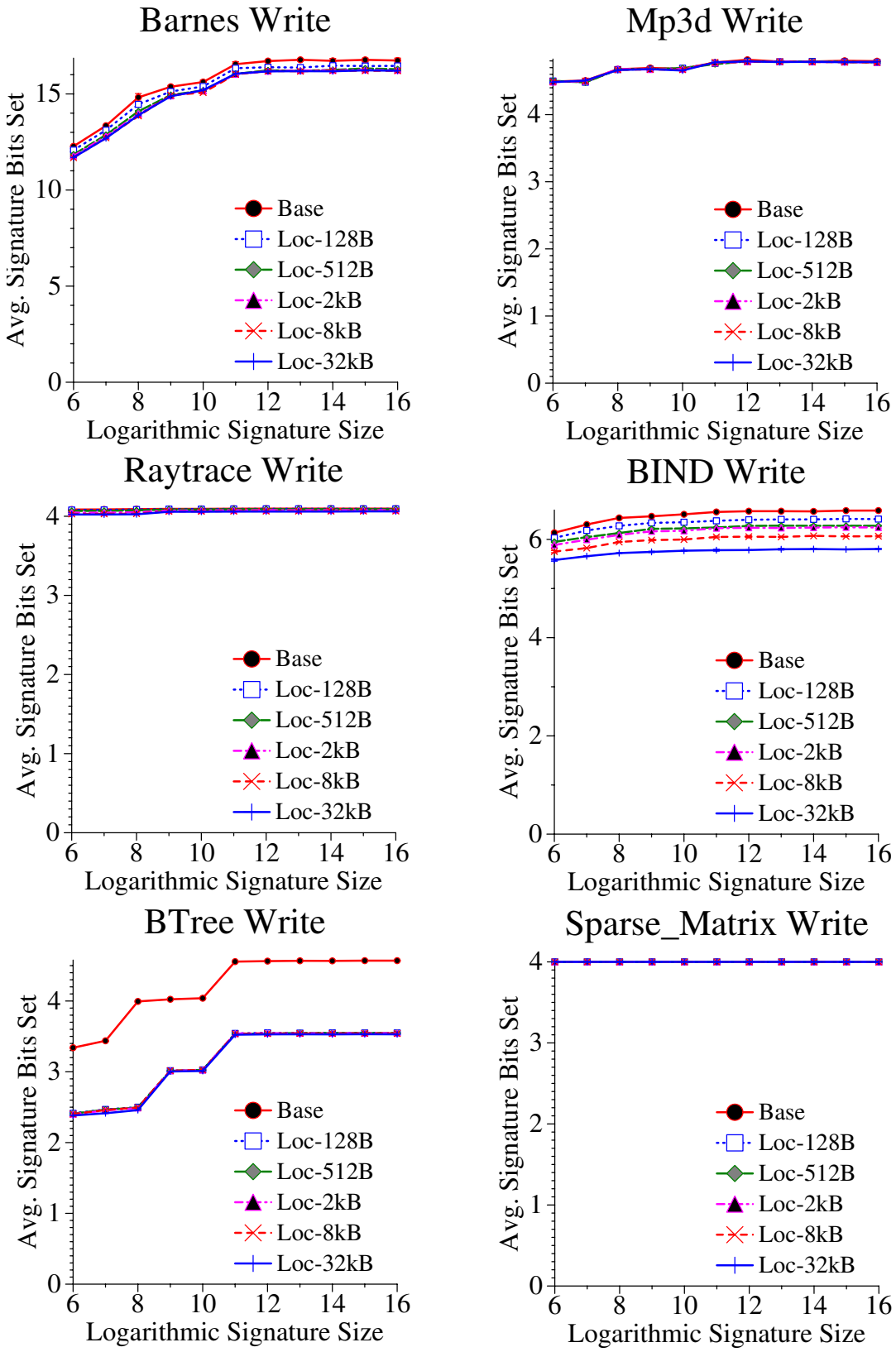


FIGURE 7-8. Average write signature bits set for static signatures. X-axis is logarithm base 2 of the signature size in bits

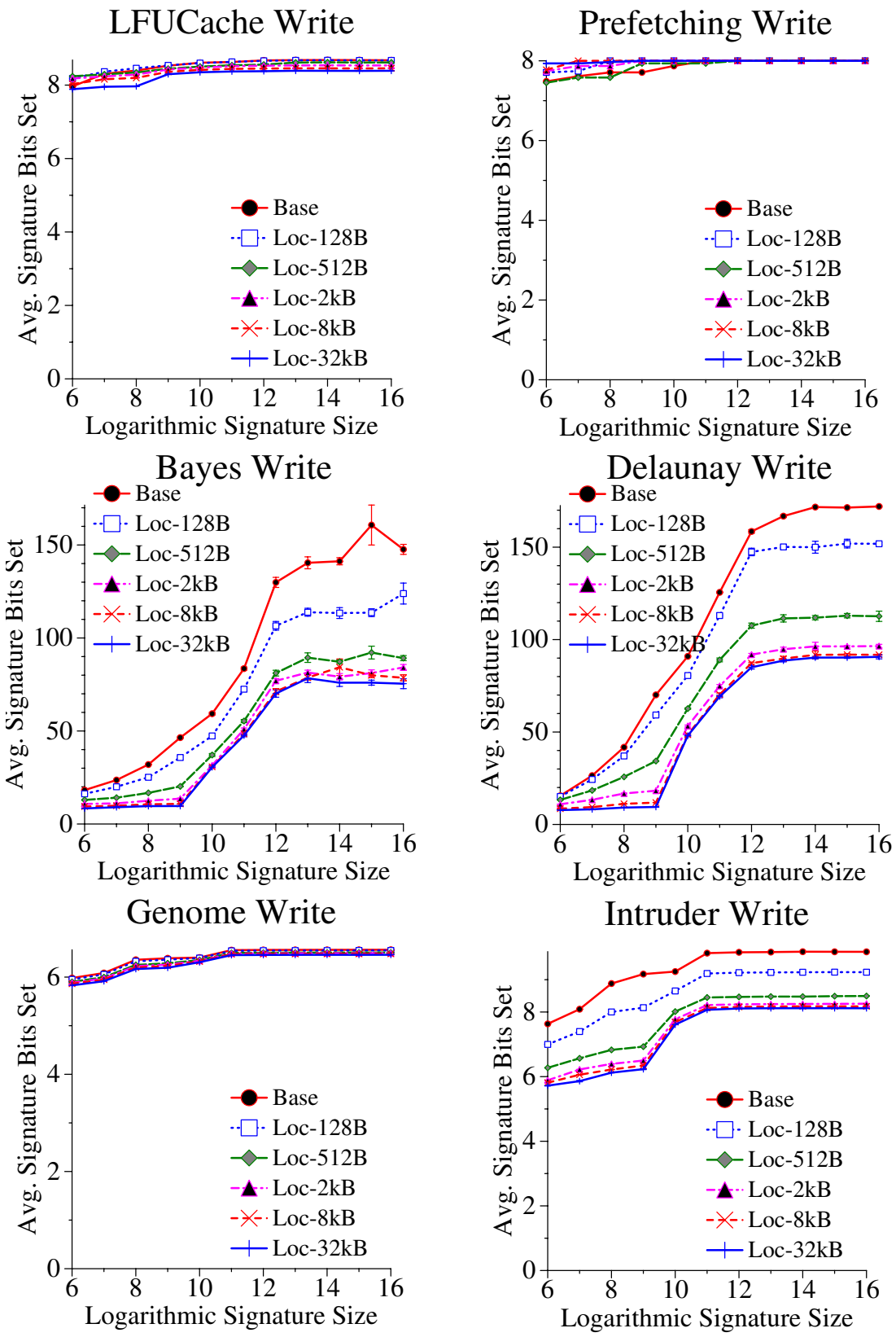


FIGURE 7-8. Average write signature bits set for static signatures. X-axis is logarithm base 2 of the signature size in bits

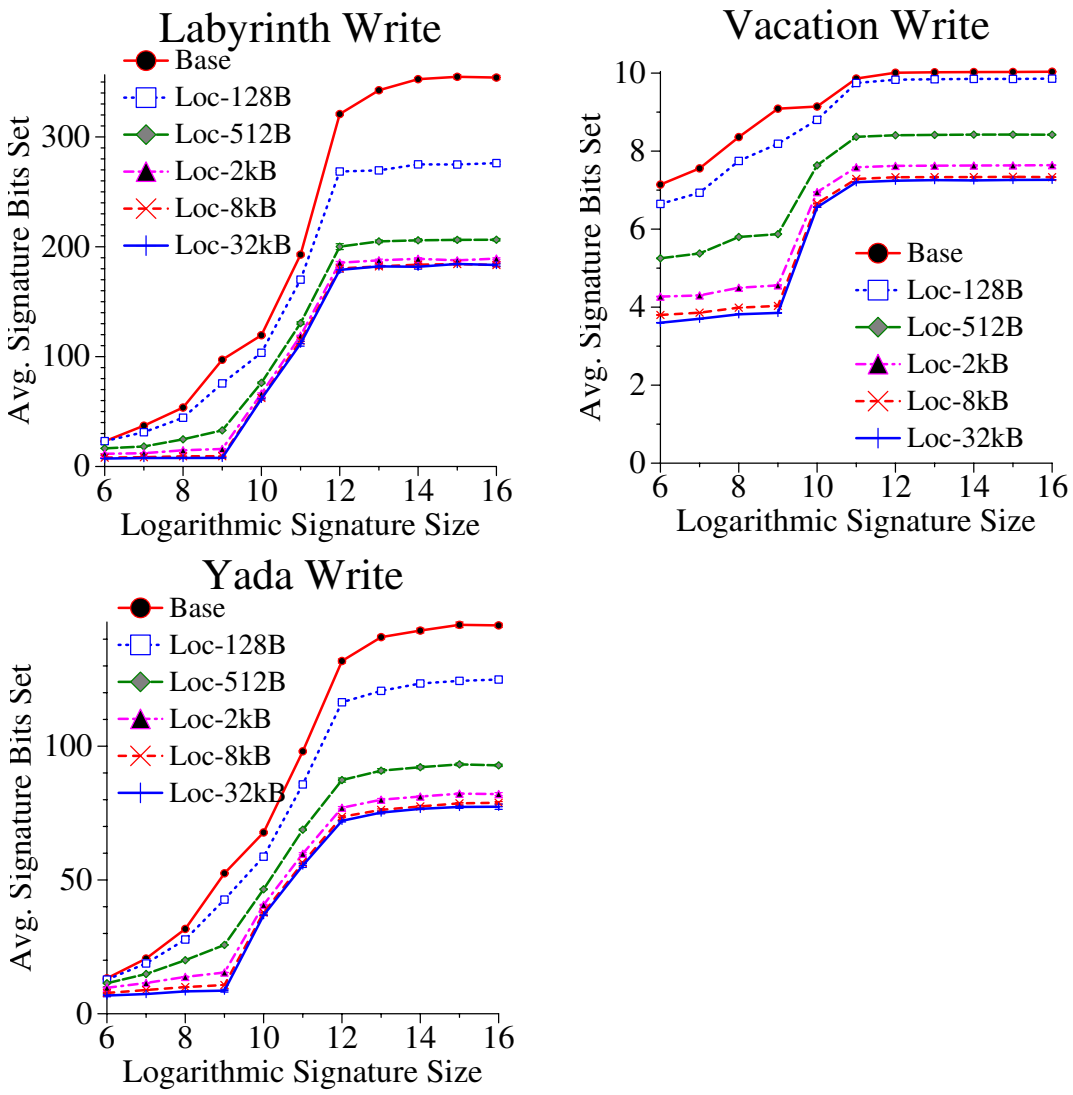


FIGURE 7-8. Average write signature bits set for static signatures. X-axis is logarithm base 2 of the signature size in bits

Results. We first examine the overall normalized runtime results, and then explain the results by examining the reductions in signature bits set (for both the read and write signatures). We will refer to Figure 7-6, Figure 7-7, and Figure 7-8. Figure 7-6 shows the normalized execution times for all our workloads after spatial locality is used. Figure 7-7 shows the average number of read signature bits set for each static granularity, and Figure 7-8 shows the same metric but for the write signature. Note that the x-axis in the last two figures represent the logarithm base two of the signature size in bits (e.g., a 256b signature is value 8 on the x-axis). The lines with spatial locality are labeled with the granularity used (128B to 32kB).

We now point out some interesting trends. First, the SPLASH-2 (Barnes, Mp3d, Raytrace) and BIND workloads do not show significant benefits in execution time from using spatial locality. This can be explained by the average number of signature bits set for the smaller signature sizes (Figure 7-7 and Figure 7-8). For these workloads spatial locality does not significantly reduce the number of bits set (i.e., the lines for spatial locality are close to the original Base signature). Second, there are many workloads which benefit from using spatial locality. These workloads include LFUCache, Bayes, Delaunay, Intruder, Labyrinth, Vacation, and Yada. These reductions in execution time correspond directly to reductions in bits set in the read and write signatures. Finally, spatial locality may sometimes hurt signature performance when the granularity is too large. Examples of this degradation occur in BTree, Genome, Prefetch, and Intruder. For example, in BTree using a 32kB granularity leads to degradations in the normalized execution time (i.e., a slowdown of 1.14 for the 32kB granularity versus 1.05 for Base). As fewer signature bits are set, more addresses are mapped to the same set of signature bits, and false conflicts increase. At some point, transactions start serializing. Genome is the outlier in this group of workloads because spatial locality do not significantly affect the number of bits set. Yet execution time degrades. Further

analysis with our performance debugging reveals that spatial locality alters program execution through a combination of more stalls to non-transactional memory requests and additional non-transactional cycles (including register window spill/fill and TLB traps).

In summary, spatial locality using static signatures can be an effective hardware scheme that reduces the number of signature bits that are set given a set of addresses. A weakness of static signatures is its inability to adapt hashing granularity to a specific signature size. If the granularity is too small there will be no significant effects on execution time. If the granularity is too large, few signature bits will be set, and the resulting false conflicts will serialize the system and degrade execution time.

7.3.4 Spatial Locality using Dynamic Signatures

Description. Spatial locality using static signatures, although beneficial for most workloads, requires that the designer specify the granularity of hashing statically. Furthermore the best granularity likely differs from workload to workload. We propose dynamic signatures to adapt the granularity based on feedback information from the hardware.

Implementation. We idealize the hardware overheads of dynamic signatures. Specifically, we implement it through a set of signatures, each hashing on a distinct granularity (128B to 32kB). There are no overheads to access signatures. Insertions operate by inserting an address in all signatures in the set. Lookups operate as follows. Every lookup queries the signatures associated with the granularity that is currently deemed the “best” (with the default being signatures operating on cache blocks). Our proxy for determining the best signature is the metric of signature hits (i.e., positive signature lookups). Fewer signature hits directly correspond to fewer false positives. Each signature has a separate set of performance counters that tracks the cumulative number of

signature hits for each static XID. In our evaluations we never clear the hit these counters. At every transaction begin our policy selects the signature with the lowest hit counter. This signature is used for lookups when a subsequent instance of that XID executes.

Alternative implementations may implement signature lookups by querying all signatures, rather than just one chosen signature. However, this alternative is likely to incur increased power overheads over our proposed implementation, because multiple signatures are active rather than just one active signature.

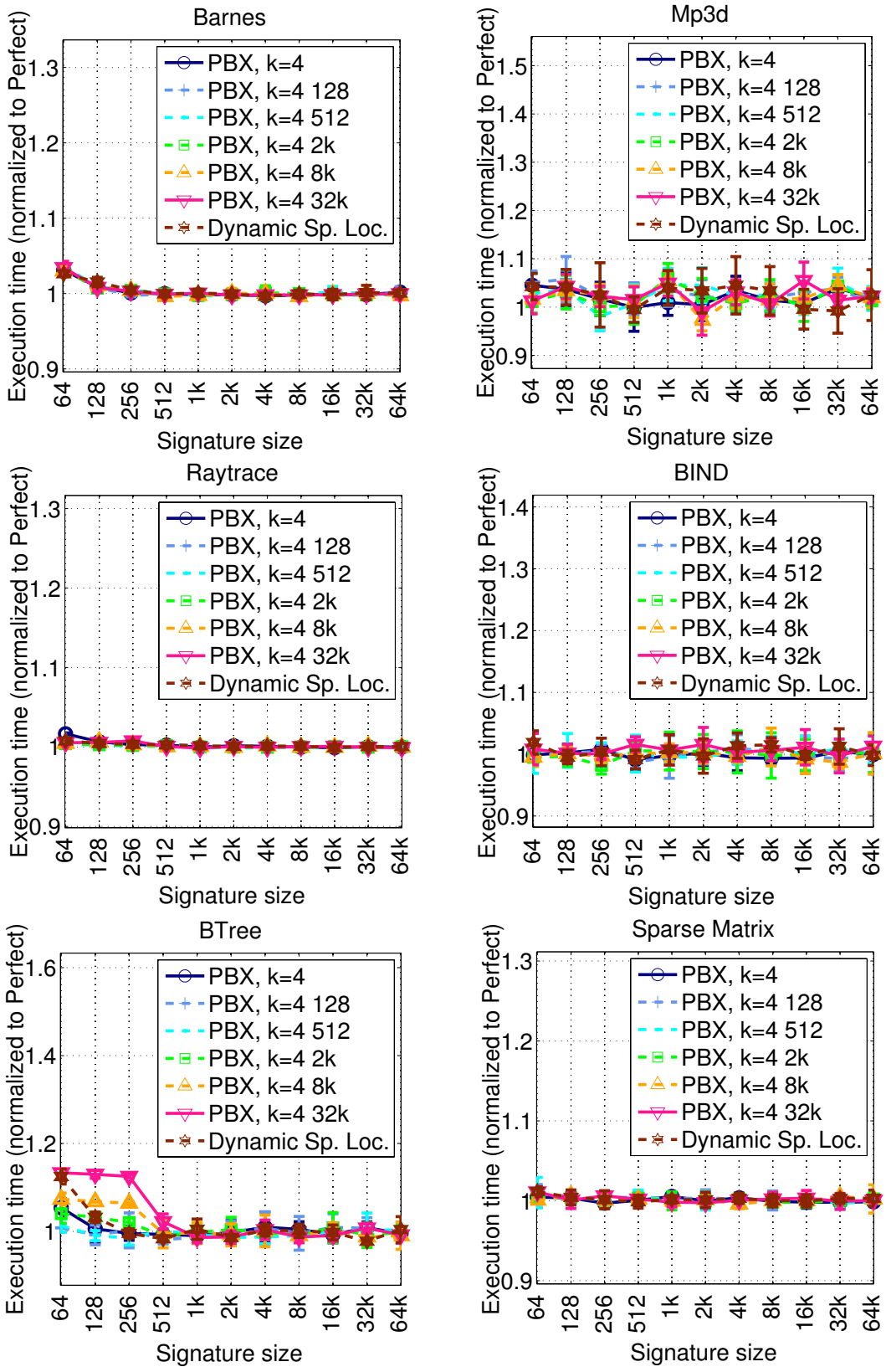


FIGURE 7-9. Normalized execution times of spatial locality for dynamic signatures

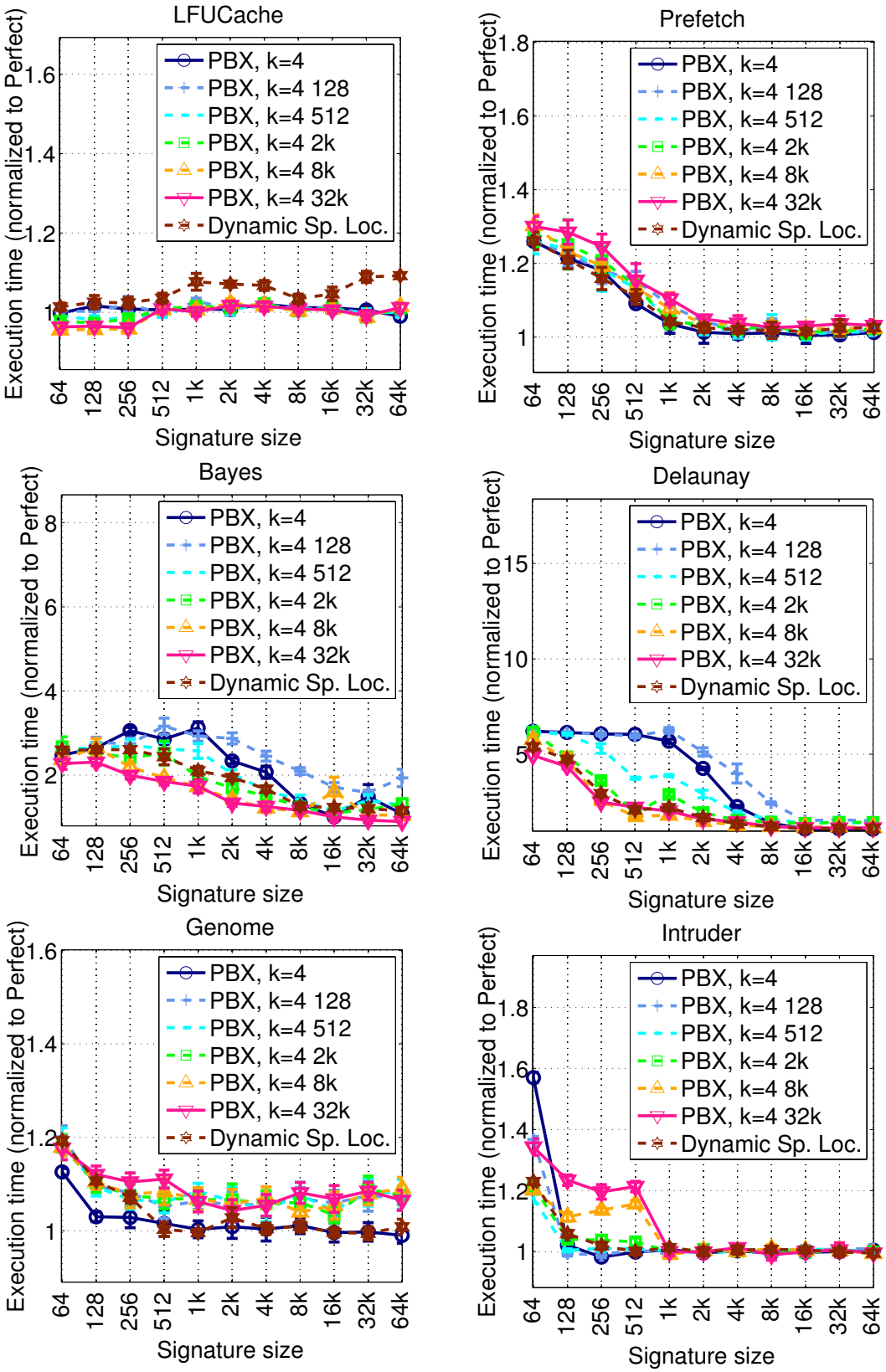


FIGURE 7-9. Normalized execution times of spatial locality for dynamic signatures

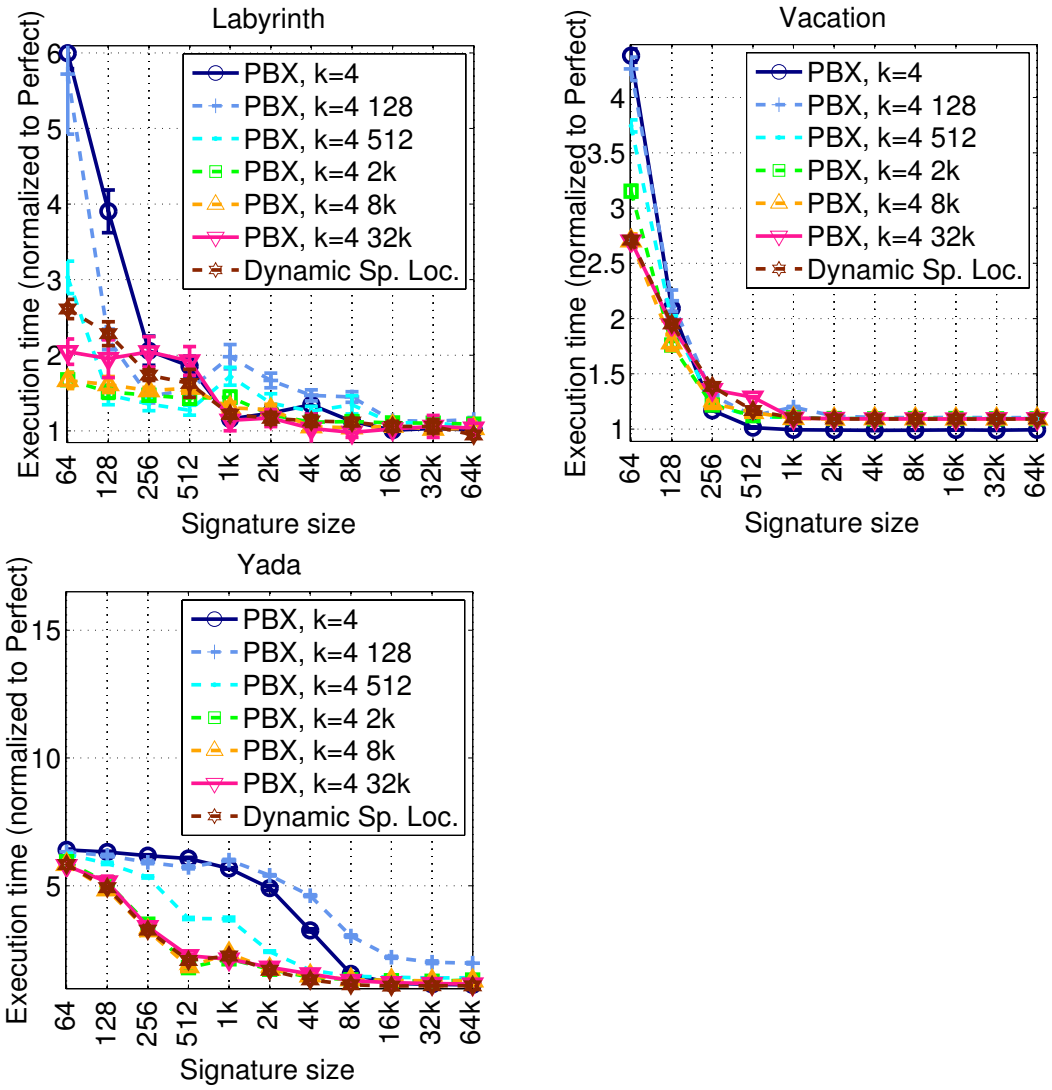


FIGURE 7-9. Normalized execution times of spatial locality for dynamic signatures

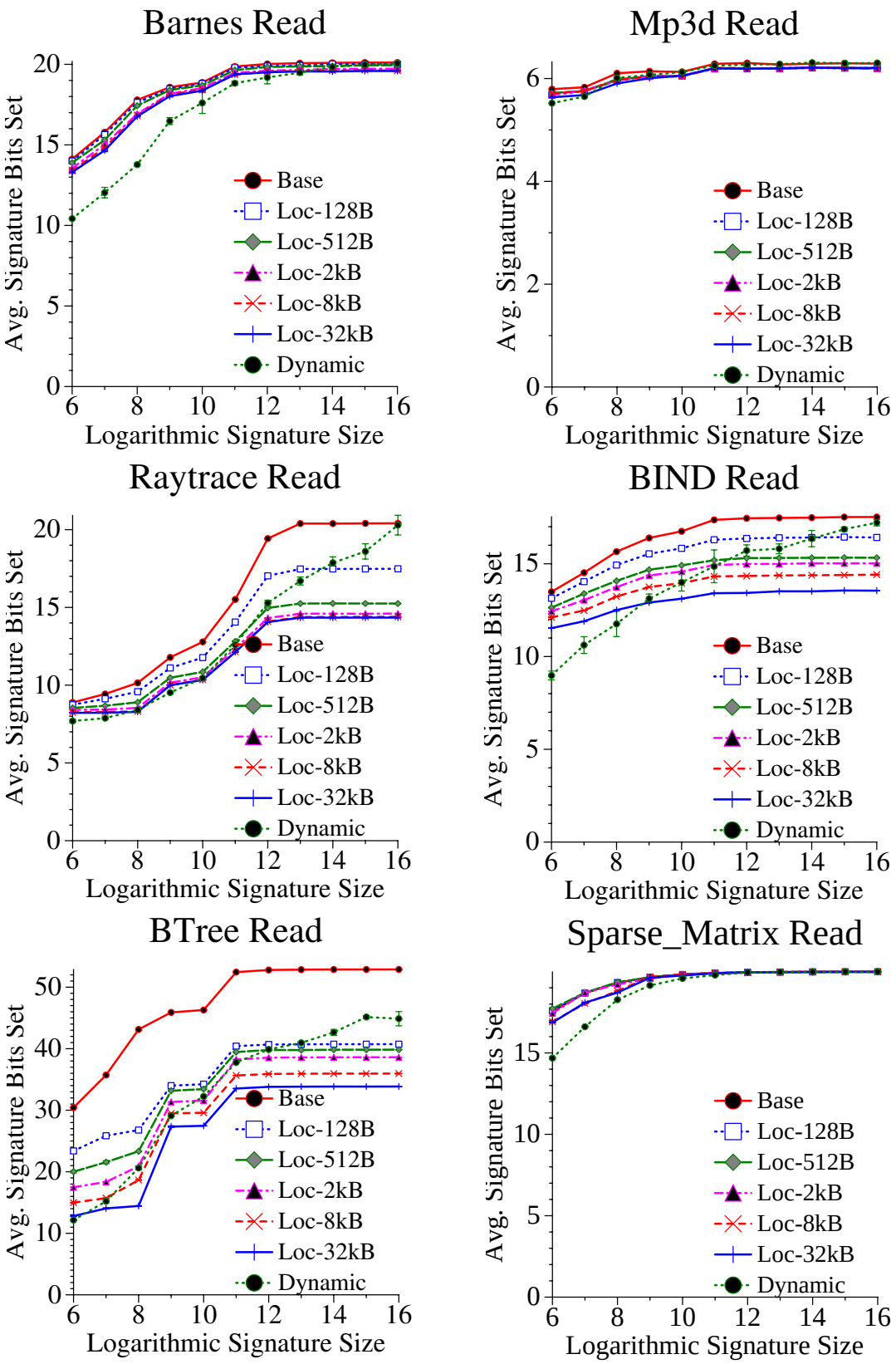


FIGURE 7-10. Average number of read signature bits for dynamic signatures. X-axis is logarithm base 2 of the signature size in bits

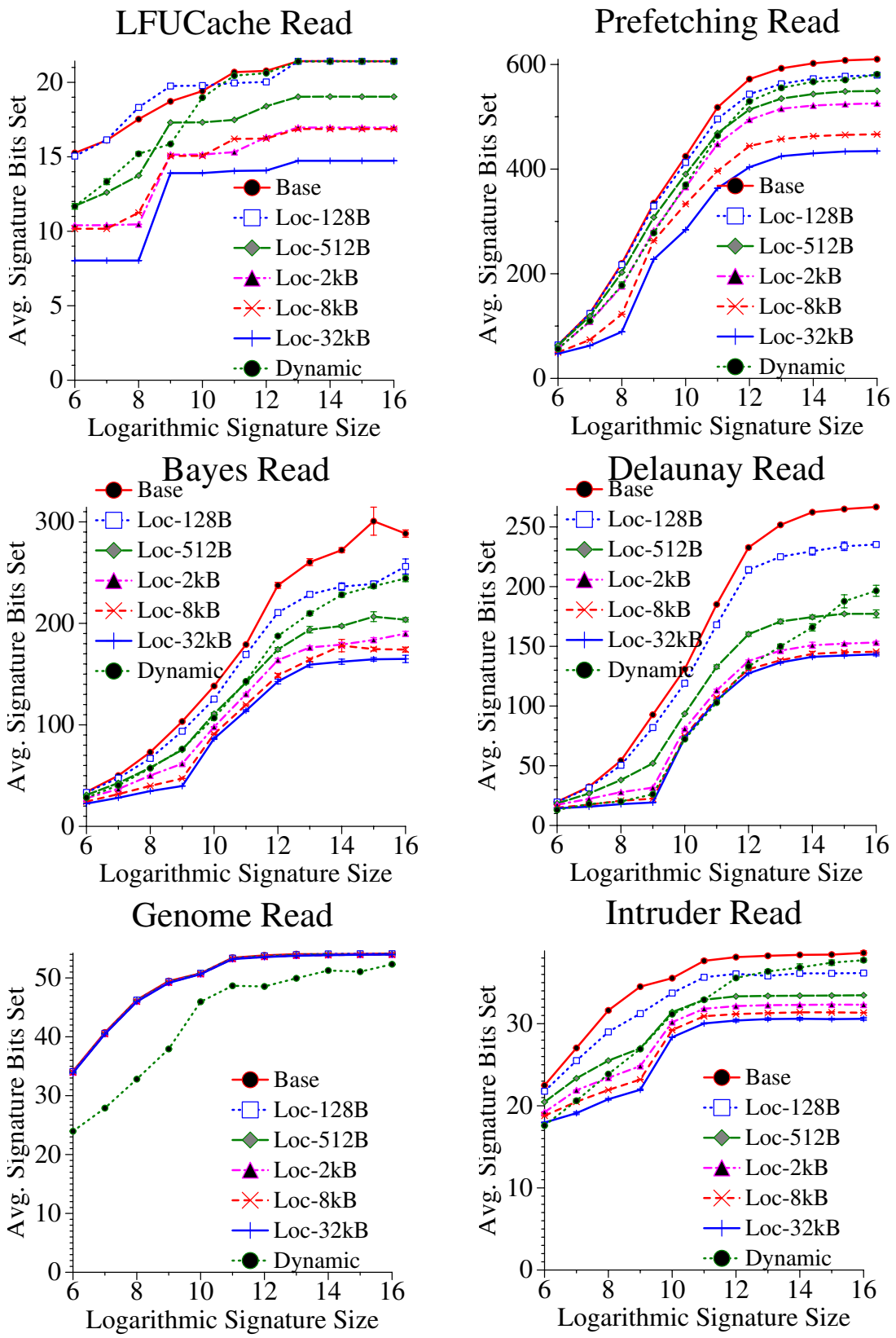


FIGURE 7-10. Average number of read signature bits for dynamic signatures. X-axis is logarithm base 2 of the signature size in bits

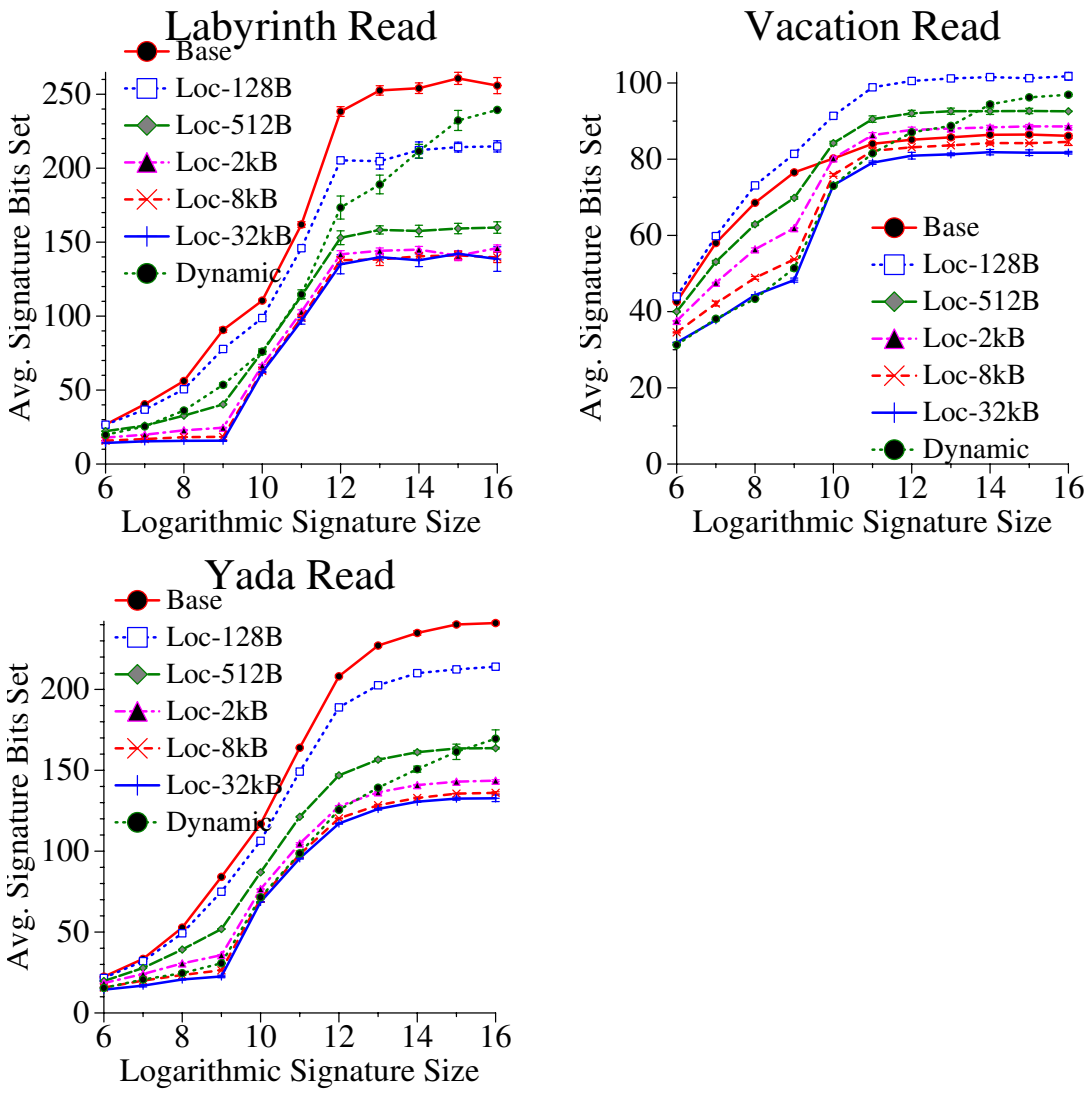


FIGURE 7-10. Average number of read signature bits for dynamic signatures. X-axis is logarithm base 2 of the signature size in bits

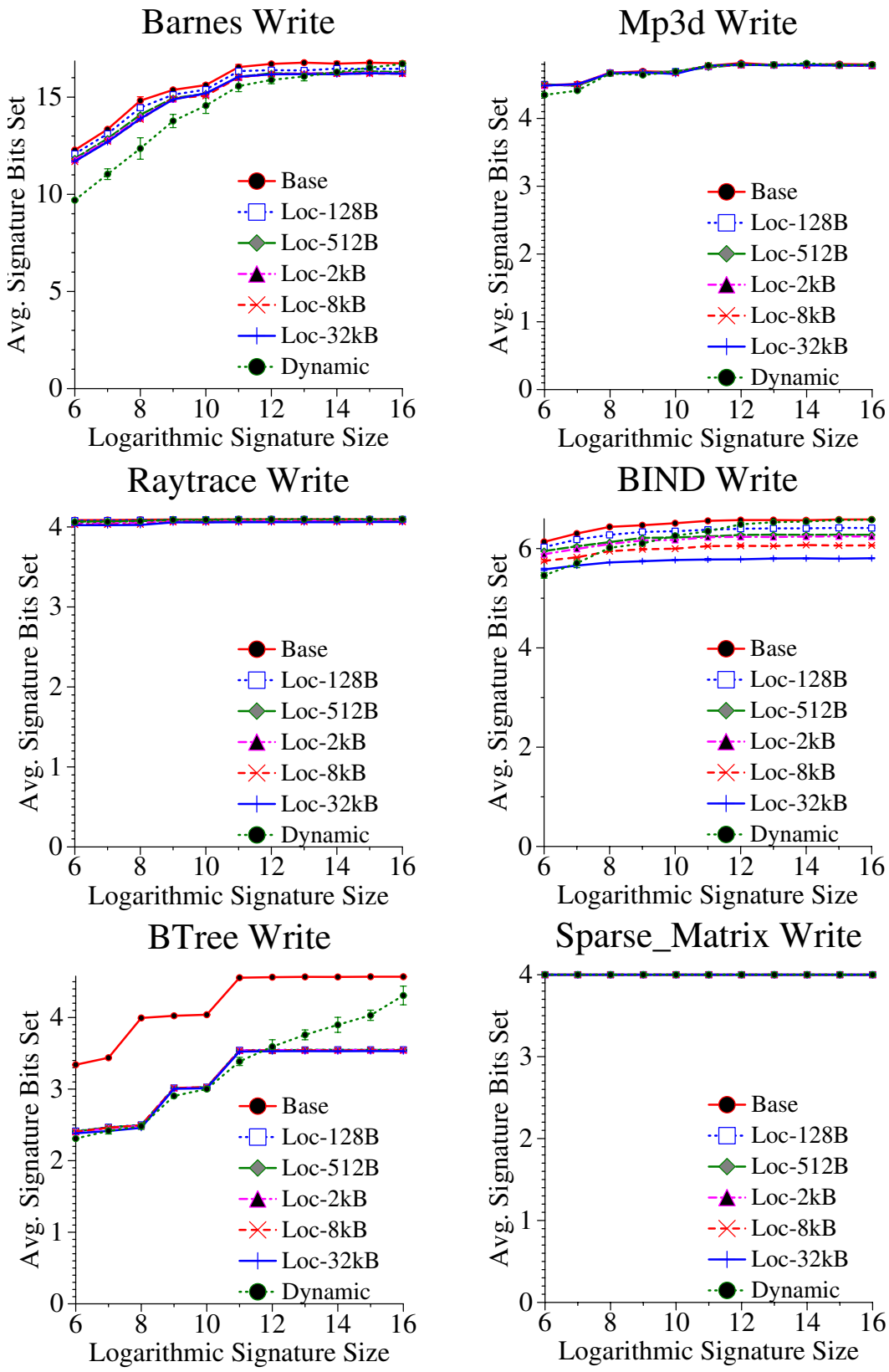


FIGURE 7-11. Average number of write signature bits for dynamic signatures. X-axis is logarithm base 2 of the signature size in bits

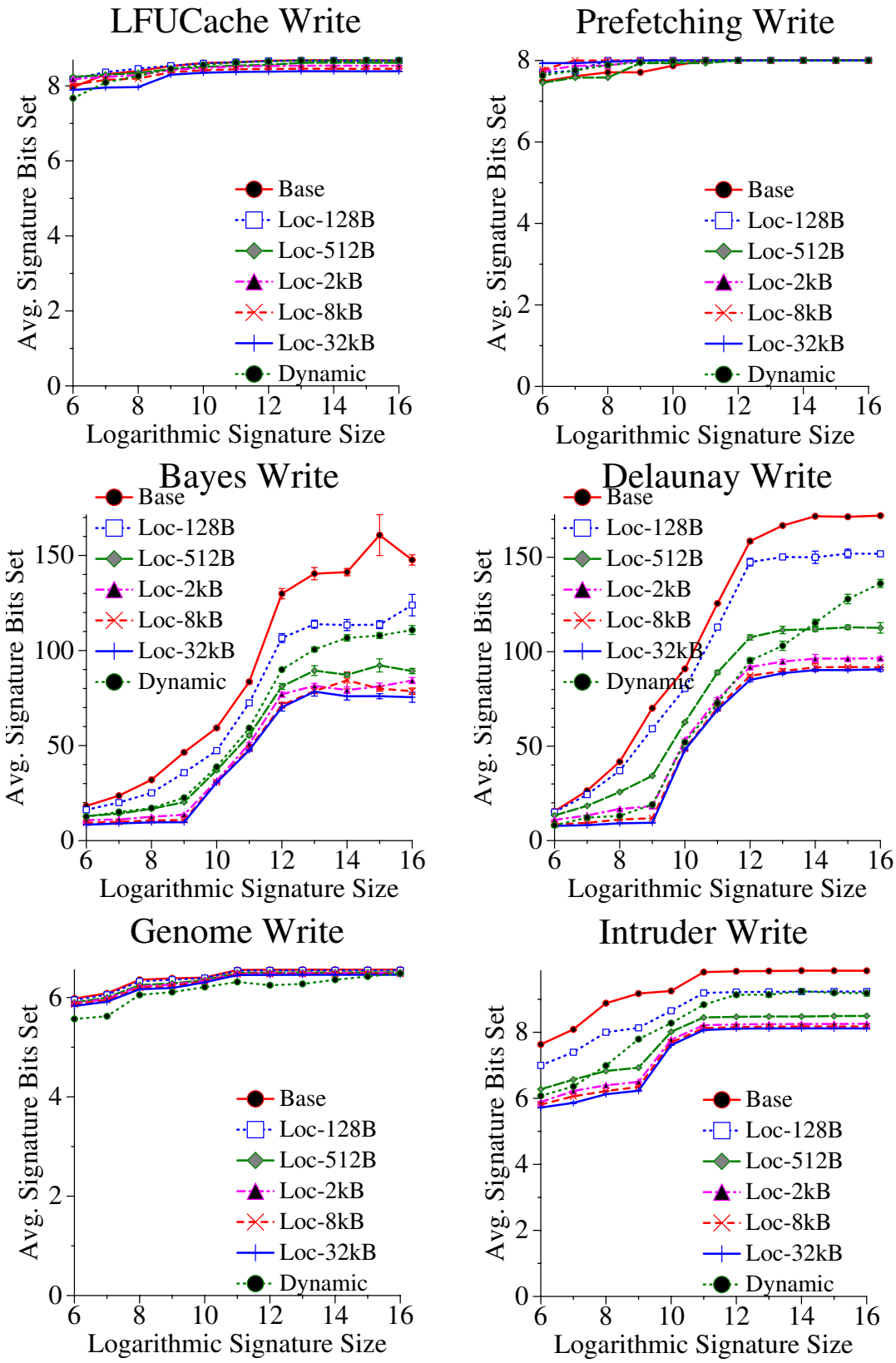


FIGURE 7-11. Average number of write signature bits for dynamic signatures. X-axis is logarithm base 2 of the signature size in bits

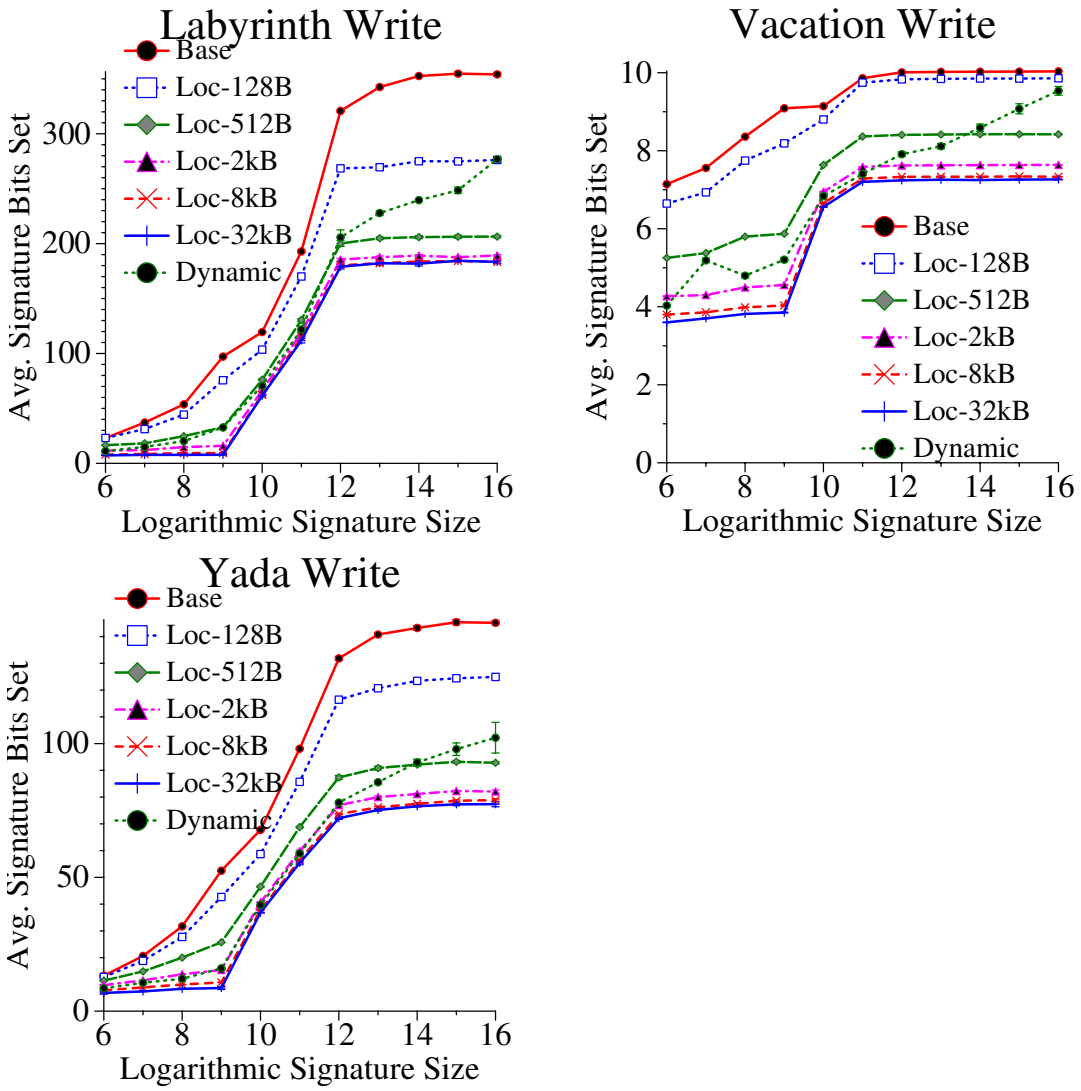


FIGURE 7-11. Average number of write signature bits for dynamic signatures. X-axis is logarithm base 2 of the signature size in bits

Results. Figure 7-9 shows the normalized execution times of dynamic signatures that use spatial locality (lines labeled "Dynamic Sp. Loc."). In order to explain the effectiveness of dynamic signatures we also graph the read- and write-set sizes of the signatures with this optimization (lines labeled "Dynamic"). For comparison purposes we also illustrate the results for static signatures with spatial locality and base configuration (remaining lines).

As Figure 7-9 shows, dynamic signatures are similar or better than Base for all workloads except Genome, LFUCache, and BTree. Dynamic signatures are able to adapt to the best spatial locality hashing granularity for the non-exceptional workloads. Figure 7-10 and Figure 7-11 support these trends, by illustrating the average number of signature bits set. For the majority of workloads the average number of bits set is bounded by the lowest number of bits set by the static signatures (usually at the 32kB granularity, line labeled "Loc-32kB") and by the original signature with no spatial locality (line labeled "Base"). However for some workloads (Barnes, BIND, Raytrace, Sparse Matrix, and Genome) dynamic signatures may set fewer signature bits than static signatures. The primary reason for this is because dynamic signatures optimizes all XIDs, regardless of the number of read- and write-set lines in them. In contrast, for static signatures we choose to optimize only those XIDs with the largest read- and write-sets.

Finally, we examine the execution time results for the exceptional workloads (Genome, LFUCache, and BTree). These workloads perform worse than the base signature configuration. We use our performance debugging framework to gather profiling data for these workloads. This data shows that the differences are due to higher numbers of conflicts for dynamic signatures compared to Base. In Genome, there are more RW requestor younger stalls (false conflicts). In LFUCache, there are more RW requestor younger stalls (true conflicts). Finally, in BTree there are

more RW requestor younger stalls (false conflicts) and more non-transactional stall cycles. We provide an intuitive explanation for these results. The policy used by dynamic signatures is to continually utilize the signature which yields the fewest positive lookups. This works for the majority of the workloads we studied. However this policy ignores information related to the control-flow and criticality of the stalls. Thus optimizing away certain stalls may perturb the order of conflict formation and therefore the criticality of other stalls. For the exceptional workloads this phenomenon manifests as additional RW requestor younger stalls compared to the base signature.

In summary, dynamic signatures continually adapt to the best hashing granularity. It yields similar or better execution times compared to the base signature and static signatures. However some workloads perform worse. This is likely due to negative interactions with the policy of dynamic signatures which affects the criticality of stalls. Finally, it is not clear whether the performance benefits justify its additional hardware complexity (using a set of signatures and performance counters).

7.3.5 Coarse-fine Hashing

Description. The main idea behind coarse-fine hashing is to combine characteristics of two types of bit fields to try to exploit spatial locality. In static and dynamic signatures some number of low-order address bits are skipped before using the address for insertion or lookups. We name the low-order bits that are skipped the *fine* bits, because they convey fine-grain memory location information (e.g., cache block within a memory page). The remaining high-order bits are named the

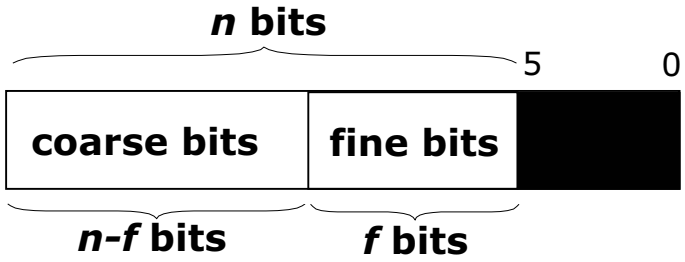


FIGURE 7-12. Bit-fields used for coarse-fine hashing. Low-order *f* bits are the fine bits, and the remaining *n-f* bits are the coarse bits. Coarse-fine allows zero or more hash functions to operate on each bit region.

coarse bits (e.g., memory page number). Figure 7-12 illustrates these bit-fields in an address, and we refer to this figure in our subsequent discussions.

Coarse-fine hashing, like both static and dynamic signatures, attempts to reduce the number of signature bits set by exploiting spatial locality. However, coarse-fine hashing takes a different approach. In coarse-fine hashing, zero or more hash functions can operate on the *f* fine bits (we name these the fine hash functions), and the remaining hash functions operate on the *n-f* coarse bits (we name these the coarse hash functions).

At a high-level coarse-fine combines the characteristics of fine and coarse bits in setting signature bits. The fine bits are used to distinguish amongst different addresses (and mitigate the probability of false positives from setting too few signature bits). The coarse bits are used to discern memory regions (and mitigate the probability of false positives from a fully utilized signature).

Coarse-fine hashing has two main configurable parameters. The first is the number of low-order address bits that denotes the fine bits. This is the variable *f* in Figure 7-12. If *f* is zero, coarse-fine will perform similar to the original PBX hashing scheme. The second is the number of hash functions devoted to coarse and to fine bits. In general, the more hash functions devoted to

the fine bits the more signature bits will be set. We see exceptions to this in our results, because many hash functions operating on a small number of fine bits can lead to correlation in the hash values. Correlations reduce the range of hash values and thus number of signature bits that are set.

Implementation. The hardware implementation is similar to static signatures and requires lower hardware than dynamic signatures. The parameters can be hard-wired, by statically setting the *f* value and choosing a fixed number to be fine hashes. A dynamic version of coarse-fine can be implemented by adding programmable logic for setting *f* and the number of fine hash functions. Additional performance counters (e.g., number of signature lookup hits, similar to dynamic signatures) can be added to provide feedback for a dynamic coarse-fine implementation. We examine some representative parameter values for coarse-fine hashing in order to illustrate interesting trends. We choose two fine bit regions, 2kB and 16kB (i.e., *f* is 5 bits and 8 bits), and vary the number of fine hash functions from zero to three. Similar to PBX, each hash function (fine and coarse) XORs two bit-fields within the fine or coarse bit region, depending on which type it is. These two bit-fields are chosen using a simple interleaving algorithm that takes into account the number of hash functions of each type (to avoid choosing the same bits for the same type of hash). In addition, the algorithm avoids choosing the same bits for the two bit-fields (by starting the bit selection from opposite directions). Section 7.5 describes how entropy can be applied to coarse-fine hashing in order to dynamically fine-tune its parameters.

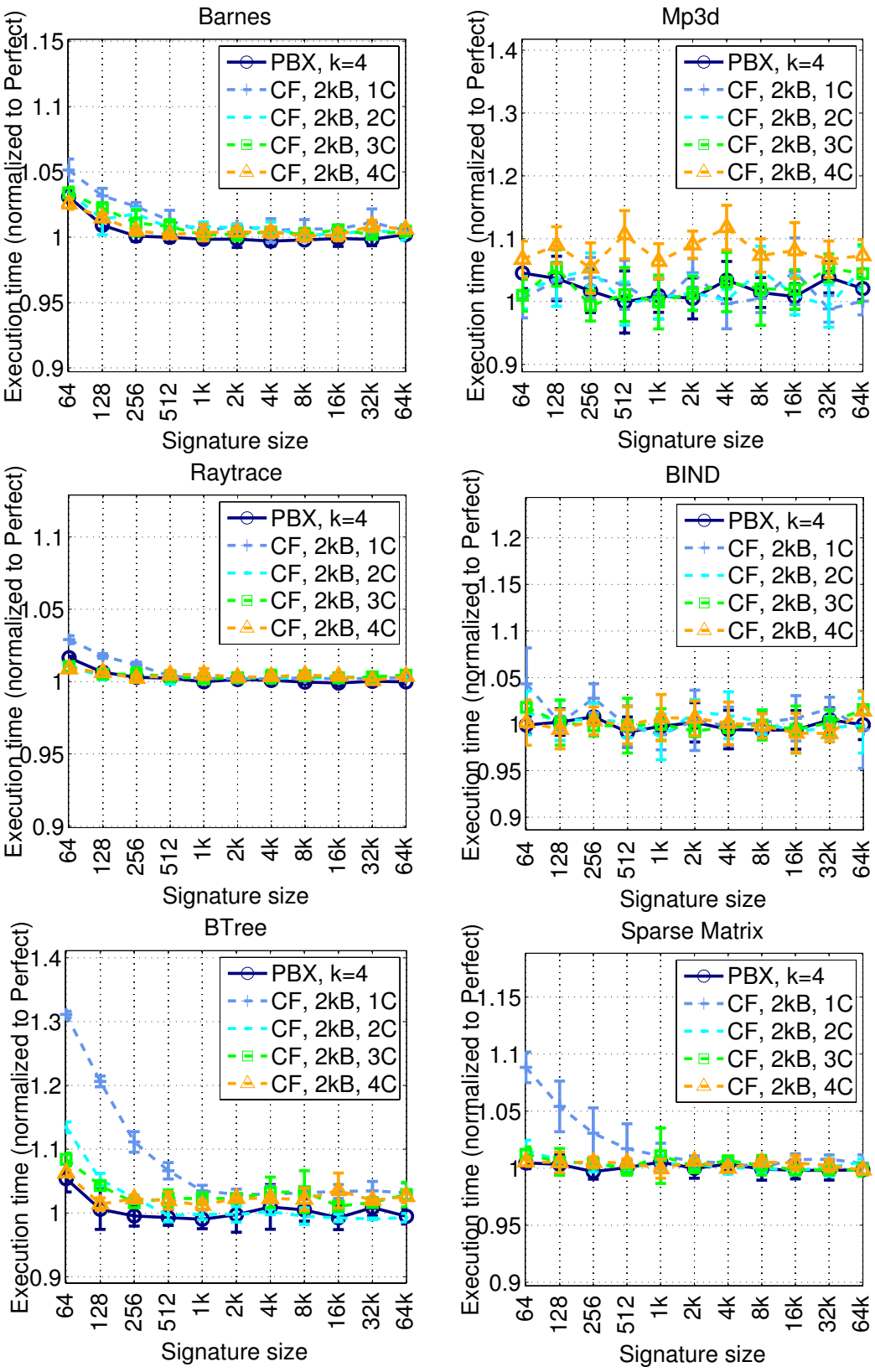


FIGURE 7-13. Normalized execution times with coarse-fine hashing with 2kB fine regions

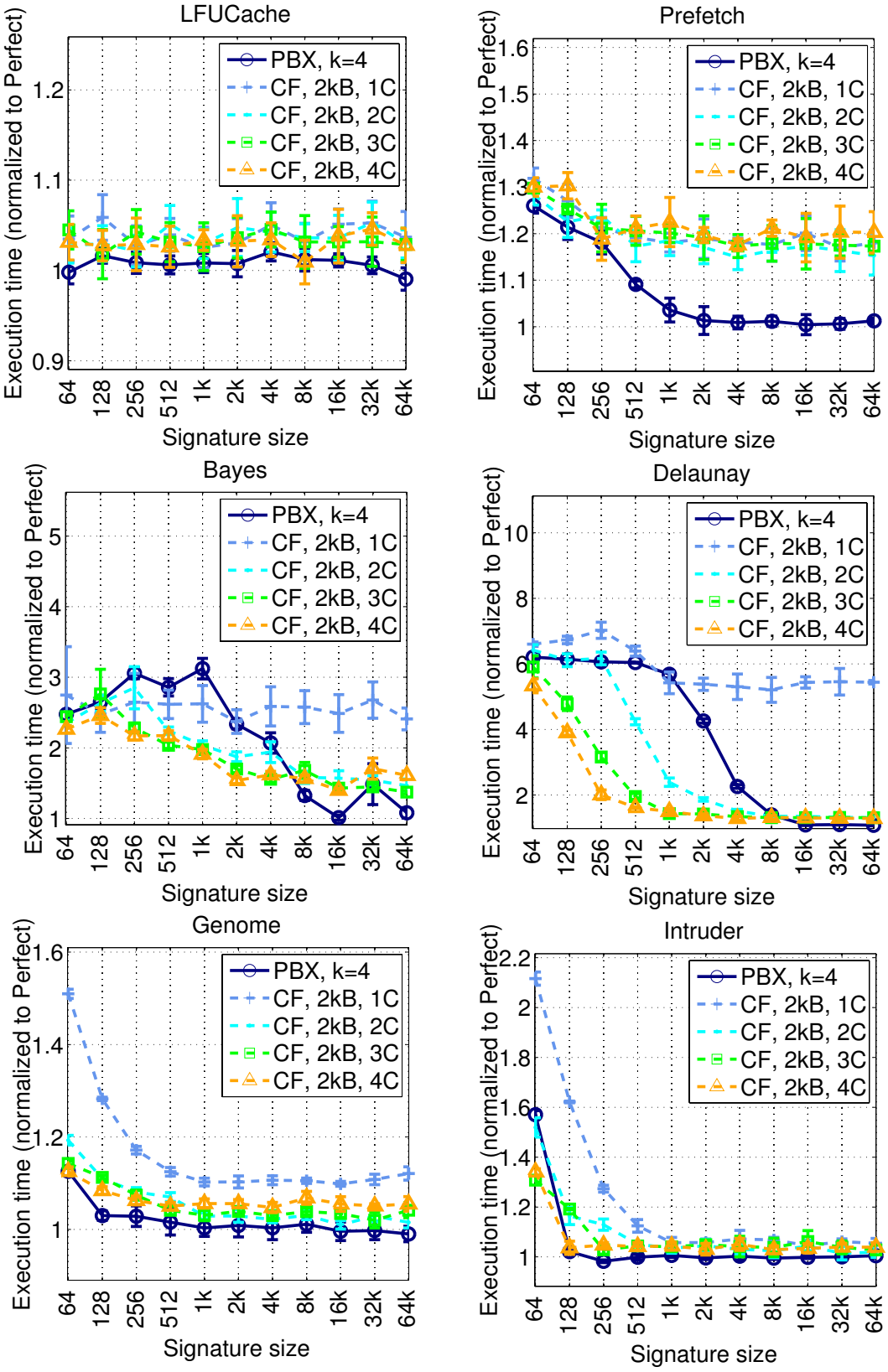


FIGURE 7-13. Normalized execution times with coarse-fine hashing with 2kB fine regions

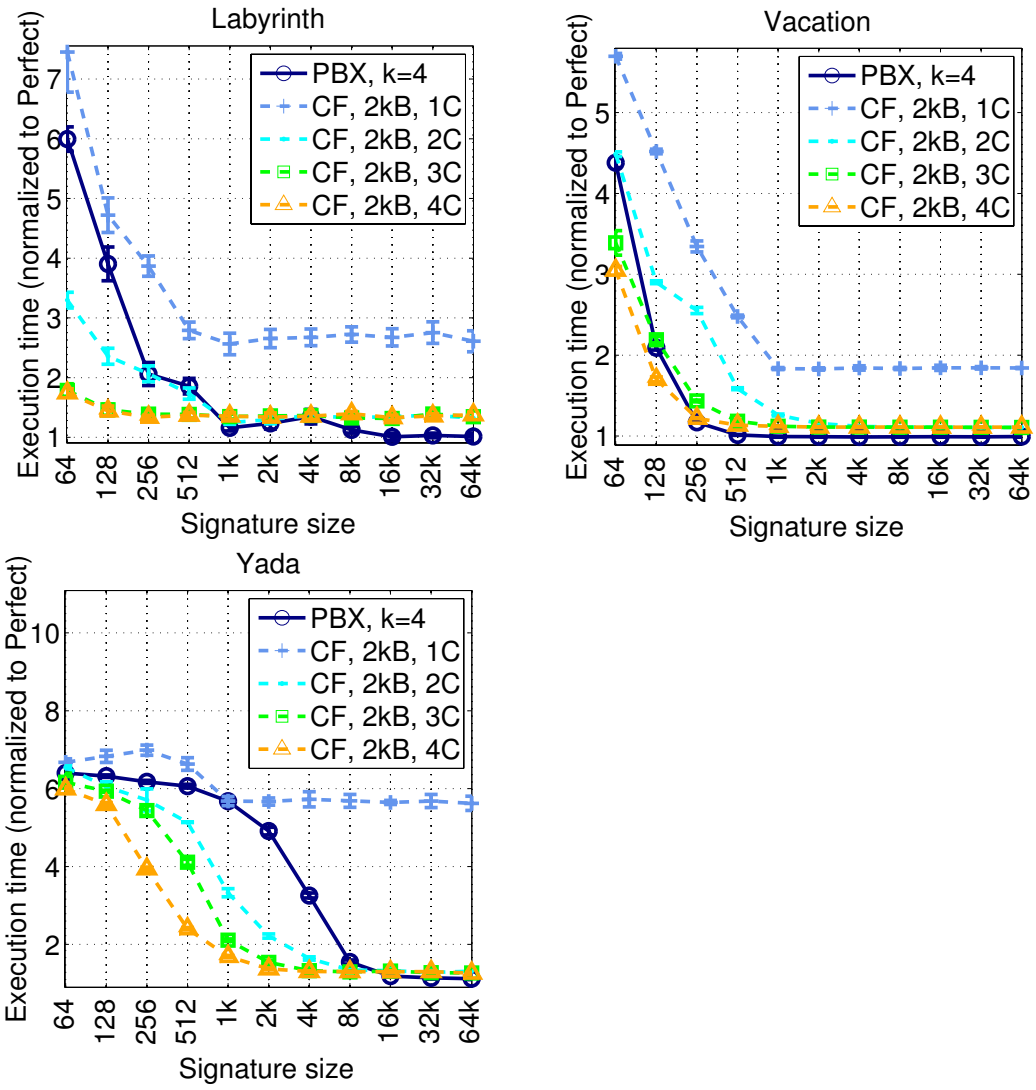


FIGURE 7-13. Normalized execution times with coarse-fine hashing with 2kB fine regions

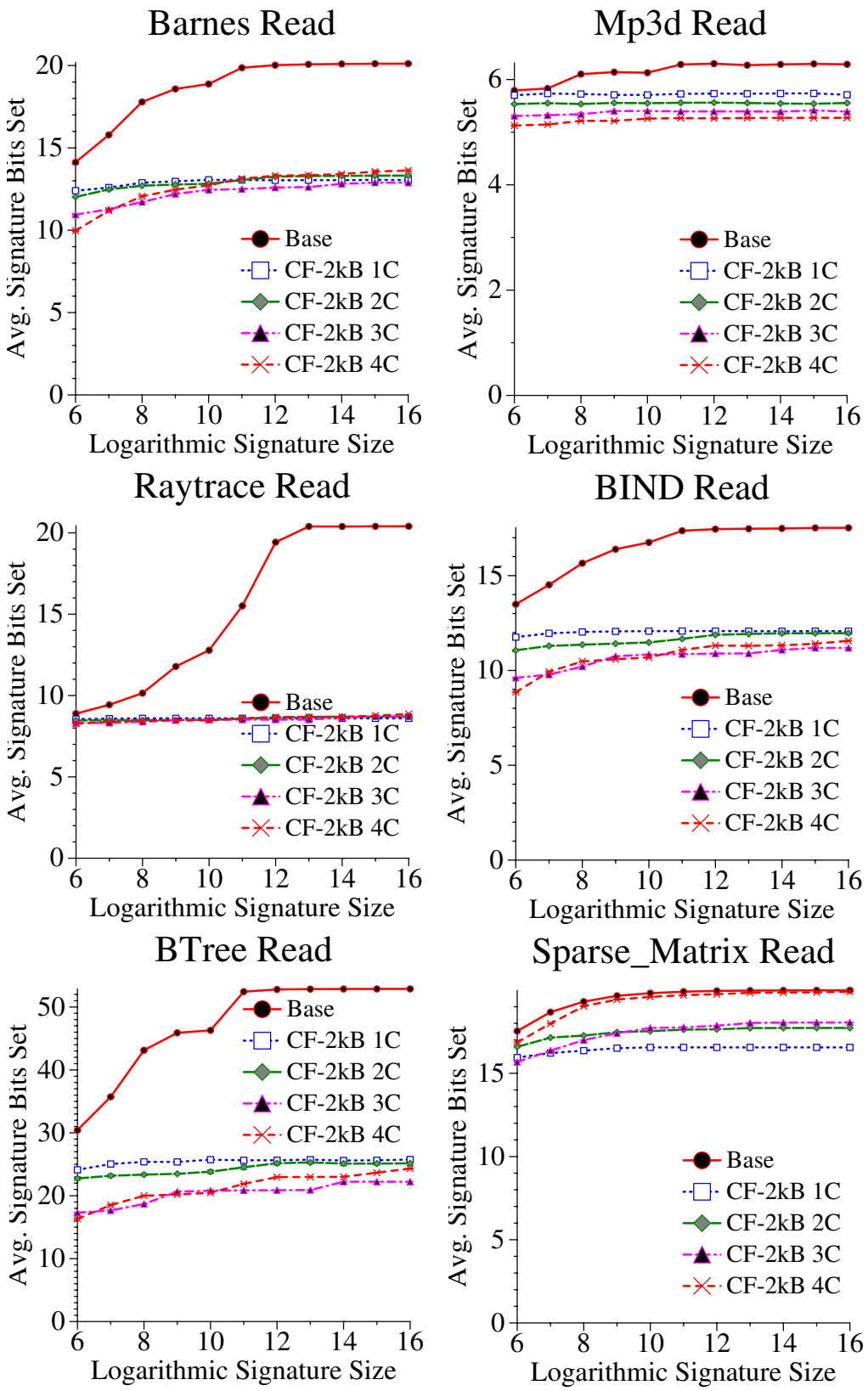


FIGURE 7-14. Average number of read signature bits set for coarse-fine hashing with 2kB fine regions. X-axis is the logarithm base 2 of the signature size in bits

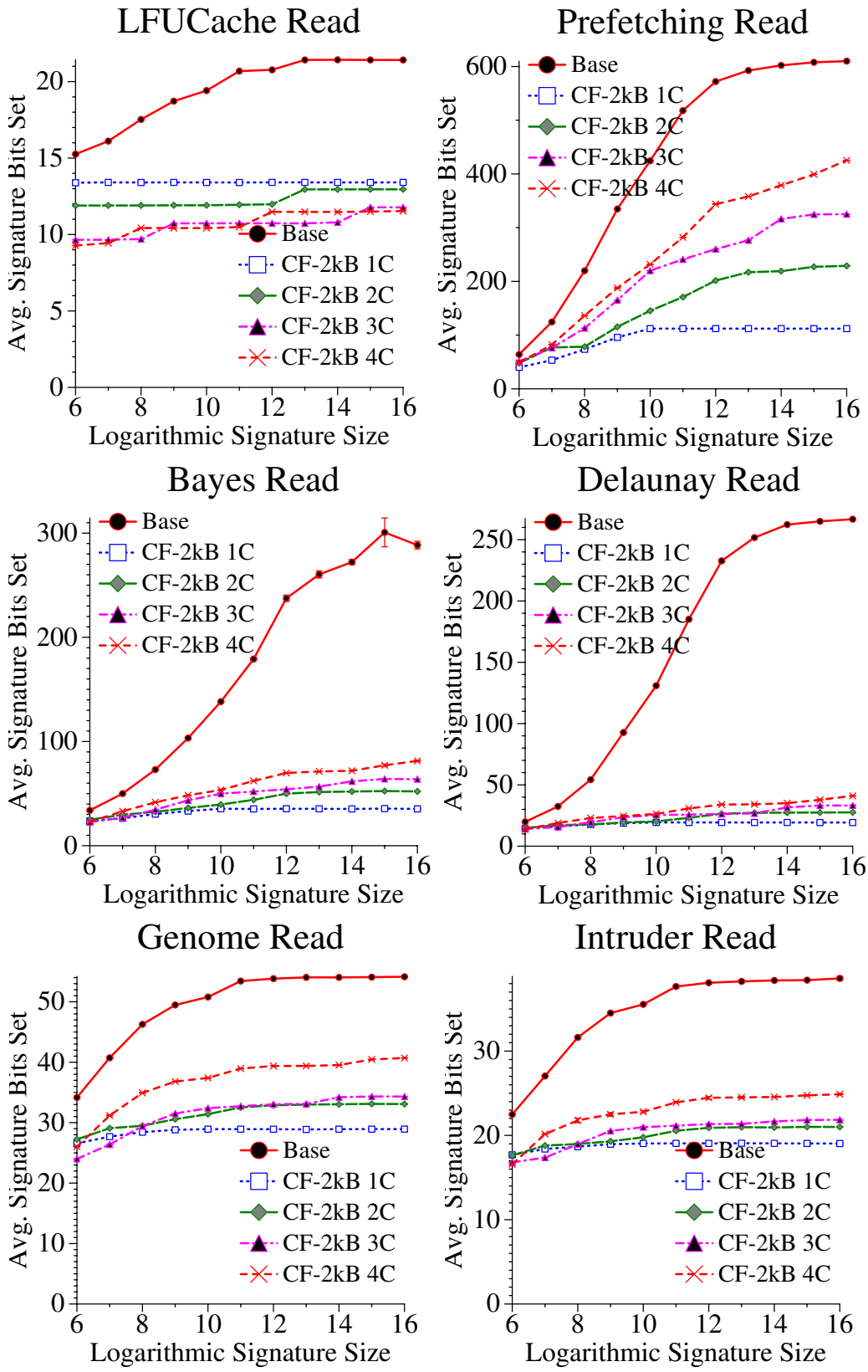


FIGURE 7-14. Average number of read signature bits set for coarse-fine hashing with 2kB fine regions. X-axis is the logarithm base 2 of the signature size in bits

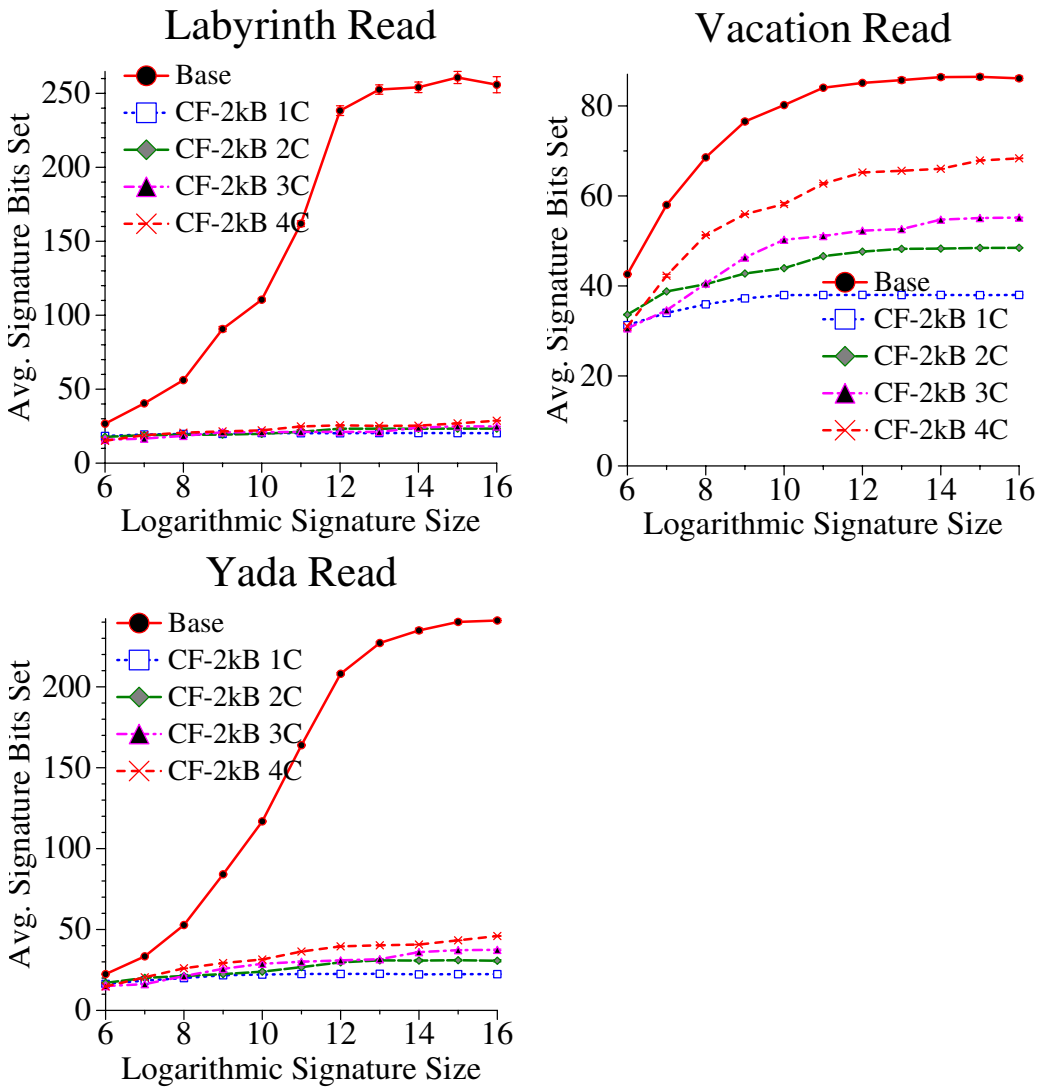


FIGURE 7-14. Average number of read signature bits set for coarse-fine hashing with 2kB fine regions. X-axis is the logarithm base 2 of the signature size in bits

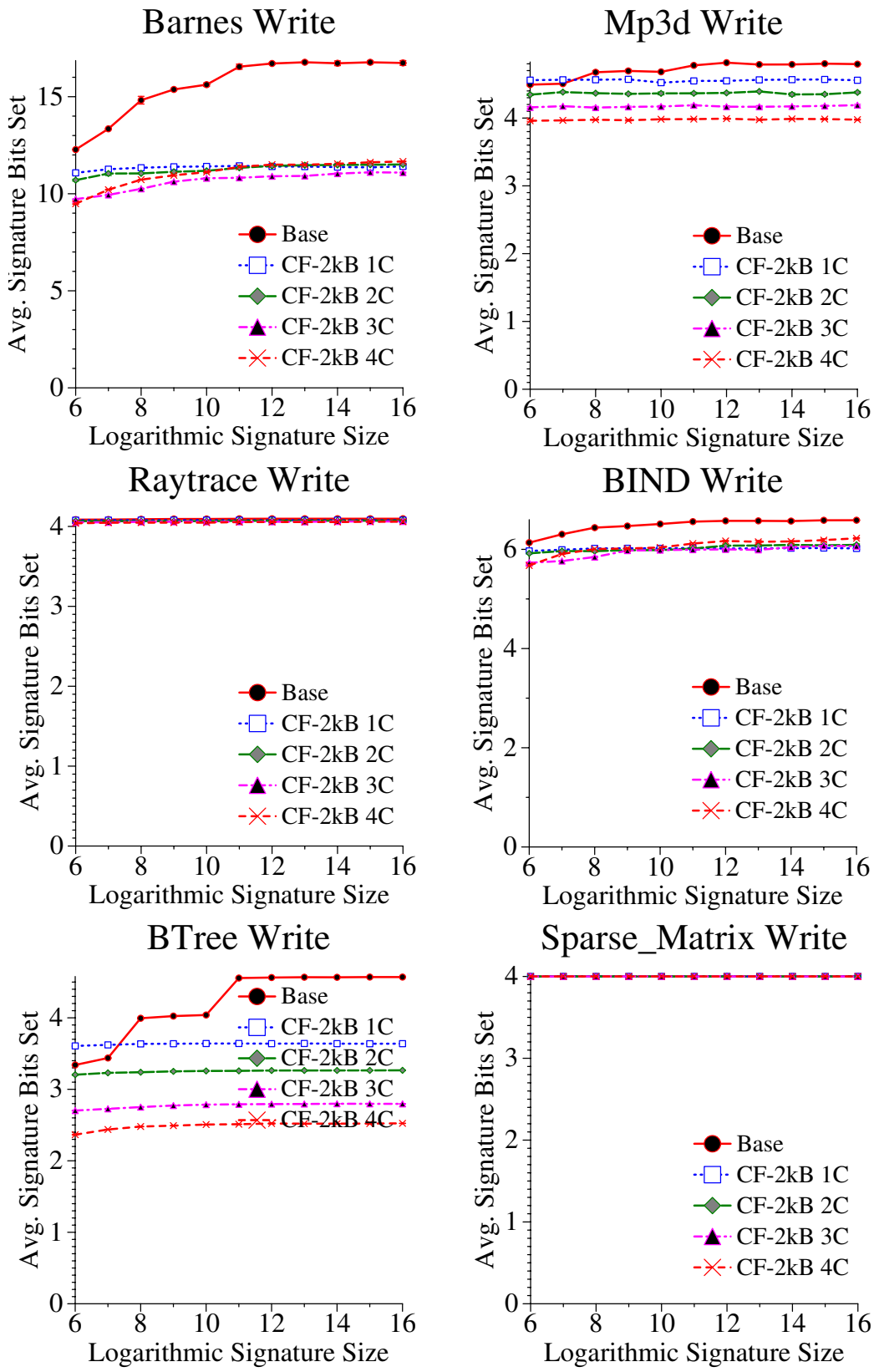


FIGURE 7-15. Average number of write signature bits set for coarse-fine hashing with 2kB fine regions. X-axis is the logarithm base 2 of the signature size in bits

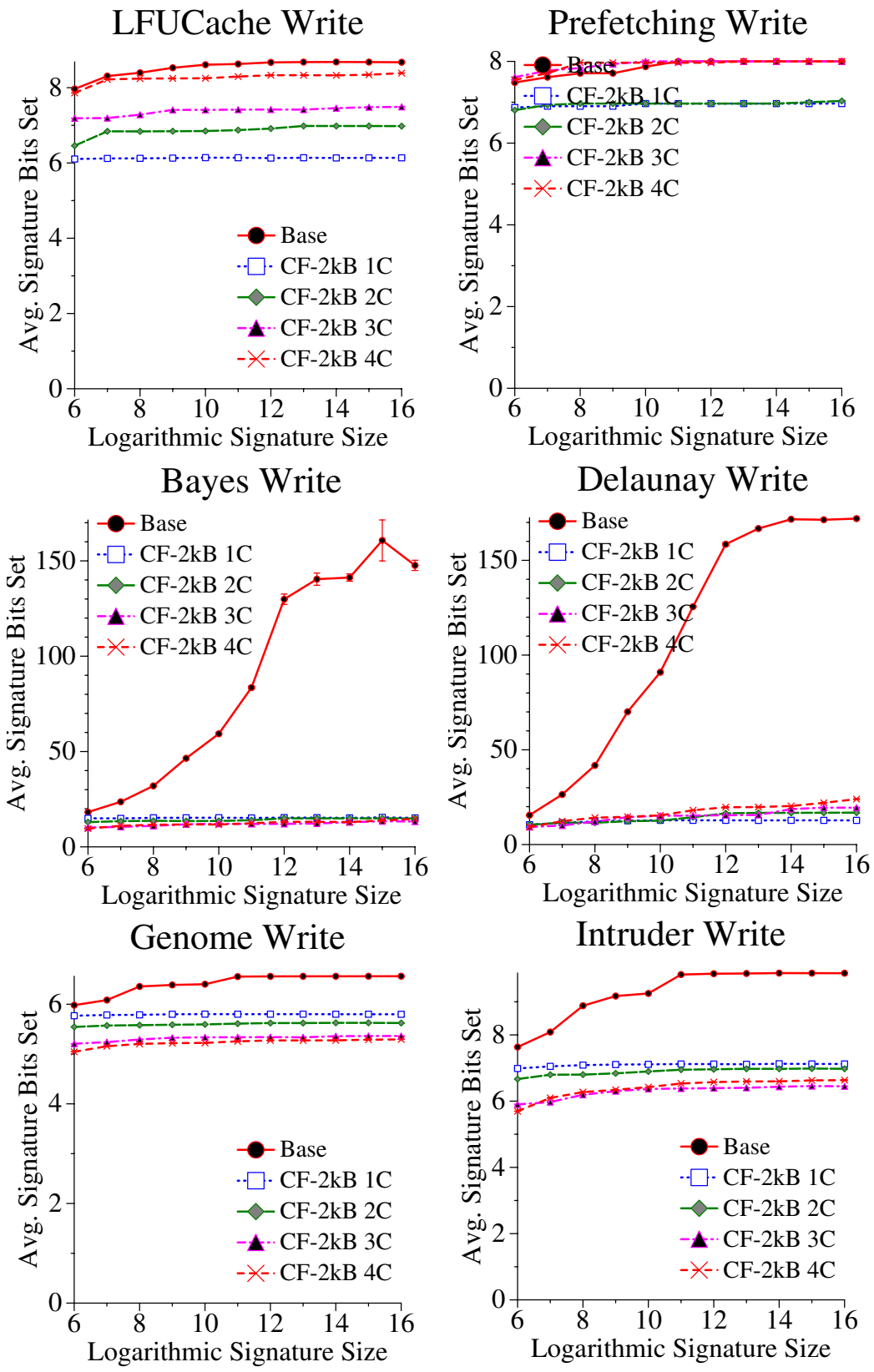


FIGURE 7-15. Average number of write signature bits set for coarse-fine hashing with 2kB fine regions. X-axis is the logarithm base 2 of the signature size in bits

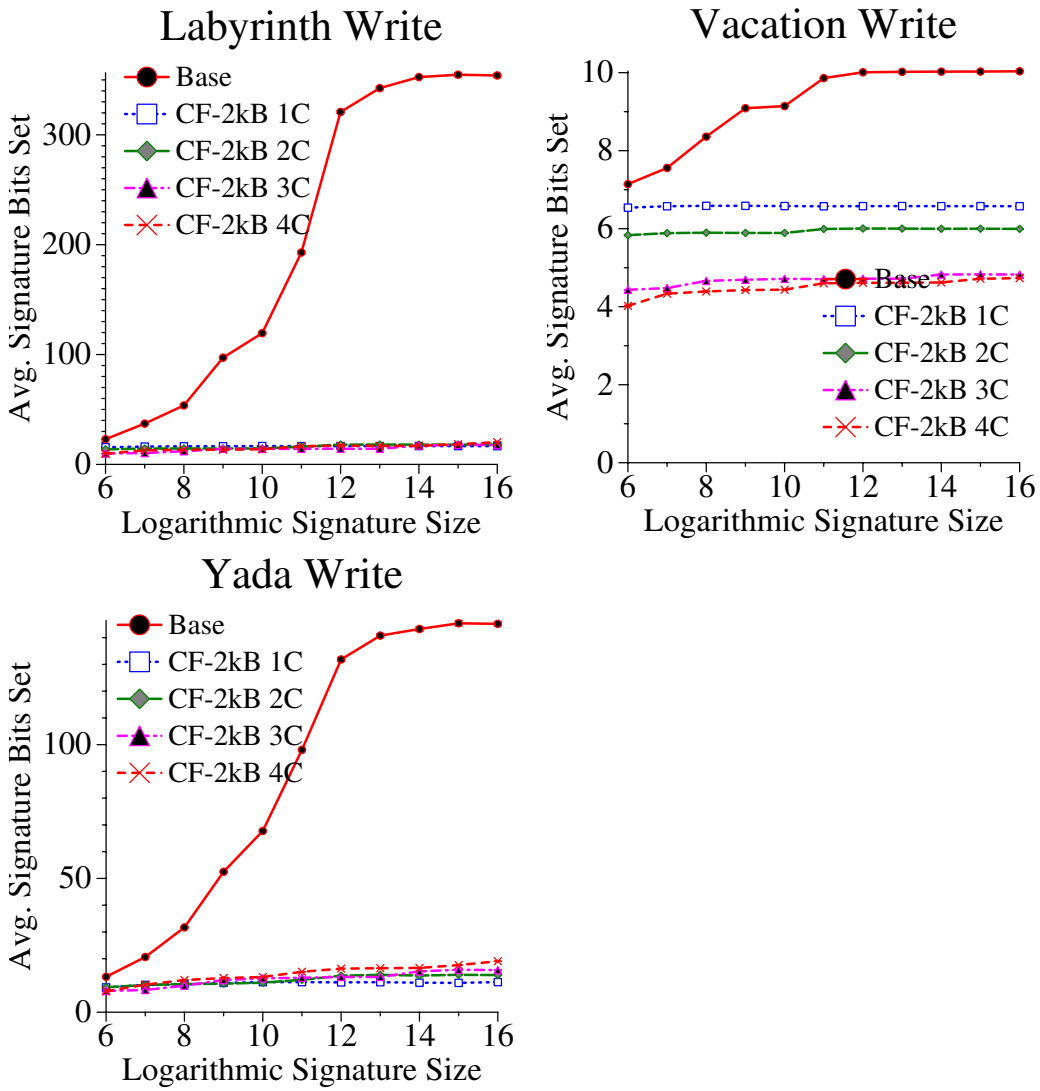


FIGURE 7-15. Average number of write signature bits set for coarse-fine hashing with 2kB fine regions. X-axis is the logarithm base 2 of the signature size in bits

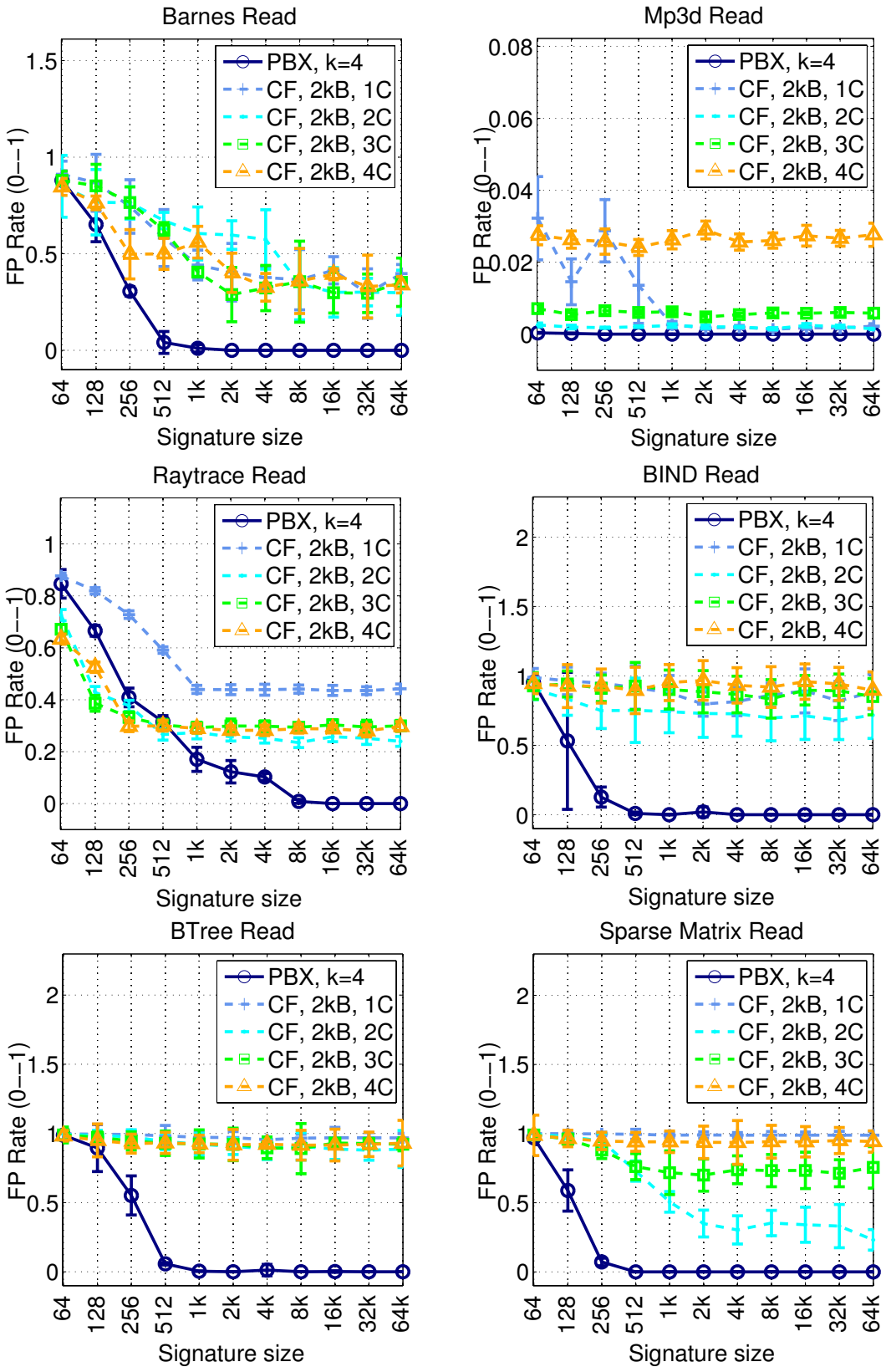


FIGURE 7-16. Average false positive rate of read signature using coarse-fine hashing

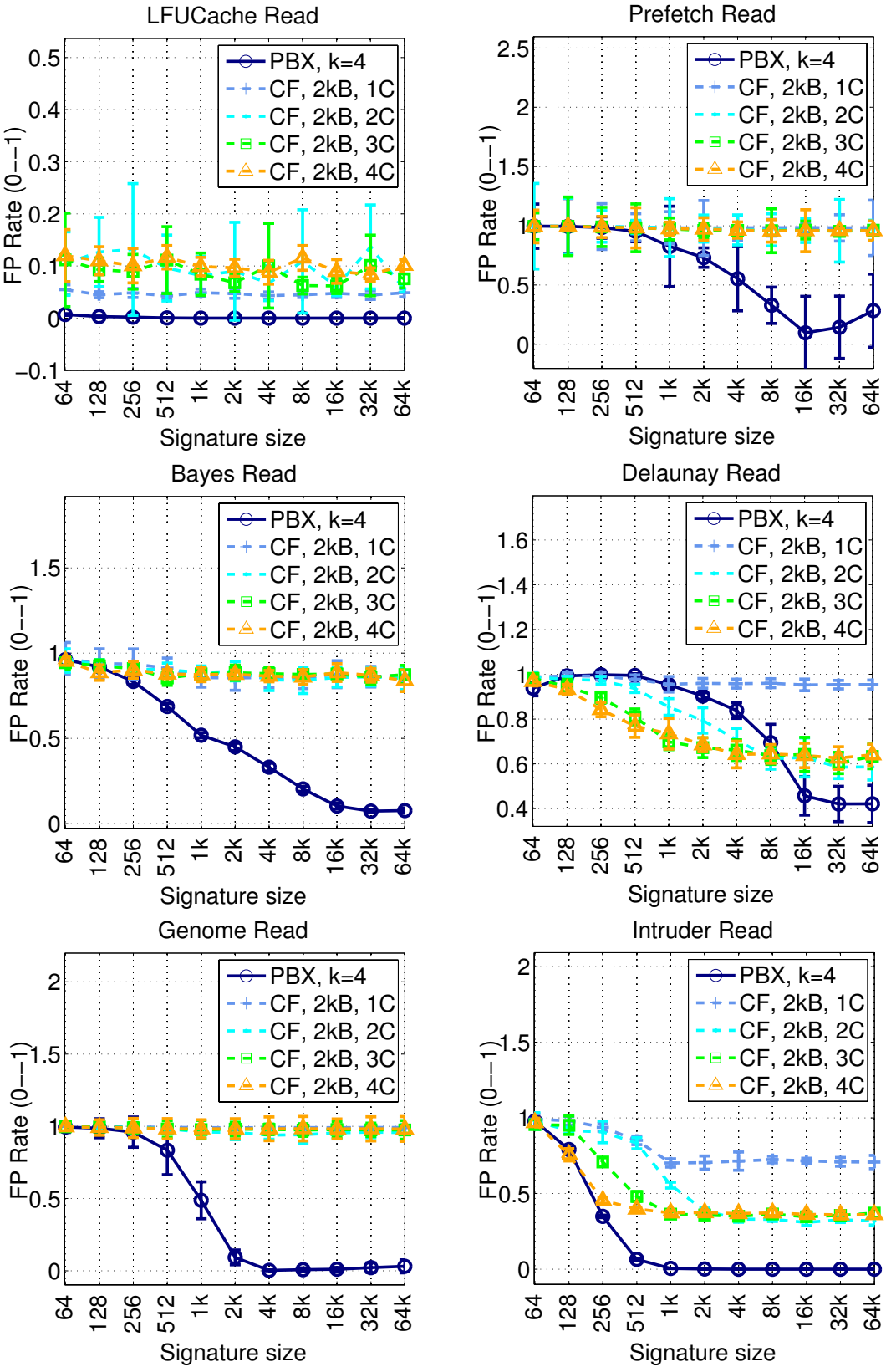


FIGURE 7-16. Average false positive rate of read signature using coarse-fine hashing

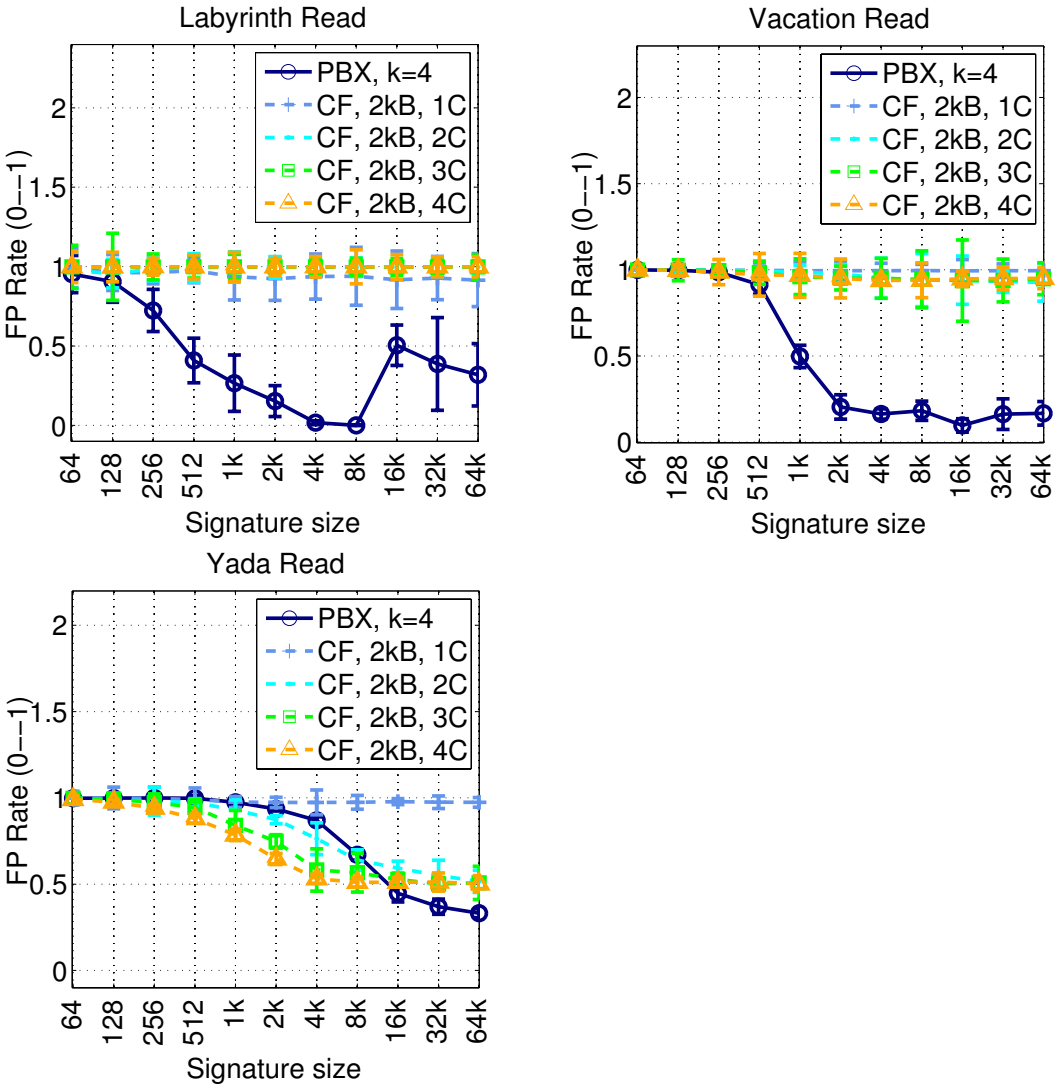


FIGURE 7-16. Average false positive rate of read signature using coarse-fine hashing

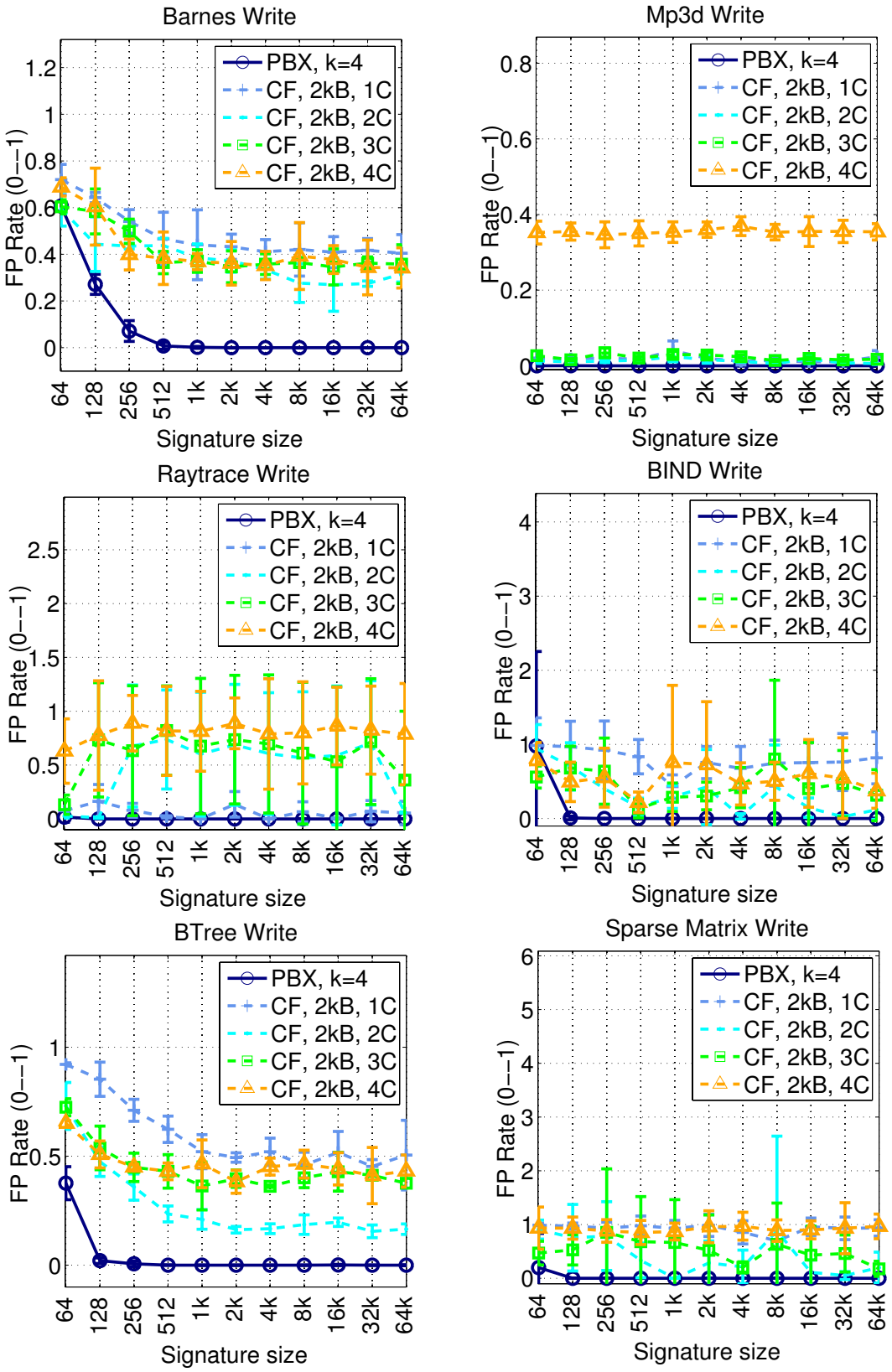


FIGURE 7-17. Average false positive rate of write signature using coarse-fine hashing

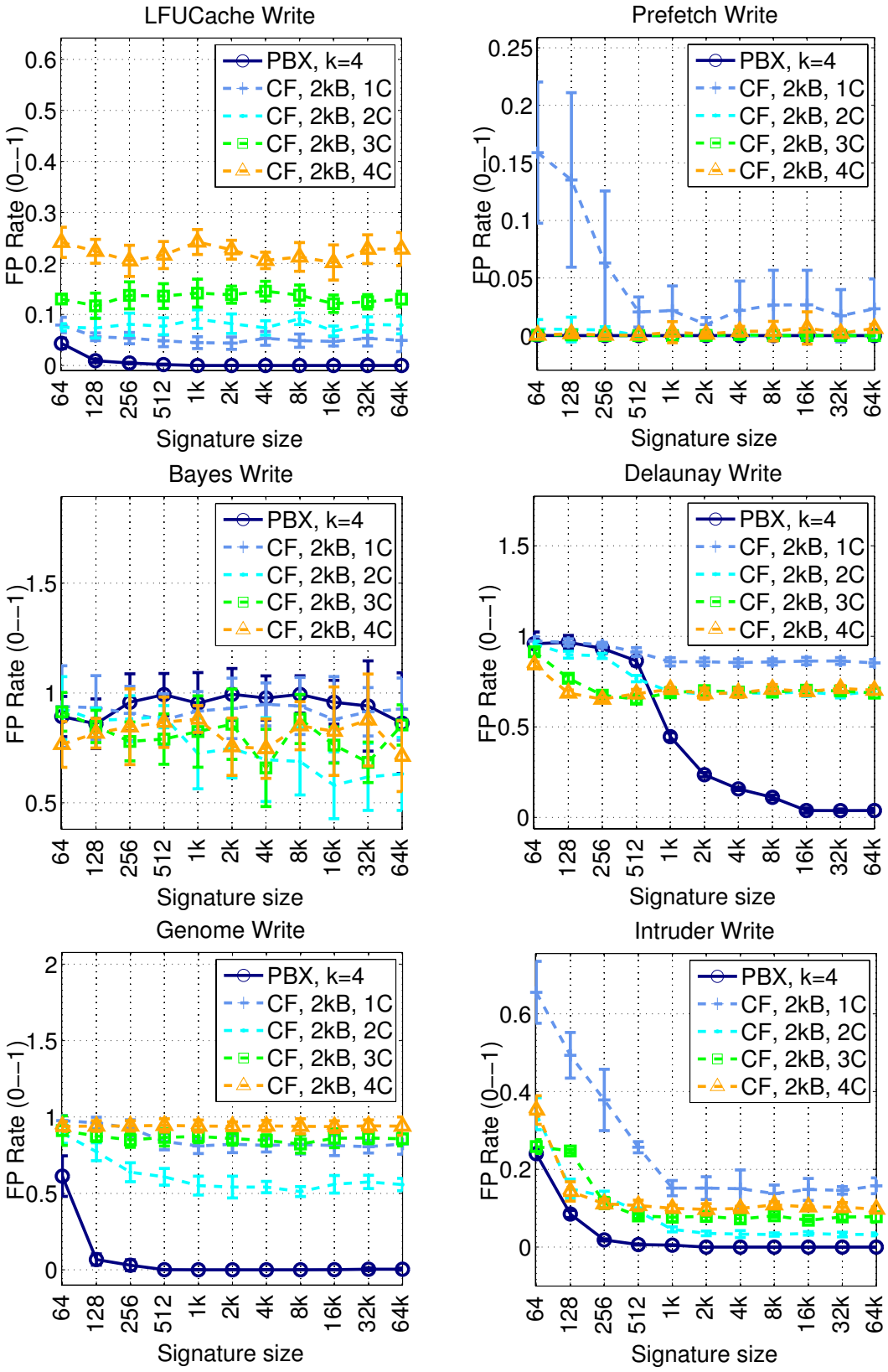


FIGURE 7-17. Average false positive rate of write signature using coarse-fine hashing

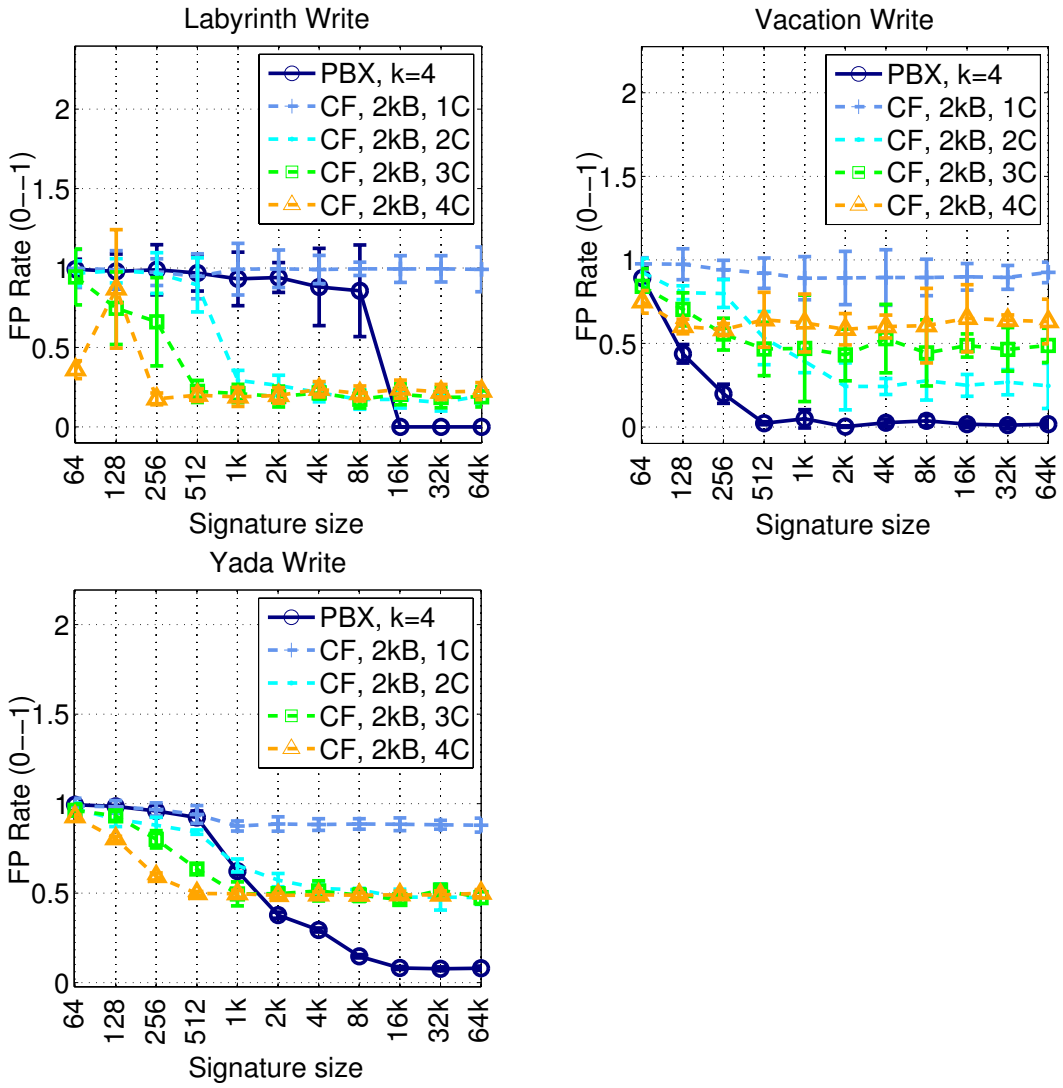


FIGURE 7-17. Average false positive rate of write signature using coarse-fine hashing

Results with 2kB fine bit regions. We first provide the results for the 2kB fine bit region, and then follow with results for the 16kB fine bit region. Figure 7-13 shows the normalized execution time for coarse-fine hashing with 2kB fine bit regions (lines labeled "CF"). Figure 7-14 illustrates the average number of read signature bits set, and Figure 7-15 shows the average number of write signature bits set.

Unlike static signatures, coarse-fine hashing allows some number of hash functions to operate on the fine bits. This complicates a straightforward analysis with the existing graphs since there are now more complex memory interleavings (primarily from different memory request stalls) compared to static signatures. We found the most intuitive (and best illustration) of these subtle effects is to examine the false positive rate of the read and write signatures. Figure 7-16 and Figure 7-17 show the average false positive rates (between 0 and 1, with 0 indicating no false positives) of the read and write signatures, respectively.

In general, coarse-fine hashing significantly reduces the number of bits set in both the read and write signatures (Figure 7-14 and Figure 7-15). However, only six workloads (Bayes, Delaunay, Intruder, Labyrinth, Vacation, and Yada) show some benefit when coarse-fine hashing is used. For the beneficial workloads the reduction in execution time comes from an improvement in the false positive rate of one or both signatures. For example, in Delaunay and Yada coarse-fine hashing reduces the false positive rate of the read and write signatures of the 2C, 3C, and 4C configurations (signatures which use two to four coarse hash functions). In Bayes and Labyrinth the false positive rate of the write signatures is reduced in several of the 2C, 3C, and 4C configurations.

Finally, Intruder and Vacation exhibit exceptional behavior. Although they benefit from coarse-fine hashing, none of the existing graphs explains the root causes. In fact the signature have higher false positives when coarse-fine hashing is used compared to the base signature configurations. Our performance debugging framework sheds light on this phenomenon. Coarse-fine hashing perturbs the conflict formation ordering and leads to an overall reduction in RW requestor younger false stalls, and reduces the overall cycles spent processing aborts. These stalls are likely on the critical-path, and their removal improve overall execution time.

Results with 16kB fine bit regions. We now summarize our results when the fine bit region is increased from 2kB to 16kB. The differences comes in improvements in the execution times of the 1C and 2C coarse-fine configurations. For these configurations there are an increased number of signature bits set compared to the 2kB results. Furthermore there is a corresponding reduction in the false positive rates of the read and write signatures.

In summary, coarse-fine hashing may yield improvements in overall execution times. This is driven by reductions in signature bits set and decreased false positive rates. The analysis of coarse-fine hashing is more complex than static signatures. This can be attributed to the design of coarse-fine hashing, which allows a mixture of coarse and fine hashes on the input address bits. This characteristic allows for a potentially wider range of conflicts (and associated interactions) than observed under static signatures.

7.4 Dynamic Re-hashing

7.4.1 Motivation

Signature false positives arise when two or more addresses map to the same bit(s), and a test operation indicates the presence of an address when it has not been stored in the signature. However, these false positives are the artifact of hashing behavior, and may disappear if the hash functions are altered to remap the conflicting addresses to different bit(s). In particular, our proposed technique of dynamic re-hashing uses the following insight: **Persistent false positives may be converted into *transient* false positives if the hash function is dynamically changed on transaction commit and/or abort.** TM systems that perform signature intersection to detect conflicts (e.g., Bulk) make dynamic re-hashing more challenging to implement, since re-hashing now involves external processors. On the contrary re-hashing is a local event in other TM systems that use signature test operations (e.g., LogTM-SE, SigTM).

7.4.2 Implementing Dynamic Re-hashing

Three design parameters to consider when implementing dynamic re-hashing for signatures include hardware mechanisms, re-hashing multiple hash functions, and the frequency of re-hashing.

Hardware mechanisms. In the base (static) PBX implementation, the input bits to the XOR hash functions are statically assigned. However, dynamic re-hashing can be implemented by permuting one or both of the selected input bit-fields *before* feeding these bits to the XOR gates. Timing and area constraints preclude complex permutations, thus our dynamic PBX signature implementations use bit rotation, which is simple but does a good job of permuting the bits. In our

signature configurations we assume only one bit-field is rotated (the cache-index bit-field), which reduces the hardware complexity of the rotation hardware.

Re-hashing multiple hash functions. Our signatures use four hash functions, and implementing re-hashing for all hashes may increase hardware overheads. Thus re-hashing a subset of hash functions is an attractive option. In our experiments we only perform re-hashing on half (i.e., two) hash functions.

Frequency of dynamic re-hashing. Lastly, a designer chooses the frequency in which to perform dynamic re-hashing. An obvious choice would be to enable re-hashing whenever aborts occur, before the retry of the aborted transaction. Additionally, the TM system may pro-actively *avoid* future false conflicts by re-hashing on every transaction commit as well, regardless of the presence or absence of aborts. Our signature implementations re-hash on both commit and abort. Note that in a signature-based TM system it may not be possible to differentiate between false and true conflicts. Thus a policy for dynamic re-hashing may try to re-hash for a pre-defined number of times before trapping to a software contention manager [98] for maximal flexibility in resolving conflicts.

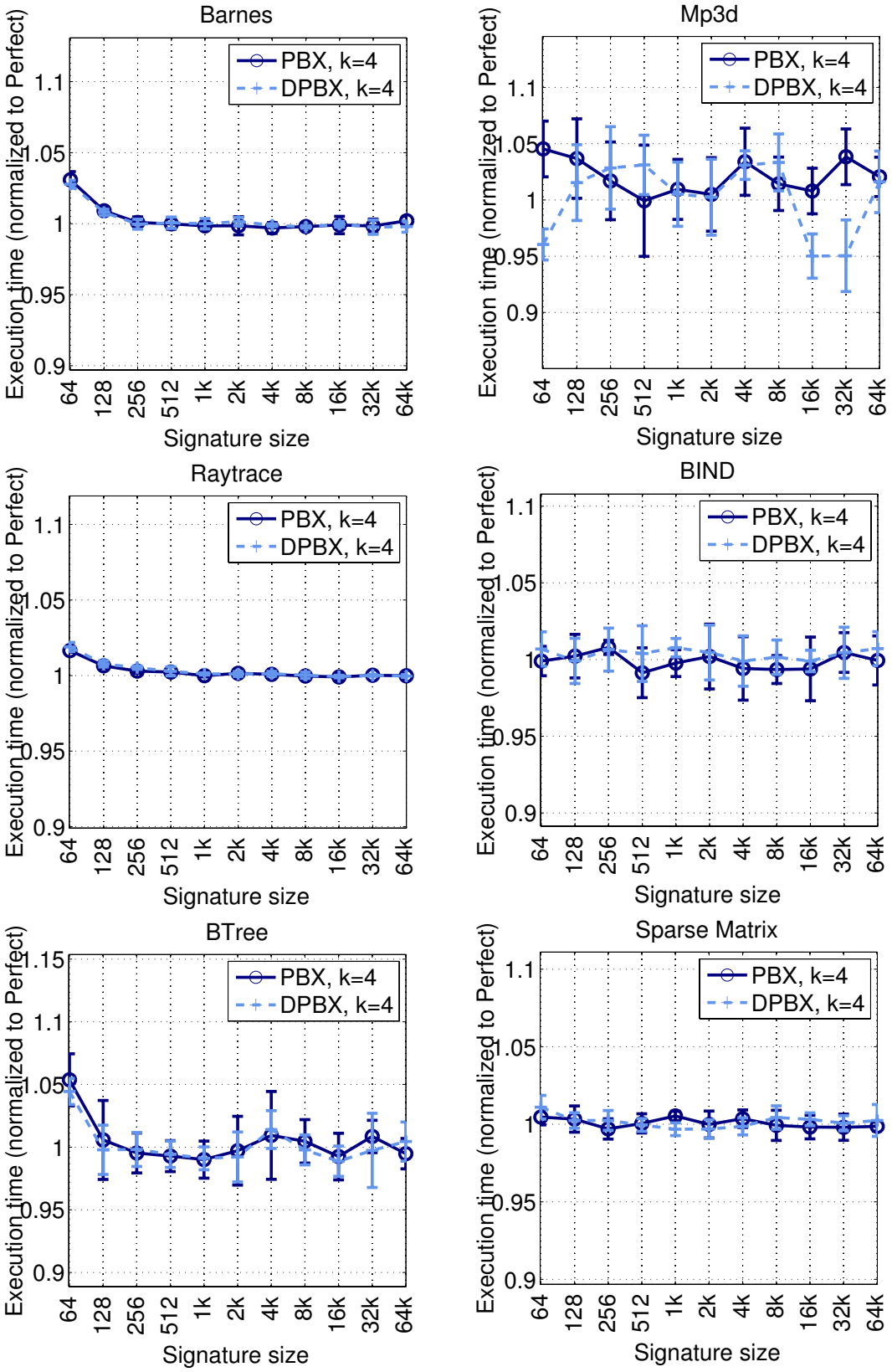


FIGURE 7-18. Normalized execution times before and after dynamic re-hashing

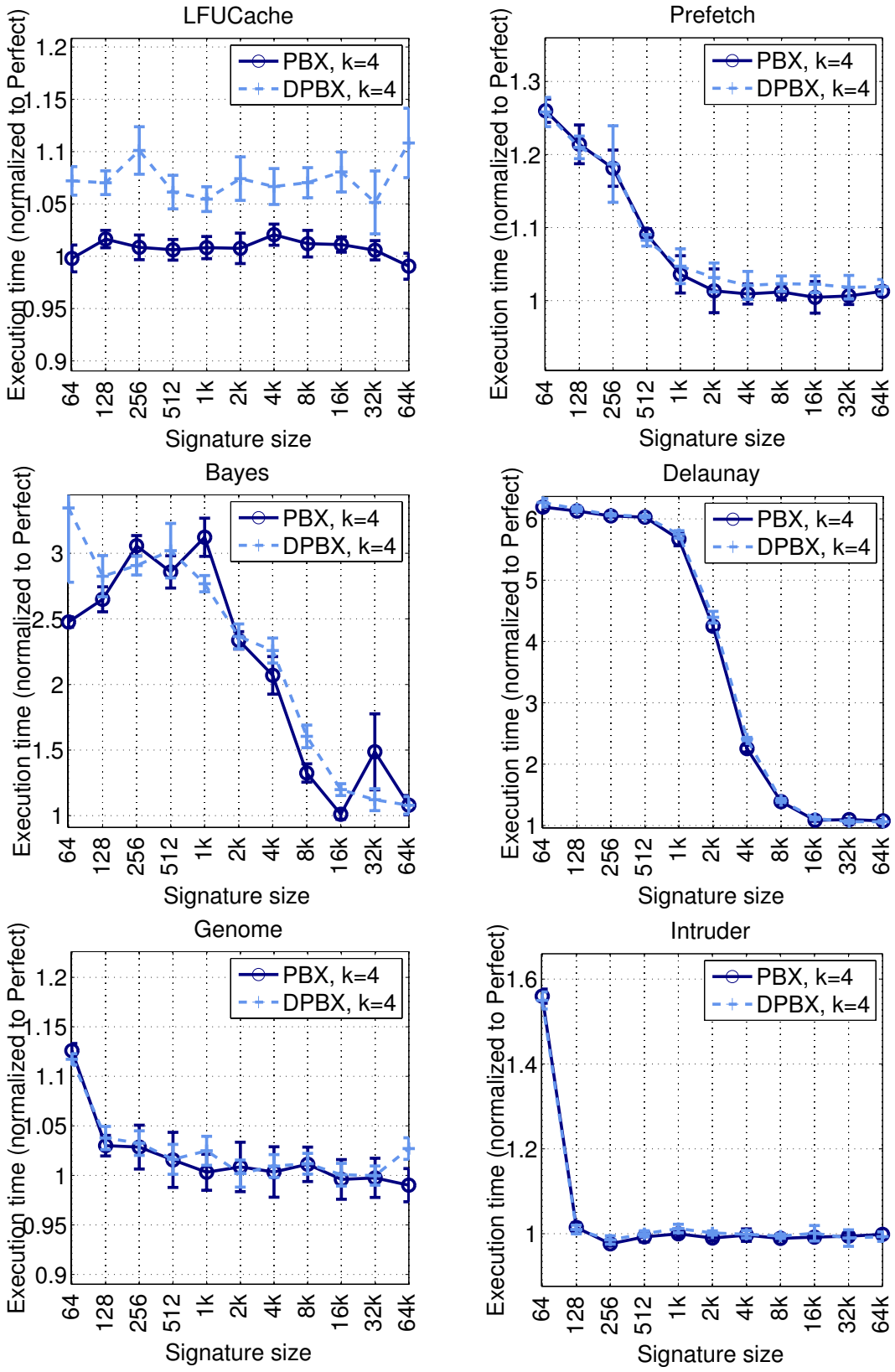


FIGURE 7-18. Normalized execution times before and after dynamic re-hashing

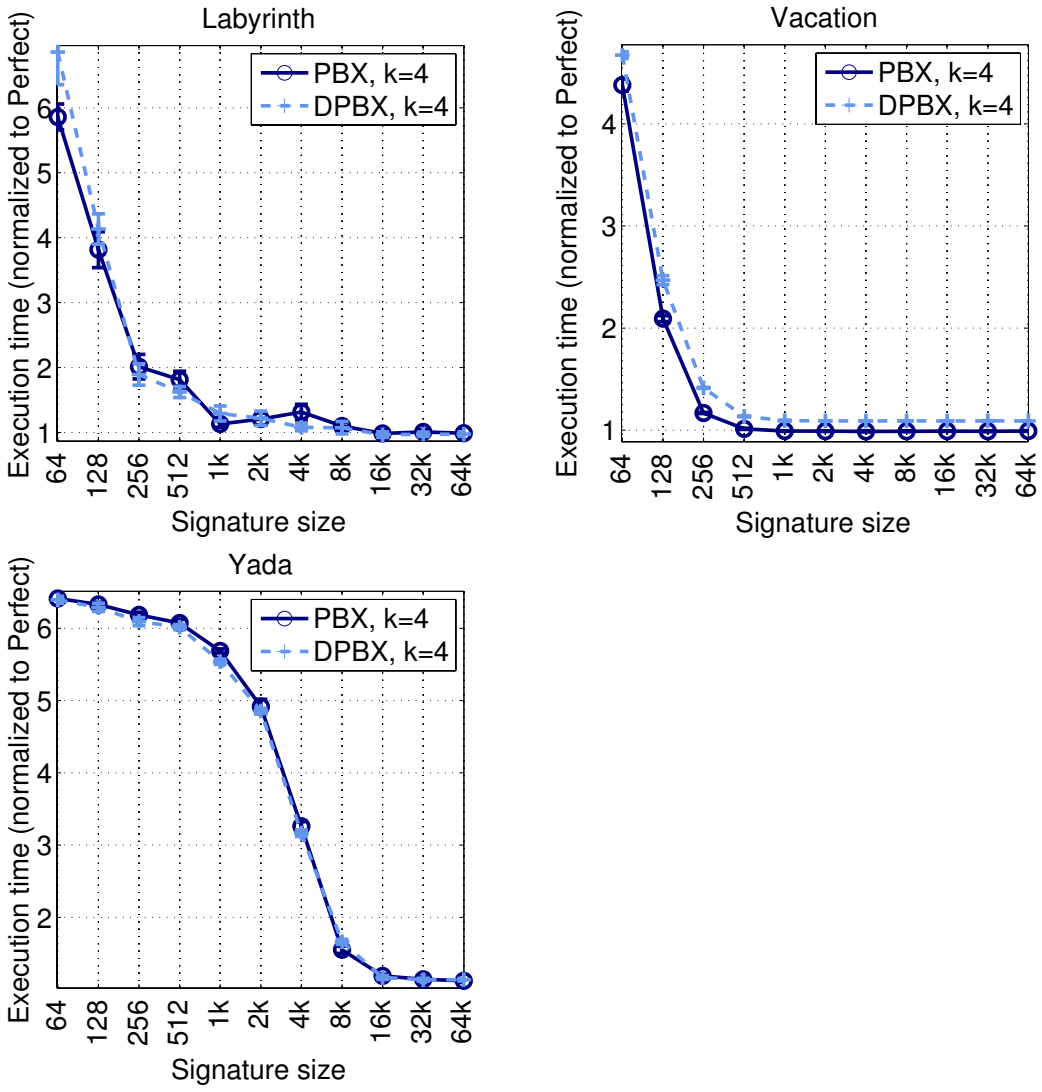


FIGURE 7-18. Normalized execution times before and after dynamic re-hashing

7.4.3 Results

Figure 7-18 shows the normalized execution times before and after dynamic re-hashing is used. Results with dynamic re-hashing are lines labeled “DPBX”. Overall, we find that dynamic re-hashing does not help our workloads. There are a number of reasons for this. First, the many of our workloads do not spend much time inside transactions. Examples include the SPLASH-2 applications (Barnes, Raytrace, and Mp3d) and the micro-benchmarks (Btree, Sparse Matrix). Thus, there is little opportunity to use re-hashing. Second, many STAMP workloads have large transactions. When small signatures are used the transactions fill up the signatures. Dynamic re-hashing only perturbs the mapping of addresses to signature bits, and not the overall utilization (i.e., how many signature bits are set) of the signature. Techniques that reduce the signature utilization (e.g., spatial locality) are more effective for these workloads. Finally, dynamic re-hashing may be harmful when most of the conflicts are true rather than false conflicts. An example is LFUCache. The majority of signature conflicts are true conflicts, and re-hashing increases the amount of aborts (and thus runtime) of the program. Re-hashing likely changes the set of processors that are involved in conflicts (e.g., processor P1 may conflict with processors P2 and P3 before aborting, and after re-hashing it may conflict with P4). Instead of serializing behind one set of processors, re-hashing may continually alter the sets of processors that an aborted processor conflicts with. A solution to avoiding performance degradation in these scenarios is to trap to a software contention manager after a pre-determined number of retries to avoid using re-hashing indefinitely.

In summary, we find that dynamic re-hashing, although theoretically appealing, does not exhibit much benefit for our workloads.

7.5 Future Research Applying Entropy to Signature Extensions

In Chapter 5 we discussed how the concept of entropy can be used to find the most random bits in input addresses. These bits can then be fed into our simple PBX hash functions to produce random hash values. We now describe how entropy can be applied in future research on the signature extensions of spatial locality and coarse-fine hashing.

With our spatial locality signature extension, we first characterize our workload's spatial locality and then tailor signature hashing for spatial locality using static signatures, or use additional physical signatures to dynamically adapt to a workload's spatial locality. The way we envision entropy being used to adapt to spatial locality is that signature designers may choose to add hardware to measure the amount of entropy for each bit in the address over a period of time. Address bits belonging to addresses that fall within a spatially local region will exhibit more randomness than the address bits belonging to addresses that fall outside the region. Furthermore, these random bits may not to be contiguous (unlike our results in Chapter 5) because of paging activity in physical memory. The most random bits (which exhibit randomness above a specified threshold value) can then be selected as inputs to signature hash functions (using programmable select logic), while the less random bits can be ignored. This allows a single physical signature to adapt to dynamic changes in a workload's spatial locality, despite different dynamic memory behavior in executions of the same workload.

Coarse-fine hashing may also apply entropy in a manner similar to spatial locality described above. Recall that coarse-fine hashing allows two configurable parameters. The first specifies the number of fine bits and the second specifies how many hash functions to use on the coarse and fine address bits. Entropy may be used to dynamically find the number of contiguous fine bits,

since these are bits that exhibit more randomness than their neighboring bits. Note that unlike how entropy is used for spatial locality we assume that fine bits are contiguous in the address. This assumption is reasonable in systems with limited paging activity. Thus with coarse-fine hashing the signature hardware looks for sharp changes in entropy between contiguous bits of the address in order to detect the partition between coarse and fine bits. Once this partition is found, another policy may dynamically decide based on other metrics how many hash functions to devote to the coarse and fine bits. An example policy may decide to count how many positive lookups occur with different hash configurations over time.

7.6 Related Work

7.6.1 Architectural Support for Monitoring and Debugging Memory Locations

At a high level the related work that is closest to XID independence is any proposal that involves hardware support for monitoring and debugging memory locations. Examples of such proposals include MemTracker [116] and iWatcher [131]. These systems have the capability to monitor memory locations using hardware meta-data and take actions (hardware or software-defined) if a monitored location matches some pre-defined rule and triggers an exception (MemTracker) or is the target of pre-defined operations (read-only, write-only, or read-write accesses in iWatcher).

For XID independence, we need similar hardware to operate on XIDs instead of memory addresses. This can potentially require smaller total hardware state. Similar to the related work, if

XIDs conflict the hardware can raise an exception and trap to a user-specified handler to handle the conflict. Otherwise XID conflicts can be handled via a conflict resolution policy in hardware.

7.6.2 Colorama

The related work that is closest to our object ID proposal (both software and hardware) is Ceze et al.'s Colorama [22]. In Colorama, the programmer explicitly specifies the color of shared objects or fields. This is similar to object IDs because we also propose an interface for the programmer to annotate objects or fields as an object ID. However instead of operating on colors we operate on memory addresses.

Colorama's underlying hardware detects and prevents conflicts (via hardware TM with implicit transactions or a lock-based implementation) and ensures a serializable execution amongst all dependent threads. If implemented with a HTM, concurrent transactions that access the same color can result in conflicts, while transactions that access different colors should not conflict. Colorama's hardware builds upon Mondrian memory protection [122], which associates protection meta-data on each word of memory. Object IDs operate at higher granularities, and an ID is typically an address value representing the address of an object or field. Colorama adds additional hardware to track the color IDs that are accessed by each processor (in per processor hardware tables). This is similar to the hardware required to implement the lookup and storage of object IDs in each processor. Both Colorama and object IDs support a finite number of colors or IDs (e.g., maximum of 4k colors for Colorama). However object IDs can store IDs using Bloom filters to allow a logically unbounded number of IDs. Finally, if Colorama is implemented on top of a HTM system, it may incur overheads similar to object IDs in the coherence messages to transfer the color meta-data to other processors for conflict detection.

7.6.3 Software Transactional Memory Support for Object-based Conflict Detection

The idea of performing conflict detection on object granularity [34,47,96,46] rather than on cache-line or word-sized granularities [34,44,118] has previously been examined in the context of software transactional memory (STM) systems. In the context of STM systems, objects are defined as instances of types declared by the programmer in the TM program. Our definition of objects is less strict and includes portions of a data type (e.g., different hash buckets can be different objects under our definition).

Saha et al. [96] describe how their McRT-STM system supports both object-based and cache-line conflict detection. For managed languages such as Java, their McRT-STM can easily support conflict detection on objects. However, for unmanaged languages such as C/C++, they rely on their specialized McRT-Malloc [51] memory allocation library to detect when to invoke object-based conflict detection. Cache-line or word-based conflict detection operates by mapping addresses (e.g., cache-lines or byte addresses) to transaction records which record the ownership information for those memory locations.

There are at least three reasons why object-based conflict detection is more attractive than conflict detection on cache-lines or words in STMs. First, computing conflict information on small granularities such as cache-lines or words may incur high overheads in a STM. This is because STMs need to locate and read the transaction records for cache-lines or words, and this is done in software. These overheads are exacerbated if transactions are dominated by memory instructions. Second, storing and querying object-based conflict detection information may lead to better cache locality. Adl-Tabatabai et al. [2] discuss that their STM's transaction records can

be embedded within objects, and thus the transactional meta-data can be readily accessed once the object is accessed. Alternatively, a cache-line or word-based approach may need to query a separate software structure which stores the transaction record information for the memory address that is accessed. This may lead to additional cache accesses and poor cache locality. Third, object-based conflict detection may allow additional opportunities for the compiler to optimize the transactional program. Both Harris et al. [46] and Adl-Tabatabai et al. [2] describe numerous compiler optimizations that can occur for object-based STMs. These same optimizations are harder to perform on STMs which implement cache-line or word-based conflict detection, since objects in a transactional program may span multiple cache-lines and the cache-lines may incur false sharing.

A disadvantage for object-based conflict detection in STMs is that the performance of the STM may not scale with an increasing number of processors. This is because conflict detection is performed at a much coarser granularity compared to cache-lines or words, and false conflicts could occur when different threads access different fields of the same object (and at least one of the accesses is a write). These performance degradations have influenced the design decisions in several STMs to implement both object-based and cache-line or word-based conflict detection, including McRT [96] and TL2 [34].

7.6.4 Software Support for Spatial Locality

In this chapter we have focused solely on hardware support for optimizing signatures for spatial locality. However, there exists the potential for even greater performance benefits by exploring the synergy between software and hardware support for spatial locality. The programmer could use spatial locality-aware memory allocators in the program, and the software may pass

hints down to the hardware regarding which hashing granularity would likely yield the lowest signature false conflicts.

For example, Vam [37] and ccmalloc [27] are cache-conscious heap memory allocators. Vam allocates heap objects based on a page-sized units, thereby increasing the spatial locality of objects by allocating pieces of an object on spatially-local pages. ccmalloc defines an interface that allows a programmer to pass in a data structure element that the program is likely to contemporaneously access with the element to be allocated, thus taking advantage of both temporal and spatial locality. If no hardware support for spatial locality exists, these cache-conscious memory allocators have the potential to reduce the set of addresses that needs to be inserted into a signature without modifying signature hardware. If hardware support for spatial locality is available, there exists the opportunity for the allocators to align data structures in a deterministic fashion and to pass this information down to the hardware. The hardware can then use this information to customize the signature hashes (e.g., hash on the software-determined granularity). For example, a simple software policy could assign thread-private and shared heap data structures to disjoint regions of the heap, and individual objects could be aligned within these regions. These regions could be assigned and located such that some number of virtual and physical bits could be used to distinguish the regions. Then signature hashes could ignore some number of non-relevant address bits that could alias amongst the different regions (e.g., low-order address bits).

Furthermore, there have been numerous research on restructuring computation or data structures to improve temporal and spatial locality in caches. In computation restructuring, given a fixed data layout, the goal is to reorder memory accesses such that multiple accesses to the same object (or cache line) occur close in time, thereby enhancing temporal locality [20,123]. Data

relocation focuses on maximizing spatial locality, such that memory words that are accessed closely in time reside in the same cache lines. Examples include list linearization [32] and clustering [27]. However optimizing spatial locality also requires understanding the trade-offs in cache conflicts and false sharing. False sharing and spatial locality were examined by Torrelas et al. [111], and they provided five programmer-transparent techniques in which data locality could be improved without necessarily increasing false sharing. Cache conflicts can be reduced through techniques such as data-coloring [27] and data copying [63]. These techniques are complementary to hardware support for spatial locality and could be used in conjunction to the proposals we describe in this chapter.

7.7 Conclusions and Future Work

This chapter explores six extensions to signatures. First, we examine the notion of XID independence. XID independence does not significantly improve execution time. This is perhaps due to our conservative estimate of which XIDs might potentially dynamically conflict. Second, we examine object IDs. Object IDs do offer performance benefits but are likely to require significant hardware and may unduly burden the TM programmer.

We also examine the notion of program spatial locality and its applicability to signatures. We evaluate three signature extensions that optimize for spatial locality: static signatures, dynamic signatures, and coarse-fine hashing. We find that static and dynamic signatures work well, and overall better than coarse-fine hashing. Coarse-fine hashing can lead to complex conflict formations that make it much harder to reason about the performance of the TM system. Finally, it is not clear whether the benefits of dynamic signatures justify its high implementation costs (a set of signatures and performance counters).

Lastly, we examine the concept of dynamic re-hashing. Although conceptually appealing, there is no evidence to indicate any benefits for our workloads.

Going forward, spatial locality seems to be the most promising technique for signature extensions. It likely requires little additional hardware and little or no change in the TM programming interface (depending on whether software support for spatial locality is available). We recommend future research on signature extensions target this area, and perhaps incorporate feedback from entropy measurements as discussed in Section 7.5 in alternative implementations.

Chapter 8

Summary and Reflections

Transactional memory (TM) is a promising parallel programming paradigm that avoids many of the problems of lock-based programming — deadlocks and software composability. We focus on hardware transactional memory (HTM) systems because simulated HTM designs perform better than any real software TM (STM) system or simulated hybrid TM system. This dissertation makes several contributions in the space of conflict detection using signatures in HTM systems, and also performance debugging HTM systems. This chapter first summarizes the dissertation's contributions in Section 8.1. In Section 8.2 I offer some reflections and opinions based on this research. Future work was previously discussed in Section 4.10, Section 5.8, Section 6.7, and Section 7.7.

8.1 Summary

This dissertation focuses on the design and analysis of conflict detection using hardware signatures in HTM systems, as well as a framework for performance debugging HTM systems. First, we contributed a new HTM system called LogTM-SE. LogTM-SE combines LogTM's log for version management and Bulk's signatures for conflict detection. The synergy of these ideas allows LogTM-SE to implement simple hardware (e.g., Bloom filters) in each processor core and to export this hardware to the operating system in order to virtualize transactions under context switching and page remapping. Furthermore, unbounded nesting is supported through the actions

of saving and restoring the signatures onto the log. Overall, LogTM-SE is an unbounded HTM system that decouples HTM state from the caches.

Second, this dissertation contributes Notary, a collection of two hardware techniques to tackle two problems with signature designs. The first problem is that H_3 signature hashes may incur high area and power overheads in its implementation. We solve these problems by using the idea of address-bit entropy to locate the address bits that exhibit the most randomness, and using those as inputs into a newly-defined hash function called Page-Block-XOR (PBX). We show PBX incurs lower area and power overheads than H_3, and performs similarly to H_3 across our TM workloads. The second problem is that signature false conflicts may be due to signature bits set by thread-private memory addresses. We contribute two privatization techniques to mitigate this problem. If stack pages are thread-private, we discuss that it is possible for the compiler or runtime system to statically mark stack pages as private. Signatures can then refrain from inserting addresses to private stack pages. However, removing stack references may not be sufficient to reduce false conflicts. We contribute a heap-based privatization interface in which we give the programmer the ability to mark which memory objects do not participate in conflict detection. Our evaluations show that privatization significantly improves execution times for several STAMP workloads.

Third, we contribute TMProf, a lightweight hardware profiling infrastructure for performance debugging HTM systems. TMProf consists of a set of per-processor performance counters which track the frequency and overheads of HTM events. The base TMProf implementation accounts for all cycles in a HTM system, by breaking each processor's cycle into stalls, aborts, wasted transaction, useful transaction, commit, non-transactional, and implementation-specific cycles. An extended TMProf implementation adds transaction-level profiling that reveals in-depth informa-

tion not found in the base implementation. We evaluate TMProf using two HTMs (LogTM-SE and an approximation of TCC), and find two results. First, TMProf provides in-depth information to help the HTM designer understand the performance impact of changing several HTM parameters. Second, we find empirical data suggesting TMProf should be augmented to provide hardware support for critical-path analysis. Overall, TMProf is a useful profiling infrastructure for HTM designers and TM programmers.

Fourth, we contribute six extensions to signatures. These extensions focus on additional techniques to reducing false conflicts. This is done by leveraging static transaction independence, tracking the set of objects accessed within transactions, optimizing for a program's spatial locality, and dynamically adapting hash functions to conflicts. Our evaluations show most of the extensions do not lead to significant performance improvements, but spatial locality is a promising extension for future researchers to explore.

8.2 Reflections

In this section, I reflect on the research in my dissertation with the benefit of hindsight and the freedom to state my opinions.

When I first started performing research on signatures, it seemed to me an obvious place to start would be to tackle its false positive problem. However, many papers regarding different uses of signatures were published concurrently with my research. Overall, I found those proposals to be thought-provoking, and I wish I had the opportunity to devote more resources on research in this area.

Regarding the probability of signatures being implemented in commercial processors, I think there needs to be a compelling application that demands its wide-spread use. Industrial affiliates seem to be fine with the impreciseness offered by signatures, but are less compelled to implement it in hardware if there are not enough uses for it. I can certainly see Bloom filters being implemented for micro-architectural features, but it remains an open question whether it will ever be visible to the software layers (e.g., through a software API).

Although commercial HTMs exist or have been announced (e.g., Sun's Rock), there are many hurdles that remain until they viable for general purpose programming. I feel a core area that needs much more research is on TM programming. Specifically, researchers (and industry) need more compelling evidence indicating that TM programming is indeed much easier than lock-based programming. This process applies to new TM programs as well as adding transactions in existing lock-based programs. If this doesn't happen, I fear that TM will degrade into a set of optimizations (hardware or software) for lock-based programs, and never become a full-fledged parallel programming paradigm.

References

[1] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard. Dynamic Separation for Transactional Memory. Technical Report MSR-TR-2008-43, Microsoft Research, 2008.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, 2006.

[3] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[4] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proc. of the 9th IEEE Symp. on High-Performance Computer Architecture*, pages 7–18, Feb. 2003.

[5] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, Jul/Aug 2006.

[6] AMD Corporation. AMD Introduces the World's Most Advanced x86 Processor, Designed for the Demanding Datacenter. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_15008~119768,00.html, Sept. 2007.

[7] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. of the 11th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2005.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[9] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM Transactional Application Programming Interface. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2007.

[10] C. Ballapuram, K. Puttaswamy, G. H. Loh, and H.-H. S. Lee. Entropy-Based Low Power Data TLB Design. In *Proc. of the Intnl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2006.

[11] J. C. Becker, A. Park, and M. Farrens. An Analysis of the Information Content of Address Reference Streams. In *Proc. of the 24th Annual IEEE/ACM International Symp. on Microarchitecture*, Nov. 1991.

[12] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[13] C. Blundell, J. Devietti, E. C. Lewis, and M. M. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[14] C. Blundell, E. C. Lewis, and M. M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[15] C. Blundell, E. C. Lewis, and M. M. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, June 2006.

[16] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[17] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[18] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. Dongarra. Matrix Market: A Web Resource for Test Matrix Collections. *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, 1997.

[19] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, June 2004.

[20] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler Optimizations for Improving Data Locality. In *Proc. of the 6th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[21] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions (extended abstract). In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 106–112, 1977.

[22] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectura Support for Data-Centric Synchronization. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2007.

[23] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[24] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[25] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, pages 97–108, Feb. 2007.

[26] H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: A Transactional Application Profiling Environment. In *Proc. of the 19th Intnl. Conf. on Supercomputing*, June 2005.

[27] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making Pointer-Based Data Structures Cache Conscious. *IEEE Computer*, 33(12):67–74, Dec. 2000.

[28] W. Chuang, S. Narayanasmy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, O. Colavin, and B. Calder. Unbounded Page-Based Transactional Memory. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[29] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2006.

[30] J. Chung, C. C. Minh, A. McDonald, H. Chafi, B. D. Carlstrom, T. Skare, C. Kozyrakis, and K. Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[31] D. Citron and L. Rudolph. Creating a Wider Bus Using Caching Techniques. In *Proc. of the 1st IEEE Symp. on High-Performance Computer Architecture*, pages 90–99, Feb. 1995.

[32] D. W. Clark. *List structure: measurements, algorithms, and encodings*. PhD thesis, Carnegie-Mellon University, 1976.

[33] P. Damron, A. Fedorova, Y. Lev, V. Luchango, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[34] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, Sept. 2006.

[35] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-generation Processors. *IEEE Micro*, 17(5):12–18, September/ October 1997.

[36] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE Transactions on Networking*, 8(3):281–293, June 2000.

[37] Y. Feng and E. D. Berger. A Locality-Improving Dynamic Memory Allocator. In *Memory System Performance Workshop*, pages 68–77, 2005.

[38] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Using Interaction Costs for Microarchitectural Bottleneck Analysis. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 228–241, Dec. 2003.

[39] B. Goetz. Optimistic Thread Concurrency. Azul Systems Whitepaper, Jan. 2006.

[40] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-Based Placement Functions. In *Proc. of the 1997 Intnl. Conf. on Supercomputing*, July 1997.

[41] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture*, pages 172–181, Jan. 1999.

[42] D. W. Hammerstrom and E. S. Davidson. Information Content of CPU Memory Referencing Behavior. In *Proc. of the 4th Annual Symp. on Computer Architecture*, Mar. 1977.

[43] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, June 2004.

[44] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. of the 18th SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2003.

[45] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, June 1991.

[46] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, June 2006.

[47] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Twenty-Second ACM Symposium on Principles of Distributed Computing, Boston, Massachusetts*, July 2003.

[48] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Annual Intnl. Symp. on Computer Architecture*, pages 289–300, May 1993.

[49] J. K. Hollingsworth. An Online Computation of Critical Path Profiling. In *Symposium on Parallel and Distributed Tools*, pages 11–20.

[50] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Race Recording. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[51] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management*, June 2006.

[52] Intel. Intel C++ STM Compiler, Prototype Edition 2.0. http://softwarecommunity.intel.com/articles/eng/1460.htm.

[53] Intel Press Release. Dual Core Era Begins, PC Makers Start Selling Intel-Based PCs. http://www.intel.com/pressroom/archive/releases/20050418comp.htm, Apr. 2005.

[54] Internet Systems Consortium. Berkeley Internet Name Domain (BIND). http://www.isc.org/index.pl?/sw/bind/.

[55] Internet Systems Consortium. BIND (Berkeley Internet Name Domain). http://www.isc.org.

[56] T. Jinmei and P. Vixie. Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server. In *Proceedings of the 2006 Usenix Annual Technical Conference*, June 2006.

[57] T. JINMEI and P. Vixie. Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server. In *2006 USENIX Annual Technical Conference*, June 2006.

[58] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proc. of the 25th Annual Intnl. Symp. on Computer Architecture*, pages 15–26, June 1998.

[59] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *Proc. of the 10th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2004.

[60] M. Kharbutli, Y. Solihin, and J. Lee. Eliminating Conflict Misses Using Prime Number-Based Cache Indexing. *IEEE Transactions on Computers*, 54(5):573–586, 2005.

[61] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[62] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 209–220, Mar. 2006.

[63] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the 4th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[64] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[65] H. Le, W. Starke, J. Fields, F. O'Connell, D. Nguyen, B. Ronchetti, W. Sauer, E. Schwarz, and M. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6), 2007.

[66] T. Li, A. R. Lebeck, and D. J. Sorin. Quantifying Instruction Criticality for Shared Memory Multiprocessors. In *Proc. of the 15th ACM Symp. on Parallel Algorithms and Architectures*, June 2003.

[67] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[68] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[69] M. M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin, 2003.

[70] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[71] A. Matveev, O. Shalev, and N. Shavit. Dynamic Identification of Shared Transactional Locations. Technical report, Tel-Aviv University, 2007.

[72] C. J. Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. In *Proc. of the 2002 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[73] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[74] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2005.

[75] R. McDougall, J. Mauro, and B. Gregg. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Pearson Professional, 2006.

[76] M. Milovanovic, R. Ferrer, O. S. Unsal, A. Cristal, X. Martorell, E. Ayguade, J. Labarta, and M. Valero. Transactional Memory and OpenMP. In *Proceedings of the International Workshop on OpenMP*, June 2007.

[77] C. C. Minh, M. Trautmann, J. Chung, A. Mcdonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[78] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[79] K. E. Moore. *Log-Based Transactional Memory.* PhD thesis, University of Wisconsin-Madison, 2007.

[80] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, pages 258–269, Feb. 2006.

[81] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting Nested Transactional Memory in LogTM. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, Oct. 2006.

[82] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. In *Proc. of the 7th IEEE Symp. on High-Performance Computer Architecture*, Jan. 2001.

[83] J. E. B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[84] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 1–10, Aug. 1995.

[85] R. Pagh and F. F. Rodler. Cuckoo Hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms*, pages 121–133, 2001.

[86] A. Park and M. Farrens. Address Compression Through Base Register Caching. In *23rd Annual Symposium and Workshop on Microprogramming and Microarchitectures*, Nov. 1990.

[87] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 189–198, June 2002.

[88] C. Perfumo, N. Sonmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-Core Environment. In *Conference on Computing Frontiers*, May 2008.

[89] D. E. Porter, O. S. Hofmann, and E. Witchel. Predicting and Tuning the Performance of Multicore Systems. Technical Report 08-10, University of Texas at Austin, 2008.

[90] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[91] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. of the 34th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2001.

[92] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proc. of the 10th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[93] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Annual Intnl. Symp. on Computer Architecture*, June 2005.

[94] M. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997.

[95] B. R. Rau. Pseudo-randomly interleaved memory. In *Proc. of the 18th Annual Intnl. Symp. on Computer Architecture*, pages 74 – 83, May 1991.

[96] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Mar. 2006.

[97] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *Proc. of the 40th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2007.

[98] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *24th ACM Symp. on Principles of Distributed Computing*, July 2005.

[99] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *Proceedings of the IEEE International Workload Characterization Symposium (IISWC-2007)*, pages 107–113, Sept. 2007.

[100] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2003.

[101] A. Seznec. A Case For Two-way Skewed-associative Caches. In *Proc. of the 20th Annual Intnl. Symp. on Computer Architecture*, May 1993.

[102] C. E. Shannon. Prediction and Entropy of Printed English. *Bell Systems Technical Journal*, (30):50–64, 1951.

[103] A. Shriraman and S. Dwarkadas. Refereeing Conflicts in Transactional Memory Systems. Technical Report 939, University of Rochester, 2008.

[104] A. Singhal and A. J. Goldberg. Architectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000. In *Proc. of the 21st Annual Intnl. Symp. on Computer Architecture*, Apr. 1994.

[105] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. Power5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4), 2005.

[106] Sleepycat Software. Sleepycat Software: Berkeley DB Database. http://www.sleepycat.com.

[107] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, Sept. 1982.

[108] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[109] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Technical Report 915, University of Rochester, Feb. 2007.

[110] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett Packard Labs, June 2006.

[111] J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[112] M. Tremblay. Transactional memory for a modern microprocessor. In *26th ACM Symp. on Principles of Distributed Computing*, Aug. 2007.

[113] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. In *Proc. of the 13th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[114] A. Tucker, B. Smaalders, D. Singleton, and N. Kosche. Method and apparatus for execution and preemption control of computer process entities, Aug. 1999. U.S. Patent 5,937,187.

[115] H. Vandierendonck and K. D. Bosschere. XOR-Based Hash Functions. *IEEE Transactions on Computers*, 54(7):800–812, 2005.

[116] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, pages 273–284, Feb. 2007.

[117] Virtutech Inc. Simics Full System Simulator. http://www.simics.com/.

[118] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proc. of the international symposium on Code generation and optimization*, Mar. 2007.

[119] J.-M. Wang, S.-C. Fang, and W.-S. Feng. New Efficient Designs for XOR and XNOR Functions on the Transistor Level. *IEEE Journal of Solid-State Circuits*, 29(7):780–786, 1994.

[120] G. Weikum and H.-J. Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. Morgan Kaufmann, 1992.

[121] Wikipedia: The Free Encyclopedia. Birthday Paradox. http://en.wikipedia.org/wiki/Birthday_paradox.

[122] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proc. of the 10th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, Oct. 2002.

[123] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proc. of the SIGPLAN 1991 Conf. on Programming Language Design and Implementation*, June 1991.

[124] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Intnl. Symp. on Computer Architecture*, pages 24–37, June 1995.

[125] C.-Q. Yang and B. P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *8th International Conference on Distributed Computing Systems*, pages 366–373.

[126] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[127] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, pages 261–272, Feb. 2007.

[128] L. Yen, S. C. Draper, and M. D. Hill. Notary: Hardware Techniques to Enhance Signatures. In *Proc. of the 41st Annual IEEE/ACM International Symp. on Microarchitecture*, Nov. 2008.

[129] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. of the 1996 Intnl. Conf. on Supercomputing*, 1996.

[130] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Proc. of the 33rd Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2000.

[131] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, pages 224–237, June 2004.

[132] C. Zilles and R. Rajwar. Brief Announcement: Transactional Memory and the Birthday Paradox. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2007.

Appendix A

Supplements for (Chapter 4)

TABLE A-1. Raw cycles (in thousands) for BerkeleyDB, Cholesky, Mp3d, Radiosity, Raytrace

Workload	Lock	Perfect	BS	CBS	DBS	BS_64
BerkeleyDB	1,804	1,368	1,400	1,382	1,397	1,452
Cholesky	5,134	5,355	5,328	5,354	5,334	5,411
Mp3d	6,172	6,649	6,611	6,572	6,723	6,766
Radiosity	11,096	11,555	11,908	11,830	11,560	13,154
Raytrace	31,176	20,823	22,633	20,851	20,796	23,399

Appendix B

Supplements for (Chapter 5)

TABLE B-1. Raw cycles (in thousands) for perfect signatures before and after removing stack references

Workload	Ruby cycles before / after removing stack references
Barnes	15,105 / 15,017
Mp3d	6,351 / 6,535
Raytrace	32,044 / 31,491
BIND	11,760 / 11,790
BTree	10,987 / 11,156
Sparse Matrix	12,208 / 12,187
LFUCache	2,917 / 2,957
Prefetch	8,101 / 8,063
Bayes	13,967 / 14,074
Delaunay	12,283 / 12,322
Genome	5,214 / 5,169
Intruder	8,897 / 8,844
Labyrinth	16,357 / 16,477
Vacation	5,729 / 5,728
Yada	26,082 / 26,722

TABLE B-2. Raw cycles (in thousands) for perfect signatures before and after privatization

Workload	Ruby cycles before / after privatization
Bayes	13,967 / 14,295
Delaunay	12,283 / 12,210
Genome	5,214 / 5,001
Labyrinth	16,357 / 15,994
Yada	26,082 / 26,593

Appendix C

Supplements for (Chapter 6)

TABLE C-1. Raw cycles (in thousands) for Barnes, Raytrace, Mp3d

Workload	Ruby cycles	Stalls	Aborts	Wasted Trans.	Useful Trans.	Committing Stores in Wr. Buffer	Commit Token Stalls	Backoff	Barrier	Nontrans.
Barnes	1. 15,330	1. 3,599	1. 0	1. 1,747	1. 2,311	1. 0	1. 0	1. 3,721	1. 28,135	1. 205,775
	2. 14,835	2. 2,348	2. 0	2. 342	2. 2,395	2. 0	2. 0	2. 336	2. 27,400	2. 204,548
	3. 14,676	3. 2,511	3. 0	3. 372	3. 2,483	3. 0	3. 0	3. 241	3. 28,268	3. 200,948
	4. 15,189	4. 4,951	4. 0	4. 242	4. 2,556	4. 0	4. 0	4. 318	4. 26,956	4. 208,003
	5 .15,002	5. 185	5. 0	5. 1,203	5. 1,400	5. 659	5. 3,230	5. 114	5. 28,202	5. 205,036
	6. 14,704	6. 334	6. 0	6. 1,280	6. 1,440	6. 661	6. 89	6. 108	6. 27,057	6. 204,296
	7. 15,142	7. 2,902	7. 499	7. 769	7. 2,766	7. 0	7. 0	7. 9	7. 27,573	7. 207,754
	8. 14,923	8. 391	8. 0	8. 1,344	8. 1,404	8. 688	8. 286	8. 116	8. 27,076	8. 207,470
	9. 14,961	9. 368	9. 0	9. 1,327	9. 1,401	9. 685	9. 682	9. 126	9. 27,424	9. 207,366
Raytrace	1. 34,690	1. 17,286	1. 0	1. 12,210	1. 13,110	1. 0	1. 0	1. 24,276	1. 14,831	1. 473,325
	2. 31,451	2. 480	2. 0	2. 879	2. 12,691	2. 0	2. 0	2. 254	2. 18,729	2. 470,185
	3. 31,890	3. 1,497	3. 0	3. 1,945	3. 15,237	3. 0	3. 0	3. 2,584	3. 17,379	3. 471,599
	4. 35,002	4. 34,129	4. 0	4. 2,596	4. 17,252	4. 0	4. 0	4. 21,747	4. 7,565	4. 476,752
	5. 31,437	5. 12	5. 0	5. 1,345	5. 10,393	5. 2,092	5. 141	5. 358	5. 18,924	5. 469,730
	6. 31,424	6. 12	6. 0	6. 1,371	6. 10,413	6. 2,094	6. 130	6. 360	6. 18,655	6. 469,745
	7. 31,964	7. 1,726	7. 564	7. 2,953	7. 14,616	7. 0	7. 0	7. 36	7. 18,330	7. 473,199
	8. 31,909	8. 11	8. 0	8. 4,110	8. 10,113	8. 2,022	8. 1,893	8. 693	8. 19,477	8. 472,231
	9. 32,271	9. 34	9. 0	9. 4,294	9. 10,167	9. 2,045	9. 7,120	9. 1,643	9. 18,813	9. 472,218
Mp3d	1. 14,086	1. 37,437	1. 0	1. 27,762	1. 5,675	1. 0	1. 0	1. 65,754	1. 47,722	1. 41,022
	2. 7,083	2. 9,609	2. 0	2. 15,910	2. 6,970	2. 0	2. 0	2. 9,046	2. 37,441	2. 34,360
	3. 6,954	3. 9,335	3. 0	3. 15,617	3. 7,018	3. 0	3. 0	3. 8,366	3. 36,675	3. 34,256
	4. 4,793	4. 4,154	4. 0	4. 150	4. 5,216	4. 0	4. 0	4. 43	4. 32,493	4. 34,632
	5. 5,874	5. 123	5. 0	5. 8,005	5. 4,512	5. 1,782	5. 3,050	5. 7,808	5. 35,489	5. 33,212
	6. 5,656	6. 145	6. 0	6. 8,075	6. 4,808	6. 1,788	6. 743	6. 6,811	6. 35,031	6. 33,095
	7. 6,455	7. 5,241	7. 4,508	7. 12,496	7. 8,997	7. 0	7. 0	7. 263	7. 36,940	7. 34,815
	8. 5,869	8. 169	8. 0	8. 8,151	8. 4,612	8. 1,734	8. 2,591	8. 7,580	8. 35,862	8. 33,212
	9. 6,278	9. 228	9. 0	9. 7,123	9. 4,159	9. 1,606	9. 8,361	9. 8,257	9. 37,039	9. 33,676

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-2. Raw cycles (in thousands) for Sparse Matrix, BTree, LFUCache

Workload	Ruby cycles	Stalls	Aborts	Wasted Trans.	Useful Trans.	Committing Stores in Wr. Buffer	Commit Token Stalls	Backoff	Barrier	Nontrans.
Sparse Matrix	1. 12,194	1. 13	1. 0	1. 4	1. 149,359	1. 0	1. 0	1. 1	1. 15,043	1. 30,694
	2. 12,208	2. 5	2. 0	2. 8	2. 149,382	2. 0	2. 0	2. 1	2. 15,422	2. 30,503
	3. 12,192	3. 5	3. 0	3. 10	3. 149,359	3. 0	3. 0	3. 1	3. 14,668	3. 31,026
	4. 12,183	4. 2	4. 0	4. 7	4. 149,321	4. 0	4. 0	4. 1.5e-1	4. 14,835	4. 30,759
	5 .12,146	5. 1.6e-2	5. 0	5. 2	5. 148,203	5. 1,148	5. 144	5. 1	5. 15,026	5. 29,808
	6. 12,143	6. 7.3e-2	6. 0	6. 2	6. 148,204	6. 1,147	6. 3.4e-1	6. 1	6. 14,805	6. 30,126
	7. 12,186	7. 4	7. 14	7. 7	7. 149,401	7. 0	7. 0	7. 1.4e-1	7. 14,780	7. 30,773
	8. 14,586	8. 8.0e-1	8. 0	8. 4	8. 148,239	8. 1,145	8. 33,836	8. 2	8. 17,901	8. 32,249
	9. 20,174	9. 2	9. 0	9. 5	9. 148,147	9. 1,144	9. 110,249	9. 2	9. 25,213	9. 38,020
BTree	1. 11,061	1. 811	1. 0	1. 723	1. 93,712	1. 0	1. 0	1. 983	1. 5,037	1. 75,714
	2. 10,906	2. 438	2. 0	2. 280	2. 93,661	2. 0	2. 0	2. 19	2. 5,037	2. 75,063
	3. 10,949	3. 396	3. 0	3. 266	3. 93,671	3. 0	3. 0	3. 20	3. 5,036	3. 75,791
	4. 173,535	4. 1,842,411	4. 0	4. 303,591	4. 199,398	4. 0	4. 0	4. 188,529	4. 5,037	4. 237,603
	5. 10,987	5. 52	5. 0	5. 388	5. 90,982	5. 1,633	5. 2,401	5. 38	5. 5,037	5. 75,271
	6. 10,832	6. 52	6. 0	6. 378	6. 90,964	6. 1,635	6. 9	6. 39	6. 5,037	6. 75,196
	7. 11,128	7. 287	7. 166	7. 257	7. 94,852	7. 0	7. 0	7. 19	7. 5,037	7. 77,428
	8. 11,167	8. 53	8. 0	8. 385	8. 91,099	8. 1,638	8. 3,116	8. 39	8. 5,037	8. 77,312
	9. 11,473	9. 55	9. 0	9. 383	9. 90,898	9. 1,635	9. 10,137	9. 39	9. 5,037	9. 75,385
LFUCache	1. 11,004	1. 36,377	1. 0	1. 26,579	1. 2,304	1. 0	1. 0	1. 64,450	1. 15,998	1. 30,361
	2. 3,833	2. 8,643	2. 0	2. 12,901	2. 3,328	2. 0	2. 0	2. 5,796	2. 4,780	2. 25,881
	3. 3,534	3. 6,876	3. 0	3. 9,483	3. 2,967	3. 0	3. 0	3. 4,101	3. 4,512	3. 28,603
	4. 2,788	4. 2,933	4. 0	4. 223	4. 1,929	4. 0	4. 0	4. 154	4. 3,415	4. 35,949
	5. 2,556	5. 32	5. 0	5. 151	5. 994	5. 399	5. 52	5. 124	5. 4,121	5. 35,017
	6. 2,558	6. 34	6. 0	6. 143	6. 999	6. 400	6. 28	6. 125	6. 4,134	6. 35,059
	7. 2,978	7. 3,336	7. 974	7. 4,234	7. 2,908	7. 0	7. 0	7. 66	7. 2,991	7. 33,136
	8. 2,620	8. 37	8. 0	8. 262	8. 1,013	8. 418	8. 367	8. 229	8. 2,530	8. 37,057
	9. 2,630	9. 44	9. 0	9. 550	9. 1,017	9. 427	9. 1,543	9. 674	9. 2,448	9. 35,374

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-3. Raw cycles (in thousands) for Prefetch, BIND, Vacation

Workload	Ruby cycles	Stalls	Aborts	Wasted Trans.	Useful Trans.	Committing Stores in Wr. Buffer	Commit Token Stalls	Backoff	Barrier	Nontrans.
Prefetch	1. 5,445	1. 120,175	1. 0	1. 5,072	1. 10,607	1. 0	1. 0	1. 137	1. 20,472	1. 17,782
	2. 6,417	2. 119,332	2. 0	2. 46,729	2. 7,571	2. 0	2. 0	2. 2,722	2. 9,867	2. 19,110
	3. 5,280	3. 111,751	3. 0	3. 10,484	3. 10,510	3. 0	3. 0	3. 571	3. 18,139	3. 17,505
	4. 5,040	4. 110,858	4. 0	4. 545	4. 10,645	4. 0	4. 0	4. 5	4. 21,688	4. 17,542
	5 .4,281	5. 0	5. 0	5. 72,814	5. 5,399	5. 42	5. 3	5. 9,007	5. 32,982	5. 16,747
	6. 4,330	6. 0	6. 0	6. 74,334	6. 5,402	6. 42	6. 4	6. 9,340	6. 32,994	6. 16,460
	7. 5,344	7. 97,904	7. 431	7. 14,465	7. 14,912	7. 0	7. 0	7. 12	7. 19,546	7. 23,750
	8. 4,449	8. 0	8. 0	8. 77,138	8. 5,248	8. 41	8. 25	8. 9,684	8. 32,962	8. 17,278
	9. 4,593	9.0	9. 0	9. 80,574	9. 5,245	9. 41	9. 90	9. 10,058	9. 33,607	9. 17,366
BIND	1. 11,806	1. 195	1. 0	1. 3	1. 9,153	1. 0	1. 0	1. 2.2e-1	1. 0	1. 368,429
	2. 11,664	2. 123	2. 0	2. 7	2. 9,196	2. 0	2. 0	2. 2.3e-1	2. 0	2. 363,941
	3. 11,761	3. 149	3. 0	3. 23	3. 8,893	3. 0	3. 0	3. 1.4e-1	3. 0	3. 367,290
	4. 11,766	4. 94	4. 0	4. 4	4. 8,930	4. 0	4. 0	4. 1.2e-1	4. 0	4. 367,499
	5. 11,762	5. 1.9e-1	5. 0	5. 191	5. 8,020	5. 300	5. 39	5. 8.5e-1	5. 0	5. 367,834
	6. 11,789	6. 3.8e-2	6. 0	6. 330	6. 7,712	6. 301	6. 1.2e-1	6. 1	6. 0	6. 368,898
	7. 11,784	7. 126	7. 11	7. 11	7. 9,107	7. 0	7. 0	7. 4.0e-2	7. 0	7. 367,843
	8. 11,837	8. 5.6e-1	8. 0	8. 266	8. 8,137	8. 302	8. 166	8. 1	8. 0	8. 369,906
	9. 11,782	9. 7.7e-1	9. 0	9. 184	9. 8,176	9. 297	9. 540	9. 1	9. 0	9. 367,816
Vacation	1. 8,601	1. 7,001	1. 0	1. 19,950	1. 70,074	1. 0	1. 0	1. 20,802	1. 5,653	1. 14,130
	2. 5,395	2. 254	2. 0	2. 1,218	2. 69,765	2. 0	2. 0	2. 129	2. 4,106	2. 10,843
	3. 5,386	3. 262	3. 0	3. 1,207	3. 69,746	3. 0	3. 0	3. 97	3. 4,037	3. 10,828
	4. 41,972	4. 517,589	4. 0	4. 3,107	4. 80,526	4. 0	4. 0	4. 2,334	4. 20,572	4. 47,426
	5. 21,888	5. 168	5. 0	5. 703	5. 49,632	5. 20,045	5. 156,373	5. 50	5. 95,816	5. 27,427
	6. 5,388	6. 794	6. 0	6. 691	6. 49,403	6. 20,012	6. 278	6. 84	6. 4,158	6. 10,797
	7. 5,742	7. 109	7. 324	7. 2,641	7. 70,780	7. 0	7. 0	7. 3	7. 4,213	7. 13,809
	8. 5,442	8. 755	8. 0	8. 686	8. 49,365	8. 20,007	8. 1,249	8. 78	8. 4,065	8. 10,866
	9. 5,596	9. 723	9. 0	9. 698	9. 49,378	9. 20,022	9. 3,426	9. 78	9. 4,184	9. 11,022

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-4. Raw cycles (in thousands) for Genome, Delaunay, Bayes

Workload	Ruby cycles	Stalls	Aborts	Wasted Trans.	Useful Trans.	Committing Stores in Wr. Buffer	Commit Token Stalls	Backoff	Barrier	Nontrans.
Genome	1. 4,958	1. 60	1. 0	1. 101	1. 42,324	1. 0	1. 0	1. 5	1. 16,653	1. 20,186
	2. 5,100	2. 30	2. 0	2. 96	2. 42,353	2. 0	2. 0	2. 3	2. 18,073	2. 21,054
	3. 5,064	3. 29	3. 0	3. 103	3. 42,388	3. 0	3. 0	3. 3	3. 17,635	3. 20,865
	4. 5,318	4. 506	4. 0	4. 51	4. 42,891	4. 0	4. 0	4. 2	4. 20,154	4. 21,480
	5 .8,820	5. 10	5. 0	5. 214	5. 37,469	5. 4,909	5. 31,938	5. 138	5. 42,227	5. 24,223
	6. 5,135	6. 44	6. 0	6. 82	6. 37,364	6. 4,943	6. 9	6. 6	6. 18,372	6. 21,347
	7. 4,983	7. 21	7. 27	7. 111	7. 42,469	7. 0	7. 0	7. 2	7. 16,972	7. 20,123
	8. 4,701	8. 49	8. 0	8. 83	8. 37,348	8. 4,943	8. 660	8. 7	8. 13,676	8. 18,449
	9. 4,884	9. 45	9. 0	9. 78	9. 37,352	9. 4,943	9. 2,133	9. 6	9. 14,829	9. 18,756
Delaunay	1. 18,264	1. 93,259	1. 0	1. 65,292	1. 74,951	1. 0	1. 0	1. 4,350	1. 19,820	1. 34,551
	2. 12,338	2. 58,47	2. 0	2. 13,626	2. 74,384	2. 0	2. 0	2. 101	2. 20,291	2. 30,587
	3. 12,046	3. 58,302	3. 0	3. 10,014	3. 74,851	3. 0	3. 0	3. 90	3. 21,478	3. 28,005
	4. 15,097	4. 103,859	4. 0	4. 10,206	4. 74,132	4. 0	4. 0	4. 178	4. 20,163	4. 33,011
	5. 13,978	5. 3,340	5. 0	5. 58,812	5. 64,373	5. 9,226	5. 36,055	5. 348	5. 19,031	5. 32,459
	6. 11,720	6. 4,807	6. 0	6. 57,678	6. 64,254	6. 9,222	6. 298	6. 156	6. 21,767	6. 29,342
	7. 12,160	7. 57,139	7. 2,791	7. 9,231	7. 75,240	7. 0	7. 0	7. 12	7. 20,493	7. 29,654
	8. 11,679	8. 4,817	8. 0	8. 59,868	8. 64,477	8. 9,266	8. 408	8. 165	8. 18,376	8. 29,487
	9. 11,829	9. 4,946	9. 0	9. 61,520	9. 64,737	9. 9,321	9. 612	9. 171	9. 18,925	9. 29,032
Bayes	1. 20,805	1. 32,306	1. 0	1. 99,533	1. 42,372	1. 0	1. 0	1. 72,888	1. 48,329	1. 37,452
	2. 16,676	2. 24,329	2. 0	2. 28,659	2. 47,324	2. 0	2. 0	2. 59,245	2. 73,931	2. 33,337
	3. 24,398	3. 30,891	3. 0	3. 28,221	3. 55,361	3. 0	3. 0	3. 58,282	3. 176,548	3. 41,066
	4. 17,262	4. 31,057	4. 0	4. 23,206	4. 45,917	4. 0	4. 0	4. 46,939	4. 95,142	4. 33,927
	5. 14,030	5. 18	5. 0	5. 50,318	5. 43,352	5. 473	5. 655	5. 166	5. 98,804	5. 30,694
	6. 13,809	6. 22	6. 0	6. 48,537	6. 43,340	6. 475	6. 7	6. 170	6. 97,903	6. 30,485
	7. 14,241	7. 16,364	7. 3,979	7. 30,948	7. 69,118	7. 0	7. 0	7. 16	7. 63,871	7. 43,565
	8. 13,906	8. 23	8. 0	8. 49,393	8. 43,278	8. 473	8. 29	8. 168	8. 98,547	8. 30,582
	9. 13,728	9. 26	9. 0	9. 46,925	9. 43,478	9. 477	9. 83	9. 191	9. 98,069	9. 30,405

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-5. Raw cycles (in thousands) for Labyrinth, Intruder, Yada

Workload	Ruby cycles	Stalls	Aborts	Wasted Trans.	Useful Trans.	Committing Stores in Wr. Buffer	Commit Token Stalls	Backoff	Barrier	Nontrans.
Labyrinth	1. 17,307	1. 542	1. 0	1. 106,467	1. 59,683	1. 0	1. 0	1. 144	1. 89,666	1. 20,404
	2. 17,194	2. 196	2. 0	2. 101,508	2. 59,071	2. 0	2. 0	2. 51	2. 93,995	2. 20,276
	3. 17,705	3. 145	3. 0	3. 102,414	3. 59,048	3. 0	3. 0	3. 50	3. 100,819	3. 20,811
	4. 45,116	4. 530,336	4. 0	4. 23,772	4. 60,344	4. 0	4. 0	4. 79	4. 59,108	4. 48,215
	5 .14,696	5. 4	5. 0	5. 88,307	5. 58,944	5. 161	5. 33	5. 22	5. 69,905	5. 17,763
	6. 14,723	6. 8	6. 0	6. 88,699	6. 58,989	6. 162	6. 7.1e-1	6. 23	6. 69,853	6. 17,841
	7. 15,441	7. 1,518	7. 3,363	7. 85,488	7. 58,711	7. 0	7. 0	7. 17	7. 75,893	7. 22,066
	8. 14,964	8. 6	8. 0	8. 89,046	8. 58,950	8. 162	8. 7	8. 25	8. 73,173	8. 18,053
	9. 14,979	9. 6	9. 0	9. 89,168	9. 58,970	9. 162	9. 22	9. 27	9. 73,224	9. 18,087
Intruder	1. 36,489	1. 131,522	1. 0	1. 137,570	1. 32,003	1. 0	1. 0	1. 198,250	1. 19,376	1. 65,104
	2. 8,302	2. 16,419	2. 0	2. 22,613	2. 32,258	2. 0	2. 0	2. 7,290	2. 18,076	2. 36,172
	3. 7,602	3. 14,437	3. 0	3. 16,754	3. 32,418	3. 0	3. 0	3. 4,783	3. 17,954	3. 35,283
	4. 6,294	4. 8,316	4. 0	4. 9,082	4. 30,732	4. 0	4. 0	4. 513	4. 18,152	4. 33,915
	5. 15,214	5. 402	5. 0	5. 29,647	5. 17,110	5. 12,025	5. 90,832	5. 32,508	5. 18,008	5. 42,888
	6. 7,082	6. 1,097	6. 0	6. 23,490	6. 17,585	6. 11,918	6. 2,618	6. 3,606	6. 18,375	6. 34,631
	7. 8,673	7. 14,979	7. 6,825	7. 22,053	7. 36,343	7. 0	7. 0	7. 225	7. 19,502	7. 38,811
	8. 7,216	8. 1,110	8. 0	8. 23,966	8. 17,462	8. 11,910	8. 4,077	8. 3,864	8. 18,219	8. 34,850
	9. 7,615	9. 1,069	9. 0	9. 24,911	9. 17,492	9. 11,897	9. 8,270	9. 4,626	9. 18,437	9. 35,143
Yada	1. 34,983	1. 203,455	1. 0	1.131,245	1. 157,266	1. 0	1. 0	1. 2,925	1. 17,546	1. 47,296
	2. 27,128	2. 148,862	2. 0	2. 73,038	2. 158,589	2. 0	2. 0	2. 1,788	2. 12,311	2. 39,468
	3. 26,507	3. 149,270	3. 0	3. 61,685	3. 158,841	3. 0	3. 0	3. 1,483	3. 13,803	3. 39,031
	4. 92,045	4. 917,517	4. 0	4. 268,475	4. 155,349	4. 0	4. 0	4. 6,080	4. 20,414	4. 104,884
	5. 31,434	5. 9,016	5. 0	5. 165,721	5. 133,633	5. 22,314	5. 114,375	5. 1,257	5. 13,877	5. 42,754
	6. 23,877	6. 12,069	6. 0	6. 163,530	6. 134,393	6. 22,291	6. 922	6. 488	6. 11,250	6. 37,096
	7. 27,115	7. 142,728	7. 21,899	7. 46,611	7. 165,429	7. 0	7. 0	7. 17	7. 13,460	7. 43,698
	8. 23,634	8. 12,120	8. 0	8. 162,319	8. 132,771	8. 22,053	8. 1,090	8. 453	8. 10,450	8. 36,896
	9. 24,163	9. 12,488	9. 0	9. 166,973	9. 134,149	9. 22,279	9. 1,598	9. 471	9. 11,715	9. 36,938

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-6. Stall cycles (in thousands) for Barnes, Raytrace, Mp3d

Workload	WW Req. Younger	WW Req. Older	WR Req. Younger	WR Req. Older	RW Req. Younger	RW Req. Older	Remaining ACK Trans.	NonTrans.	Remaining ACK NonTrans.	Replacement
Barnes	1. 33	1. 31	1. 988	1. 270	1. 352	1. 771	1. 80	1. 0	1. 12	1. 0
	2. 31	2. 9.7e-1	2. 1,259	2. 3	2. 248	2. 0	2. 39	2. 756	2. 10	2. 0
	3. 53	3. 4	3. 1,359	3. 73	3. 226	3. 0	3. 36	3. 749	3. 10	3. 0
	4. 456	4. 83	4. 624	4. 60	4. 289	4. 0	4. 37	4. 3,371	4. 29	4. 0
	5. 0	5. 0	5. 108	5. 25	5. 0	5. 0	5. 2	5. 49	5. 8.0e-1	5. 0
	6. 0	6. 0	6. 202	6. 48	6. 4	6. 5.1e-1	6. 4	6. 75	6. 1	6. 0
	7. 46	7. 32	7. 1,253	7. 176	7. 277	7. 0	7. 39	7. 1,066	7. 11	7. 0
	8. 0	8. 0	8. 226	8. 63	8. 3	8. 4.5e-1	8. 4	8. 93	8. 1	8. 0
	9. 0	9. 0	9. 207	9. 61	9. 3	9. 2.5e-1	9. 4	9. 91	9. 1	9. 0
Raytrace	1. 0	1. 0	1. 65	1. 5	1. 8,290	1. 8,553	1. 365	1. 7	1. 1.1e-1	1. 0
	2. 3.4e-1	2. 7.1e-2	2. 13	2. 2	2. 420	2. 0	2. 39	2. 5	2. 9.8e-2	2. 0
	3. 6.1e-1	3. 1.0e-1	3. 24	3. 5	3. 1,351	3. 0	3. 112	3. 4	3. 7.5e-2	3. 0
	4. 6,988	4. 4,715	4. 9,139	4. 5,834	4. 6,685	4. 0	4. 713	4. 54	4. 1	4. 1e-3
	5. 0	5. 0	5. 9	5. 2.7e-1	5. 0	5. 0	5. 1.2e-1	5. 3	5. 4.7e-2	5. 0
	6. 0	6. 0	6. 9	6. 4.0e-1	6. 4.3e-2	6. 0	6. 1.4e-1	6. 3	6. 4.1e-2	6. 0
	7. 9.8e-2	7. 0	7. 10	7. 10	7. 1,588	7. 0	7. 115	7. 3	7. 5.4e-2	7. 0
	8. 0	8. 0	8. 8	8. 3	8. 4.7e-2	8. 0	8. 2.3e-1	8. 1.1e-1	8. 3e-3	8. 0
	9. 0	9. 0	9. 22	9. 11	9. 6.3e-2	9. 0	9. 7.2e-1	9. 2.1e-1	9. 5e-3	9. 0
Mp3d	1. 0	1. 0	1. 1,854	1. 290	1. 16,283	1. 18,308	1. 625	1. 72	1. 2	1. 0
	2. 851	2. 1	2. 1,498	2. 241	2. 6,750	2. 0	2. 132	2. 132	2. 4	2. 0
	3. 697	3. 1	3. 1,413	3. 246	3. 6,719	3. 0	3. 126	3. 130	3. 4	3. 0
	4. 2,517	4. 1,107	4. 56	4. 16	4. 46	4. 0	4. 39	4. 367	4. 5	4. 0
	5. 0	5. 0	5. 67	5. 53	5. 0	5. 0	5. 2	5. 1	5. 7.4e-2	5. 0
	6. 0	6. 0	6. 83	6. 58	6. 0	6. 0	6. 2	6. 2	6. 8.1e-2	6. 0
	7. 471	7. 1.8e-1	7. 1,112	7. 477	7. 2,913	7. 0	7. 94	7. 168	7. 5	7. 0
	8. 0	8. 0	8. 106	8. 58	8. 4.0e-3	8. 0	8. 2	8. 2	8. 1.2e-1	8. 0
	9. 0	9. 0	9. 129	9. 85	9. 0	9. 0	9. 4	9. 4	9. 3.6e-1	9. 0

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-7. Stall cycles (in thousands) for Sparse Matrix, BTree, LFUCache

Workload	WW Req. Younger	WW Req. Older	WR Req. Younger	WR Req. Older	RW Req. Younger	RW Req. Older	Remaining ACK Trans.	NonTrans.	Remaining ACK NonTrans.	Replacement
Sparse Matrix	1. 0	1. 0	1. 3e-3	1. 2.9e-2	1. 2	1. 5	1. 8.7e-1	1. 4	1. 2.5e-1	1. 0
	2. 0	2. 0	2. 3e-3	2. 1.4e-2	2. 3	2. 0	2. 4.9e-1	2. 2	2. 2.5e-1	2. 0
	3. 0	3. 0	3. 3e-3	3. 1.1e-2	3. 3	3. 0	3. 4.7e-1	3. 1	3. 1.8e-1	3. 0
	4. 7e-3	4. 1.4e-1	4. 0	4. 0	4. 6.8e-1	4. 0	4. 3.8e-1	4. 1	4. 1.1e-1	4. 0
	5. 0	5. 0	5. 4e-3	5. 1.2e-2	5. 0	5. 0	5. 0	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 0	6. 7e-3	6. 6e-3	6. 0	6. 6.0e-2	6. 0	6. 0	6. 0
	7. 0	7. 0	7. 5e-3	7. 4e-3	7. 2	7. 0	7. 4.6e-1	7. 2	7. 1.9e-1	7. 0
	8. 0	8. 0	8. 1.4e-1	8. 2.3e-2	8. 2.2e-2	8. 0	8. 2.8e-1	8. 8e-3	8. 1.1e-1	8. 0
	9. 0	9. 0	9. 6.6e-1	9. 1	9. 2.8e-2	9. 0	9. 3.7e-1	9. 1.9e-2	9. 2.2e-1	9. 0
BTree	1. 0	1. 0	1. 354	1. 4.5e-2	1. 48	1. 376	1. 32	1. 0	1. 0	1. 0
	2. 8.4e-1	2. 0	2. 425	2. 0	2. 7	2. 0	2. 5	2. 0	2. 0	2. 0
	3. 8.1e-1	3. 0	3. 371	3. 1.5e-2	3. 15	3. 0	3. 8	3. 0	3. 0	3. 0
	4. 534,055	4. 85,325	4. 521,468	4. 406,105	4. 290,497	4. 0	4. 4,961	4. 0	4. 0	4. 0
	5. 0	5. 0	5. 48	5. 2	5. 2.5e-2	5. 0	5. 9.4e-1	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 49	6. 2	6. 1.9e-1	6. 0	6. 8.5e-1	6. 0	6. 0	6. 0
	7. 4.8e-1	7. 0	7. 270	7. 1.9e-2	7. 10	7. 0	7. 6	7. 0	7. 0	7. 0
	8. 0	8. 0	8. 48	8. 2	8. 4.3e-1	8. 8.5e-2	8. 2	8. 0	8. 0	8. 0
	9. 0	9. 0	9. 52	9. 2	9. 1.2e-1	9. 1.0e-1	9. 1	9. 0	9. 0	9. 0
LFUCache	1. 1	1. 5.2e-1	1. 1,626	1. 475	1. 16,150	1. 17,530	1. 593	1. 0	1. 0	1. 0
	2. 321	2. 5.3e-1	2. 2,205	2. 53	2. 5,977	2. 0	2. 87	2. 0	2. 0	2. 0
	3. 246	3. 2.9e-1	3. 1,915	3. 190	3. 4,459	3. 0	3. 66	3. 0	3. 0	3. 0
	4. 1,133	4. 115	4. 1,092	4. 67	4. 476	4. 0	4. 48	4. 3.7e-1	4. 8	4. 0
	5. 0	5. 0	5. 31	5. 1	5. 0	5. 0	5. 5.7e-1	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 31	6. 1	6. 1	6. 0	6. 7.0e-1	6. 0	6. 0	6. 0
	7. 103	7. 1	7. 1,723	7. 170	7. 1,307	7. 0	7. 32	7. 0	7. 0	7. 0
	8. 0	8. 0	8. 33	8. 1	8. 1	8. 1.5e-2	8. 8.2e-1	8. 0	8. 0	8. 0
	9. 0	9. 0	9. 39	9. 2	9. 2	9. 3.8e-2	9. 9.4e-1	9. 0	9. 0	9. 0

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-8. Stall cycles (in thousands) for Prefetch, BIND, Vacation

Workload	WW Req. Younger	WW Req. Older	WR Req. Younger	WR Req. Older	RW Req. Younger	RW Req. Older	Remaining ACK Trans.	NonTrans.	Remaining ACK NonTrans.	Replacement
Prefetch	1. 47,525	1. 39,131	1. 15,627	1. 7,500	1. 4,927	1. 5,344	1. 121	1. 0	1. 0	1. 0
	2. 86,938	2. 4,734	2. 20,490	2. 1,434	2. 5,607	2. 0	2. 128	2. 0	2. 0	2. 2e-3
	3. 47,544	3. 20,694	3. 17,798	3. 20,822	3. 4,794	3. 0	3. 98	3. 0	3. 0	3. 0
	4. 57,983	4. 47,181	4. 1,600	4. 3,614	4. 398	4. 0	4. 82	4. 0	4. 0	4. 7e-3
	5. 0	5. 0	5. 0	5. 0	5. 0	5. 0	5. 0	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 0	6. 0	6. 0	6. 0	6. 0	6. 0	6. 0	6. 0
	7. 42,398	7. 19,464	7. 13,909	7. 18,335	7. 3,702	7. 0	7. 98	7. 0	7. 0	7. 0
	8. 0	8. 0	8. 0	8. 0	8. 0	8. 0	8. 0	8. 0	8. 0	8. 0
	9. 0	9. 0	9. 0	9. 0	9. 0	9. 0	9. 0	9. 0	9. 0	9. 0
BIND	1. 0	1. 0	1. 6	1. 4e-3	1. 177	1. 3	1. 10	1. 0	1. 0	1. 0
	2. 2.0e-1	2. 0	2. 4	2. 3.6e-1	2. 109	2. 0	2. 8	2. 6.3e-2	2. 8	2. 0
	3. 1.3e-1	3. 0	3. 1	3. 3.6e-2	3. 138	3. 0	3. 9	3. 2.0e-2	3. 0	3. 0
	4. 1	4. 0	4. 5	4. 6.8e-1	4. 84	4. 0	4. 3	4. 4.4e-1	4. 7e-3	4. 0
	5. 0	5. 0	5. 1.7e-1	5. 1.7e-2	5. 0	5. 0	5. 3e-3	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 3.7e-2	6. 0	6. 0	6. 0	6. 1e-3	6. 0	6. 0	6. 0
	7. 1.5e-1	7. 0	7. 1	7. 1.4e-1	7. 115	7. 0	7. 9	7. 2.4e-1	7. 3.7e-2	7. 0
	8. 0	8. 0	8. 1.6e-1	8. 3.2e-2	8. 3.5e-1	8. 0	8. 8e-3	8. 0	8. 0	8. 0
	9. 0	9. 0	9. 4.3e-1	9. 1.7e-2	9. 3.1e-1	9. 0	9. 1.1e-2	9. 0	9. 0	9. 0
Vacation	1. 0	1. 0	1. 1,238	1. 6	1. 662	1. 4,801	1. 293	1. 0	1. 0	1. 0
	2. 1	2. 0	2. 226	2. 8.9e-2	2. 21	2. 0	2. 5	2. 0	2. 0	2. 0
	3. 1	3. 0	3. 238	3. 2	3. 16	3. 0	3. 4	3. 0	3. 0	3. 0
	4. 259,071	4. 215,093	4. 17,018	4. 14,861	4. 9,884	4. 0	4. 1,662	4. 0	4. 0	4. 0
	5. 0	5. 0	5. 123	5. 41	5. 1.6e-1	5. 0	5. 3	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 671	6. 53	6. 28	6. 24	6. 18	6. 0	6. 0	6. 0
	7. 8.7e-1	7. 3.4e-1	7. 81	7. 6	7. 17	7. 0	7. 3	7. 0	7. 0	7. 0
	8. 0	8. 0	8. 637	8. 47	8. 28	8. 24	8. 18	8. 0	8. 0	8. 0
	9. 0	9. 0	9. 609	9. 51	9. 24	9. 22	9. 17	9. 0	9. 0	9. 0

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-9. Stall cycles (in thousands) for Genome, Delaunay, Bayes

Workload	WW Req. Younger	WW Req. Older	WR Req. Younger	WR Req. Older	RW Req. Younger	RW Req. Older	Remaining ACK Trans.	NonTrans.	Remaining ACK NonTrans.	Replacement
Genome	1. 0	1. 0	1. 23	1. 1.8e-1	1. 9	1. 24	1. 4	1. 1.7e-1	1. 4.0e-3	1. 0
	2. 3.1e-1	2. 0	2. 19	2. 2.6e-1	2. 8	2. 0	2. 1	2. 1.4e-1	2. 3.0e-3	2. 0
	3. 3.0e-1	3. 0	3. 20	3. 1.9e-1	3. 8	3. 0	3. 1	3. 1.2e-1	3. 3.0e-3	3. 0
	4. 323	4. 82	4. 37	4. 11	4. 38	4. 0	4. 14	4. 3.0e-1	4. 7.0e-3	4. 0
	5. 0	5. 0	5. 8	5. 2	5. 0	5. 0	5. 2.1e-1	5. 6.9e-2	5. 2.0e-3	5. 0
	6. 0	6. 0	6. 36	6. 7	6. 0	6. 0	6. 8.7e-1	6. 4.5e-2	6. 1.0e-3	6. 0
	7. 2.4e-1	7. 0	7. 12	7. 4.9e-1	7. 7	7. 0	7. 1	7. 2.2e-1	7. 5.0e-3	7. 0
	8. 0	8. 0	8. 41	8. 7	8. 0	8. 0	8. 9.6e-1	8. 6.7e-2	8. 1.0e-3	8. 0
	9. 0	9. 0	9. 38	9. 5	9. 0	9. 0	9. 8.9e-1	9. 9.9e-2	9. 2.0e-3	9. 0
Delaunay	1. 1.2e-2	1. 8.0e-3	1. 62,433	1. 3,981	1. 6,316	1. 16,923	1. 3,606	1. 0	1. 0	1. 8.0e-3
	2. 1	2. 7.0e-3	2. 54,732	2. 11	2. 1,937	2. 0	2. 1,734	2. 0	2. 0	2. 4.0e-3
	3. 2	3. 0	3. 51,745	3. 3,234	3. 1,669	3. 0	3. 1,651	3. 0	3. 0	3. 0
	4. 58,712	4. 3,456	4. 31,728	4. 2,615	4. 4,320	4. 0	4. 3,028	4. 0	4. 0	4. 0
	5. 1.8e-1	5. 0	5. 2,555	5. 712	5. 1	5. 1	5. 70	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 3,511	6. 1,150	6. 38	6. 6	6. 103	6. 0	6. 0	6. 0
	7. 1	7. 1.1e-1	7. 50,324	7. 3,563	7. 1,564	7. 0	7. 1,686	7. 0	7. 0	7. 0
	8. 1	8. 8.9e-1	8. 3,414	8. 1,235	8. 49	8. 8	8. 108	8. 0	8. 0	8. 8.0e-3
	9. 0	9. 0	9. 3,563	9. 1,224	9. 41	9. 9	9. 109	9. 0	9. 0	9. 8.0e-3
Bayes	1. 0	1. 0	1. 138	1. 10	1. 2,139	1. 26,712	1. 3,306	1. 0	1. 0	1. 0
	2. 11	2. 0	2. 348	2. 11	2. 21,664	2. 0	2. 2,294	2. 0	2. 0	2. 0
	3. 10	3. 0	3. 438	3. 17	3. 27,951	3. 0	3. 2,474	3. 0	3. 0	3. 0
	4. 2,305	4. 253	4. 3,831	4. 1,011	4. 21,269	4. 0	4. 2,388	4. 0	4. 0	4. 0
	5. 0	5. 0	5. 11	5. 7	5. 0	5. 0	5. 3.3e-1	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 11	6. 7	6. 3	6. 0	6. 6.9e-1	6. 0	6. 0	6. 0
	7. 4	7. 0	7. 71	7. 9	7. 14,658	7. 0	7. 1,621	7. 0	7. 0	7. 0
	8. 0	8. 0	8. 13	8. 8	8. 1	8. 0	8. 5.1e-1	8. 0	8. 0	8. 0
	9. 0	9. 0	9. 12	9. 7	9. 5	9. 0	9. 7.8e-1	9. 0	9. 0	9. 0

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

TABLE C-10. Stall cycles (in thousands) for Labyrinth, Intruder, Yada

Workload	WW Req. Younger	WW Req. Older	WR Req. Younger	WR Req. Older	RW Req. Younger	RW Req. Older	Remaining ACK Trans.	NonTrans.	Remaining ACK NonTrans.	Replacement
Labyrinth	1. 0	1. 0	1. 15	1. 11	1. 139	1. 336	1. 41	1. 0	1. 0	1. 0
	2. 3	2. 0	2. 27	2. 1	2. 146	2. 0	2. 17	2. 0	2. 0	2. 0
	3. 3	3. 0	3. 28	3. 3	3. 100	3. 0	3. 10	3. 0	3. 0	3. 0
	4. 10,374	4. 6,877	4. 300,343	4. 205,862	4. 1,191	4. 0	4. 5,689	4. 0	4. 0	4. 0
	5. 0	5. 0	5. 4	5. 2.4e-1	5. 0	5. 0	5. 9.4e-2	5. 0	5. 0	5. 0
	6. 0	6. 0	6. 8	6. 2.5e-1	6. 0	6. 0	6. 1.9e-1	6. 0	6. 0	6. 0
	7. 1	7. 0	7. 911	7. 427	7. 140	7. 0	7. 39	7. 0	7. 0	7. 0
	8. 0	8. 0	8. 5	8. 2.7e-1	8. 0	8. 0	8. 0	8. 1.3e-1	8. 0	8. 0
	9. 0	9. 0	9. 5	9. 1.6e-1	9. 0	9. 0	9. 1.1e-1	9. 0	9. 0	9. 0
Intruder	1. 326	1. 231	1. 41,059	1. 1,461	1. 35,078	1. 50,260	1. 2,424	1. 657	1. 25	1. 0
	2. 2,397	2. 122	2. 8,046	2. 344	2. 5,293	2. 0	2. 166	2. 48	2. 2	2. 0
	3. 2,111	3. 94	3. 7,512	3. 436	3. 4,064	3. 0	3. 143	3. 73	3. 3	3. 0
	4. 4,366	4. 792	4. 1,476	4. 642	4. 885	4. 0	4. 101	4. 52	4. 2	4. 0
	5. 0	5. 0	5. 285	5. 104	5. 0	5. 0	5. 7	5. 5	5. 1.4e-1	5. 0
	6. 0	6. 0	6. 797	6. 230	6. 16	6. 22	6. 21	6. 10	6. 2.3e-1	6. 0
	7. 1,838	7. 2,141	7. 6,644	7. 1,096	7. 3,010	7. 0	7. 196	7. 50	7. 2	7. 0
	8. 0	8. 0	8. 806	8. 232	8. 17	8. 22	8. 22	8. 10	8. 2.2e-1	8. 0
	9. 0	9. 0	9. 766	9. 235	9. 15	9. 22	9. 22	9. 9	9. 1.9e-1	9. 0
Yada	1. 8,414	1. 304	1. 95,046	1. 13,550	1. 23,404	1. 53,864	1. 8,873	1. 0	1. 0	1. 1.1e-2
	2. 143	2. 6.5e-1	2. 132,538	2. 43	2. 11,161	2. 0	2. 4,976	2. 0	2. 0	2. 0
	3. 314	3. 81	3. 126,904	3. 7,101	3. 9,923	3. 0	3. 4,946	3. 0	3. 0	3. 4.0e-3
	4. 479,966	4. 86,775	4. 214,633	4. 26,664	4. 84,538	4. 0	4. 24,940	4. 0	4. 0	4. 0
	5. 0	5. 3.1e-1	5. 6,216	5. 2,590	5. 12	5. 6	5. 191	5. 0	5. 0	5. 1.1e-2
	6. 1	6. 2	6. 7,585	6. 3,957	6. 156	6. 94	6. 275	6. 0	6. 0	6. 1.4e-2
	7. 172	7. 124	7. 117,632	7. 10,107	7. 9,625	7. 0	7. 5,067	7. 0	7. 0	7. 4.0e-3
	8. 3	8. 0	8. 7,445	8. 4,159	8. 152	8. 90	8. 270	8. 0	8. 0	8. 1.7e-2
	9. 2	9. 7.1e-1	9. 7,689	9. 4,230	9. 203	9. 78	9. 285	9. 0	9. 0	9. 5.0e-3

Legend: 1. EE_Base_HWBuf 2. EE_T_HWBuf 3. EE_H_HWBuf 4. EE_HP_HWBuf 5. LL_B_S 6. LL_B_P 7. EE_H 8. LL_P_Fast 9. LL_P_Slow

Appendix D

Supplements for (Chapter 7)

TABLE D-1. Raw cycles (in thousands) for perfect signatures

Workload	Ruby cycles
Barnes	15,121
Mp3d	6,410
Raytrace	32,040
BIND	11,802
BTree	11,170
Sparse Matrix	12,190
LFUCache	2,957
Prefetch	5,277
Bayes	18,273
Delaunay	12,139
Genome	5,272
Intruder	8,833
Labyrinth	16,414
Vacation	5,794
Yada	26,332