The Common Case Transactional Behavior of Multithreaded Programs

JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom
Christos Kozyrakis, Kunle Olukotun

Computer Systems Laboratory
Stanford University
{*jwchung, hchafi, caominh, austenmc, bdc, kozyraki, kunle*}*@stanford.edu*

Abstract

Transactional memory (TM) provides an easy-to-use and high-performance parallel programming model for the upcoming chip-multiprocessor systems. Several researchers have proposed alternative hardware and software TM implementations. However, the lack of transaction-based programs makes it difficult to understand the merits of each proposal and to tune future TM implementations to the common case behavior of real application.

This work addresses this problem by analyzing the common case transactional behavior for 35 multithreaded programs from a wide range of application domains. We identify transactions within the source code by mapping existing primitives for parallelism and synchronization management to transaction boundaries. The analysis covers basic characteristics such as transaction length, distribution of read-set and write-set size, and the frequency of nesting and I/O operations. The measured characteristics provide key insights into the design of efficient TM systems for both non-blocking synchronization and speculative parallelization.

1. Introduction

With most hardware vendors shipping chip-multiprocessors (CMPs) for the desktop, embedded, and server markets [15, 16, 24], mainstream applications must become concurrent to take advantage of the multiple cores [38]. Traditionally, programmers have associated locks with shared data in order to synchronize concurrent accesses. However, locks have well-known software engineering issues that make parallel programming too complicated for the average developer [39]. The programmer must choose between the easy-to-use coarse-grain locking that leads to unnecessary blocking and the scalable fine-grain locking that requires complex coding conventions to avoid deadlocks, priority inversion, or convoying. Moreover, lock-based code does not automatically compose and is not robust to hardware or software failures.

Transactional Memory (TM) [14] provides an alternative model for concurrency management. A TM system operates on shared data using sequences of instructions (transactions) which execute in an atomic and isolated manner [8]. Transactional memory simplifies parallel programming by providing non-blocking synchronization with easy-to-write, coarse-grain transactions by virtue of optimistic concurrency [17]. It also allows for speculative parallelization of sequential code [9]. Furthermore, transactional code is composable and robust in the face of failures.

The significant advantages of the TM model have motivated several proposals for hardware-based [29, 23, 2, 25] or software-only [34, 12, 13, 22, 31, 32] implementations. Some proposals advocate continuous transactional execution in parallel systems [10, 18]. The various TM designs suggest different tradeoffs in the mechanisms used to track transactional state and detect conflicts, the size and location of buffers, and the overheads associated with basic operations. At this point, however, there is not enough transaction-based software to properly evaluate the merit of each proposal. System designers need transactional code to tune their TM designs to the common case behavior of real applications. At the same time, application developers are waiting for efficient and complete TM system before they port a significant volume of applications to the new concurrency model.

This paper attempts to break the deadlock. We study a wide range of existing multithreaded applications and analyze the likely common-case behavior of their future transactional versions. The basic premise of our approach is that the high-level parallelism and synchronization characteristics in an application are unlikely to change significantly regardless of the mechanism used to manage them (locks or transactions). Our study involves 35 multithreaded programs from a wide range of applications domains, written in four parallel programming models. To define transac-

tion boundaries in the source code, we examine the primitives used for concurrency control in each parallel model and re-cast their meaning in a transactional context for both non-blocking synchronization and speculative parallelization. Then, we trace and analyze each application to measure basic characteristics such as the transaction length, the read-set and write-set size, and the frequency of nested transactions and I/O operations. The characteristics provide key insights into the common case support necessary to implement an efficient TM system.

The major observations from our analysis are:

- Transactions are mostly short. Hence, the fixed overheads associated with starting and ending transactions must be minimized. Short transactions also allow us to handle interrupts and context switches efficiently without the need for complex hardware mechanisms for TM virtualization.

- The read-sets and write-sets for most transactions fit in L1-sized buffers. However, read-/write- set buffering in L2-sized buffers is necessary to avoid frequent overflows on the longer transactions, especially in the case of speculative parallelism. If L2 caches can store transaction read-/write- sets, complex hardware mechanisms for TM virtualization can be replaced by simple, software-only alternatives.

- When used for non-blocking synchronization, many transactions access a large number of unique addresses compared to their length. Hence, the overhead of buffering, committing, or rolling back per unique address must be minimized. This issue is not aggravated when transactions are used continuously or when they support speculative parallelization.

- Nested transactions occur in system code but rarely in user code. Since the nesting depth is low, hardware support for nested transactions can be limited to a couple of nesting levels.

- I/O operations within transactions are also rare. The observed I/O patterns are easy to handle through I/O buffering techniques.

The rest of the paper is organized as follows. Section 2 summarizes transactional memory and discusses the major design tradeoffs that motivate this study. In Section 3, we present our experimental methodology for analyzing multithreaded applications within a transactional context. Section 4 and 5 present our analysis results when transactions are used for non-blocking synchronization and speculative parallelization respectively. We conclude the paper in Section 6.

2. Transactional Memory

2.1 Transactional Memory Overview

With transactional memory, programmers define atomic blocks of code (transactions) that can include unstructured flow-control and any number of memory accesses. A TM system executes atomic blocks in a manner that preserves the following properties: a) *atomicity*: either the whole transaction executes or none of it; b) *isolation*: partial memory updates are not visible to other transactions; and c) *consistency*: there is a single order of completion for transactions across the whole system [14]. TM systems achieve high performance through optimistic concurrency [17]. A transaction runs without acquiring locks, optimistically assuming no other transaction operates concurrently on the same data. If that assumption is true by the end of its execution, the transaction commits its updates to shared memory. If interference between transactions is detected, then the transaction aborts, its updates so far are rolled back, and it is re-executed from scratch.

A TM system must implement the following mechanisms: (1) *speculative buffering* of stores (*write-set*) until the transaction commits; (2) *conflict detection* between concurrent transactions; (3) *atomic commit* of transaction stores to shared memory; (4) *rollback* of transaction stores when conflicts are detected. Conflict detection requires tracking the addresses read by each transaction (*read-set*). A conflict occurs when the write-set of a committing transaction overlaps with the read-set of an executing transaction. The mechanisms can be implemented either in hardware (HTM) [29, 23, 2, 25] or software (STM) [34, 12, 13, 22, 31, 32]. HTM minimizes the overhead of the basic mechanisms, which is likely to lead to higher performance. STM runs on stock processors and provides implementation flexibility.

HTM systems implement speculative buffering in the processor caches using a store-buffer [29, 23] or an undo-log [2, 25]. A store-buffer allows for truly non-blocking TM but defers all memory updates until the transaction commits. An undo-log accelerates commits by updating shared memory as the transaction executes, but has slower aborts, incurs per access overheads, and introduces blocking and deadlock issues. Since caches have limited capacity and buffer/log overflow is possible, some HTM systems [2, 30] provide elaborate hardware mechanisms to extend the buffer/log into virtual memory (space virtualization). The same mechanisms can be used to context switch a thread in the middle of a transaction if an interrupt occurs (time virtualization). Read-sets are also tracked in caches and conflicts are detected using the cache coherence protocol. Transaction roll-back requires flushing the transaction write-set and read-set from caches.

STM systems implement buffering, conflict detection, commit, and abort entirely in software using runtime primitives. Apart from defining transaction boundaries, STM requires a runtime call at least once per unique address read or written in a transaction so that read-set and write-set can be tracked. Again, speculative buffering can be implemented using a write-buffer [12] or an undo-log [32]. For conflict detection, the STM runtime can either acquire read-write locks during the transaction execution (faster detection) or acquire write locks only and validate the version of all read data before commit (reduced interference at increased commit overhead) [12]. Read-set and write-set tracking can be at the granularity of words, cache-lines, or objects. Commit and abort require walking through the structures that track read-set and write-set to validate, copy, or roll back values, depending on the implementation approach. Buffer/log overflow cannot occur with STM as it operates on top of the virtual memory system.

I/O calls within transactions [11] as well as the semantics and support for nested transactions are topics of active research for both HTM and STM systems.

2.2 Design Challenges & Application Characteristics

It is clear from the above overview that the design of a transactional memory system involves a large number of implementation decisions. As it is common with all systems, the guiding principle for making such decisiosn will be "*make the common case fast*". Designers will look at the common values for basic characteristics of transactional applications to select the right implementation approach for the TM mechanisms, to size buffer components, and to tune the overhead of basic operations. In this section, we review the basic application characteristics for transactional memory and the implementation aspects they interact with.

Transaction Length: Short transactions make it difficult to amortize fixed overheads for starting and committing transactions. On the other hand, long transactions may run into interrupts and require time virtualization support. Long transactions are also more likely to cause rollbacks. This study measures the distribution of transaction lengths in instructions.

Read-set and Write-set Size: The common read-set and write-set sizes dictate the necessary capacity for the buffer used to track them. They also determine the frequency of buffer overflows and whether a fast mechanism is necessary for space virtualization. The granularity for tracking read- and write-sets is also important in order to balance overheads against the frequency of rollbacks due to false sharing. This study measures the distribution of read-/write- sets in words, cache lines, and pages.

Read-/Write- Set to Transaction Length Ratio: The ratio determines if any overhead per unique address read or written in each transaction can be easily amortized. Hence, it helps the designer tune the overhead of basic operations like log update, validateion of a read, and the commit or abort of a store.

Nesting Frequency: The frequency and depth of nested transactions determines the necessary support for nesting in HTM systems. It is also important to understand the potential performance cost from flattening nested transactions, a common approach in current TM designs.

I/O Frequency within Transactions: Non-idempotent I/O within a transaction can be difficult to handle as it cannot be rolled back. Moreover, delaying I/O until the transaction commits may lead to system deadlock. This study does not implement a specific solution, but characterizes the frequency and type of I/O operations within transactions.

In measuring the above application characteristics, it is important to understand that transactions can be used in multiple ways. The original goal for TM has been to provide *non-blocking synchronization* in multiprocessor systems by implementing transactions on top of ordinary cache coherence protocols [14]. Lately, there have been proposals to build TM systems for *continuous transaction execution* to further simplify parallel hardware and software [10, 18]. Finally, the TM mechanisms can support speculative parallelization of sequential code [9]. This study measures application characteristics under all three TM use scenarios.

3. Experimental Methodology

To provide detailed insights into the common case transactional behavior of real programs, we study a large set of existing multithreaded applications. This section describes the experimental methodology, which includes application selection, defining transaction boundaries in multithreaded code, and a trace-based analysis that extracts the characteristics discussed in Section 2.2.

3.1 Multithreaded Applications

Table 1 presents the 35 multithreaded applications we used in this study. The applications were parallelized using four parallel programming models: Java threads [4], C and Pthreads [19], C and OpenMP [28], and the ANL parallel processing macros [21]. Java is increasingly popular and includes multithreading in the base language specification. OpenMP is a widely adopted model based on high-level compiler directives for semi-automatic parallelization. Pthreads is a widely available multithreading package for POSIX systems. Finally, the ANL macros were designed to provide a simple, concise, and portable interface covering a variety of parallel applications. We use the Java, Pthreads, and ANL applications to study the use of transactions for non-blocking synchronization (29 applications). We use the

Prog. Model	Application	Problem Size	Source	Domain/Description
Java	MolDyn	2,048 Particles	JavaGrande	Scientific / Molecular Dynamics
	MonteCarlo	10,000 Runs	JavaGrande	Scientific / Finance
	RayTracer	150x150 Pixels	JavaGrande	Graphics / 3D Raytracer
	Crypt	200,000 Bytes	JavaGrande	Kernel / Encryption and Decryption
	LUFact	500x500 Matrix	JavaGrande	Kernel / Solving NxN Linear System
	Series	200 Coefficients	JavaGrande	Kernel / First N Fourier Coefficients
	SOR	1,000x1,000 Grid	JavaGrande	Kernel / Successive Over-Relaxation
	SparseMatmul	250,000x250,000 Matrix	JavaGrande	Kernel / Matrix Multiplication
	SPECjbb2000	8 Warehouses	SPECjbb2000	Commercial / E-Commerce
	PMD	18 Java Files	DaCapo	Commercial / Java Code Checking
	HSQLDB	10 Tellers, 1,000	DaCapo	Commercial / Banking with hsql database
Pthreads	Apache	20 Worker Threads	Apache	Commercial / HTTP web server
	Kingate	10,000 HTTP Requests	SourceForge	Commercial / Web proxy
	Bp-vision	384x288 Image	Univ. of Chi.	Machine Learning / Loopy Belief Propagation
	Localize	477x177 Map	CARMEN	Robotics / Finding a Robot Position In a Map
	Ultra Tic Tac	5x5 Board, 3 Step	SourceForge	AI / Tic Tac Toe Game
	MPEG2	640x480 Clip	MPEG S.S.G.	MultiMedia / MPEG2 Decoder
	AOL Server	20 Worker Threads	AOL Website	Commercial / HTTP web server
OpenMP	Equake	380K Nodes	SPEComp	Scientific / Seismic Wave Propagation Simulation
	Art	640x480 Image	SPEComp	Scientific / Neural Network Simulation
	CG	1400x1400 Matrix	NAS	Scientific / Conjugate Gradient Method
	BT	12x12x12 Matrix	NAS	Scientific / CFD
	IS	1M Keys	NAS	Scientific / Large-scale Integer Sort
	Swim	1,900x900 Matrix	SPEComp	Scientific / Shallow Water Modeling
ANL Macros	Barnes	16K Particles	SPLASH-2	Scientific / Evolution of Galaxies
	Mp3d	3,000 Molecules, 50 Steps	SPLASH	Scientific / Rarefied Hypersonic Flow
	Ocean	258x258 Ocean	SPLASH-2	Scientific / Eddy Currents in an Ocean Basin
	Radix	1M Ints., Radix 1024	SPLASH-2	Kernel / Radix Sort
	FMM	2,049 Particles	SPLASH-2	Kernel / N-body Simulation
	Cholesky	TK23.0	SPLASH-2	Kernel / Sparse Matrix Factorization
	Radiosity	Room	SPLASH-2	Graphics / Equilibrium of Light Distribution
	FFT	256K points	SPLASH-2	Kernel / 1-D version of the radix-N2 FFT
	Volrend	Head-Scaledown 4	SPLASH-2	Graphics / 3-D Volumn Rendering
	Water-N2	512 molecules	SPLASH-2	Scientific / Evolution of System of Water Molecules
	Water-Spatial	512 molecules	SPLASH-2	Scientific / Evolution of System of Water Molecules

Table 1: The 35 multithreaded applications used in this study.

six OpenMP applications to study the use of transactions for speculative parallelization.

The selected programs cover a wide range of application domains. Apart from scientific computations, the list includes commercial applications (web servers, web proxies, relational databases, e-commerce systems), graphics, multimedia, and artificial intelligence programs (machine learning, robotics, games). These important application domains are good targets for current and future parallel systems as they operate on increasing datasets and use seemingly parallel algorithms.

We obtained the applications from a variety of sources. Eight of the Java programs are from the JavaGrande benchmark suite [35], while the remaining three are from the DaCapo benchmark suite [6] and the SPECjbb2000 benchmark [36]. The OpenMP applications are from the SPEComp [37] and the NAS [27] benchmark suites. All ANL applications were obtained from the SPLASH and SPLASH-2 benchmark suites [40]. The Pthreads applications come from various sources: the Apache Software Foundation (Apache [3]), SourceForge.net (Kingate and Ultra Tic-Tac-Toe), the University of Chicago (BP-vision [7]), the CMU Carmen project (Localize [41]), and the MPEG Software Simulation Group (Mpeg2 [26]). The variety of sources also implies variability in parallelization quality. While commercial programs such as Apache or benchmark suites are likely to be thoroughly optimized, other programs may not be fully tuned. We believe that measuring transactional behavior in the presence of such variability is actually good as both expert and novice developers will be coding for CMPs in the near future. Hence, we did not attempt further optimizations on any of the applications.

3.2 Transaction Boundaries

Our analysis is based on the premise that the high-level parallelism and synchronization patterns are unlikely to change when applications are ported from lock-based programming models to transactional models. After all, these patterns depend heavily on the algorithm and on the programmer's understanding of the algorithm. Hence, we create transactional versions of the applications by mapping

TM System Case	Abstract Threading Primitive	Transaction Mapping	Transaction Type
Non-blocking Synchronization	Lock	BEGIN	Critical
	Unlock	END	Critical
	Wait	END-BEGIN	Critical
Continuous Transactions	Thread Create/Entry	BEGIN	Non-critical
	Thread Exit/Join	END	Non-critical
	Notify	END-BEGIN	Non-critical
	I/O	END-BEGIN	Non-critical
Speculative Parallelization	Parallel Iteration Start	BEGIN	Critical
	Parallel Iteration End	END	Critical

Table 2: The mapping of multithreading primitives to transaction BEGIN and END markers.

the threading and locking primitives in their code to BEGIN and END (commit) markers for transactions. The mapping is performed merely for the purpose of introducing transaction boundaries in the execution trace generated by running the original multithreaded code. Automatically translating lock-based primitives to transactions (lock elision) and running the new code on a TM system is not always safe [5].

Table 2 summarizes the mapping between multithreading primitives in the application code and transaction boundaries. For brevity, we abstract out the differences between similar primitives in the four programming models. For example, the abstract primitive *Lock* represents the opening bracket on a `synchronized` block in Java, the `LOCK()` macro in ANL, the `pthread_*lock()` calls in Pthreads (regular, read, and write locks), and the opening bracket of the `CRITICAL`/`ATOMIC` pragmas and the `omp_locks()` call in OpenMP. Overall, we identified nine abstract primitives that must be mapped to transactions. Our mapping considers three TM uses: the sporadic use of transactions for non-blocking synchronization, the continuous use of transactions, and the use of transactions for speculative parallelization.

For systems that use TM for non-blocking synchronization, we map *Lock* and *Unlock* primitives to BEGIN and END transaction markers. These markers define nested transactions if locks are nested in the application code. We assume a closed-nesting model, where the stores of an inner transaction are committed to shared memory only when the outer-most transaction commits. We also mark I/O statements within transactions but these markers do not affect the transaction boundaries. The Java conditional `wait` construct (`java.lang.Object.wait()`) is an interesting case. Java requires that a thread obtains a monitor before it calls `wait`, hence `wait` statements are typically made within a `synchronized` block. When `wait` is called, the thread releases the monitor. When the thread resumes after a matching notify, the thread re-acquires the monitor. To reflect this properly, we map `wait` to an END marker (end previous transaction) and a BEGIN marker (start new transaction) pair.

For systems that execute transactions continuously, there is no code execution outside transactions [10, 18]. Again, *Lock*, *Unlock*, and `wait` primitives are mapped to BEGIN and END transaction markers. We call the corresponding transactions *critical* as they must be executed atomically under all circumstances. The difference in this case is that even the code between END and BEGIN markers defined by locks must execute as a transaction. We call these transactions *non-critical* as the TM system could freely split them into multiple transactions by introducing commits to reduce buffer pressure or for other optimizations without loss of atomicity. Thread create/entry primitives define BEGIN markers for non-critical transactions, while thread exit/join primitives define END markers for non-critical transactions. I/O statements outside of critical transactions split the current non-critical transaction into two. In other words, each I/O primitive is mapped to a non-critical transaction END followed by a non-critical transaction BEGIN.

For the use of transactions for speculative parallelization, we cast parallel loops defined through OpenMP pragmas as speculative parallel loops. The beginning and end of an iteration in a parallel loop define the BEGIN and END markers respectively for critical transactions. I/O statements within the iterations are marked but they do not affect transaction boundaries.

The transaction markers that correspond to each primitive in the multithreaded code were inserted in the following way. For Java applications, we modified the just-in-time compiler in the Jikes RVM [1] to automatically insert the markers. The SPLASH-2 applications were also annotated automatically by modifying the ANL macros. For the OpenMP and Pthreads applications, we inserted markers in the source code manually.

3.3 Trace-based Analysis

Once the transaction markers were in place, we executed the applications on a PowerPC G5 workstation and collected a detailed execution trace using `amber`, a tool in Apple's Computer Hardware Understanding Development (CHUD) suite. All tracing runs used 8 threads. The trace includes all instructions executed by the program (memory and non-memory). However, when a multithreading primitive is invoked (e.g. Lock or Unlock), the transactional marker is emitted in the trace instead of the actual code that corresponds to the primitive.

We analyzed the traces using a collection of scripts to extract transactional characteristics such as transaction length, read-set and write-set sizes, read-set and write-set to transaction length ratio, frequency of I/O, and frequency of nesting. The read-set and write-set sizes were measured in three granularities: 4-byte words, 32-byte cache lines, and 4-

Kbyte pages. The read-set (write-set) to transaction length ratio is calculated by dividing the read-set (write-set) size by the number of instructions in each transaction. We perform such calculations on individual transactions before calculating averages. For nested transactions, our scripts use a stack to push and pop transactional contexts according to BEGIN and END markers.

3.4 Discussion

Our methodology has certain limitations that require further discussion. First, it is possible that when the applications are re-written for transactional memory and executed on a TM system, some of the measured characteristics may change. However, the existing multithreaded versions provide good indicators of where parallelism exists and where synchronization is needed in the corresponding algorithms. Hence, we expect that transactional behavior will at least be similar.

Our analysis measures transactional characteristics that are largely implementation independent. For example, we do not attempt to measure the number of buffer overflows in a specific HTM system. Instead, we provide the distribution of read-set and write-set sizes from which one can easily calculate the percentage of transactions that will overflow for a given buffer size.

Our methodology cannot evaluate the frequency of transaction rollbacks for a given application. Such an analysis is difficult to make outside of the scope of a specific TM implementation (number of processors, timing of instruction and communication events, transaction scheduling approach, conflict management techniques, etc.). Nevertheless, our study indicates the potential cost of rollbacks: the amount of work wasted is on the average proportional to transaction length; the overhead of the rollback with an undo-log is proportional to the write-set size; etc.

For practical reasons, we use a single dataset for each application. For the applications evaluated for non-blocking synchronization (Pthreads, Java, ANL), synchronization is used only when a thread accesses potentially shared data. Hence, excluding the percentage of time spent in transactions, the rest of the transactional characteristics will probably be similar with different datasets. For applications evaluated for speculative parallelization, the characteristics will remain unchanged if inner-loops are parallelized, but can vary significantly with the data-set if outer-loops have been chosen for parallelization. Our analysis does not separate between truly shared and fully private addresses in the read-sets and write-sets. An optimizing compiler may be able to classify some addresses as thread private, which in turn allows the TM implementation to avoid tracking them for conflict detection.

Application	Length in Instructions			
	Avg	50th %	95th %	Max
Java avg	5949	149	4256	13519488
sparsematmult	2723	41	34987	53736
series	7756	97	43250	524636
Pthreads avg	879	805	1056	22591
mpeg2	93694	101327	167267	347339
ANL avg	256	114	772	16782
fft	157	157	157	157
radix	9	9	9	9

Table 3: The distribution of critical transaction lengths in instructions.

4. Analysis for Non-blocking Synchronization

This section presents our analysis results for the case of using transactions for non-blocking synchronization with the Java, Pthreads, and ANL applications. For each characteristic analyzed, we first present the application results and then describe a set of basic insights these results provide into building efficient TM systems. We focus mostly on critical transactions but we also comment on non-critical transactions, which are important for systems that execute transactions continuously. Throughout the section, we present tables and figures with averages taken over whole application groups (e.g. the average of all Java programs). We also present the most interesting outliers alongside the averages. Outliers are not included in the group average.

4.1 Transaction Length

Table 3 presents the distribution of critical transaction length in number of instructions. Most transactions tend to be small with 95% of them including less than 5,000 instructions. However, the distribution exhibits a long tail, and a small number of transactions become quite large. ANL applications have the shortest transactions as they are optimized for scalability (infrequent and short atomic regions). The same behavior is observed with Pthreads programs, particularly Apache which is a well-tuned commercial program. Mpeg2 is the only exception as it uses a lock during the whole operation on a video frame slice. Most Java programs exhibit short transactions on the average but have a long distribution tail. The longer transactions are partly due to the applications themselves and partly due to the long critical regions used in Jikes RVM for scheduling, synchronization, class loading, and memory management.

We do not present the length of non-critical transactions due to space limitations. They are typically longer, but they can be split at arbitrary places to avoid any issues associated with their length.

Observations: The high frequency of very short transactions (150 instructions or less) implies that the overheads associated with TM mechanisms such as starting or committing a transaction must be very low. Otherwise, the overheads may cancel out any benefits from non-blocking synchronization. This can be a significant challenge for STM systems which use runtime functions at transaction boundaries. For HTM systems, fast register checkpointing will be necessary to keep the transaction start overhead low. Alternatively, HTM systems will require compiler help to avoid checkpointing all registers on transaction starts.

The frequency of short transactions has implications on time virtualization for HTM systems. The UTM [2] and VTM [30] systems propose that I/O or timer interrups cause a transaction to be swapped out by saving its transactional state (read-/write- set) in special overflow buffers in virtual memory. The state is restored when the transaction resumes later. Given the overhead of saving/restoring state and that 95% of transactions include less than 5,000 instructions, an alternative approach makes sense. On an interrupt, we should initially wait for one of the processors to finish its current transaction and assign it to interrupt processing. Since most transactions are short (150 instructions), this will likely happen quickly. If the interrupt is a real-time one or becomes critical, we should rollback the youngest transaction and use its processor for interrupt handling. Re-executing some portion of a short transaction is faster than saving and restoring state in virtual memory buffers. Saving and restoring should be reserved only for the very long transactions that span multiple OS quanta (tens of millions of instructions). Our data suggest that this case is extremely rare hence it should be handled in software, using OS-based techniques [33, 20], without the complicated hardware structures discussed in [2] and [30].

4.2 Read-Set and Write-Set Sizes

Figure 1 presents the distribution of read-set and write-set sizes for critical transactions. For 95% of transactions, the read-set is less than 4 Kbytes and the write-set is less than 1 Kbyte. However, we must also consider that large read-/write- sets are typically found in longer transactions. Figure 2 presents the normalized time spent on transactions with read-/write- sets of a specific size. We assume that time is proportional to the transaction length. For read-sets, a 52-Kbyte buffer is needed to cover 80% or more of the transaction execution time for most applications. For write-sets, a 30-Kbyte buffer is sufficient for 80% of the transaction execution time of most applications with the exception of hsqldb. Hsqldb includes some long JDBC connections that execute large SQL queries. We should also note from Figures 1 and 2 that read-set and write-set sizes do not exceed 128 Kbytes. Our analysis also indicates that there is

	Read-Set		Write-Set	
Application	Words/Line	Lines/Page	Words/Line	Lines/Page
Java avg	1.7	3.7	1.7	4.3
Pthreads avg	2.0	3.8	3.5	8.5
ANL avg	2.1	6.8	2.3	7.9
sparsematmul	1.3	1.7	2.1	3.0
mpeg2	7.8	71.2	2.3	7.9

Table 4: The number of words used per line and lines used per page for the transaction read-sets and write-sets.

significant overlap between read- and write-sets. The ratio of their intersection divided by their union is typically 15% to 30%. If both sets are tracked in a unified buffer such as a data cache, large overlaps reduce the pressure on buffer capacity.

Transaction read- and write-sets can be tracked at the granularity of words, cache lines, or memory pages. Coarser granularity tracking reduces overheads but may lead to unnecessary work due to loss of accuracy. Table 4 shows the number of words used per line and the number of lines used per page when the read-/write- sets are tracked at line and page granularity respectively. Words are 4 bytes, lines are 32 bytes, and pages are 4 Kbytes. For most applications, only 2 out of 8 words per cache line in the read-set are actually used. For write-sets, the ratio is at 3 words used per cache line. For the case of page granularity, less than 10 cache lines from the 128 per page are actually used in the read-/write- sets. Mpeg2 is the only exception in both cases as it exhibits very good spatial locality in its read-set and write-set accesses.

Observations: Figure 2 shows that transaction read-sets and write-sets are often too large to fit in small side-buffers as in the original HTM design [14]. However, they are within the capacity of processor caches, which can be modified to track transaction's read-sets and write-sets. The majority of transactions fit in an L1 cache (16 to 32 Kbytes) but support at the L2 cache (128 Kbytes to Mbytes) is needed to avoid overflows for the few long transactions. With read-/write- set tracking in L2 caches, overflows will be extremely rare events. Hence, complex hardware mechanisms that allow read-/write- sets to overflow in virtual memory [2, 30] will hardly ever get used. Instead, it is preferable to have a simple virtualization mechanism that is completely software based. In the rare event of a read-/write- set that exceeds the L2 capacity, an exception is raised and the system executes the transaction using OS-based transactional mechanisms that operate on top of the virtual memory system without any hardware limitations [33, 20].

Tracking read-/write- sets at cache line granularity [29, 2, 25] leads to significant accuracy loss and may lead to performance inefficiencies: unnecessary rollbacks due to false

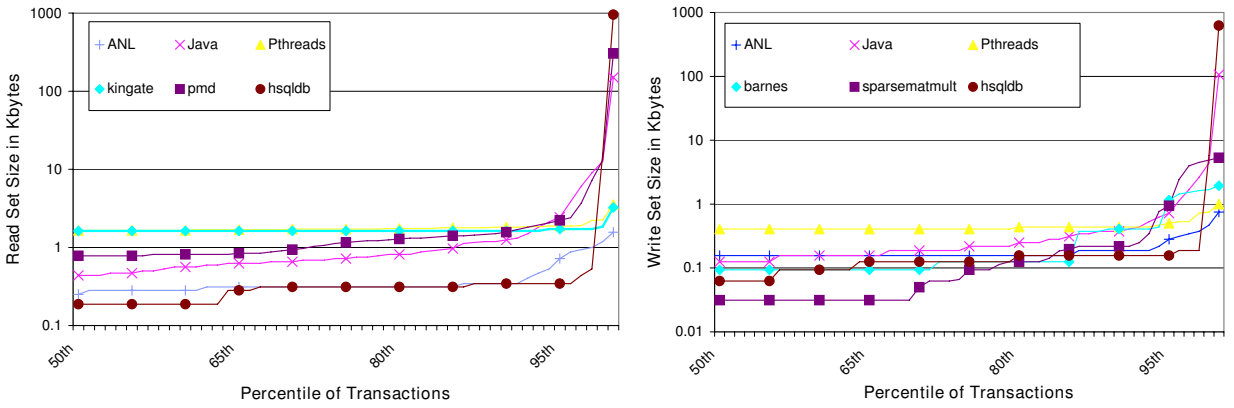

Figure 1: The cumulative distribution of read-set (left) and write-set (right) sizes in Kbytes.

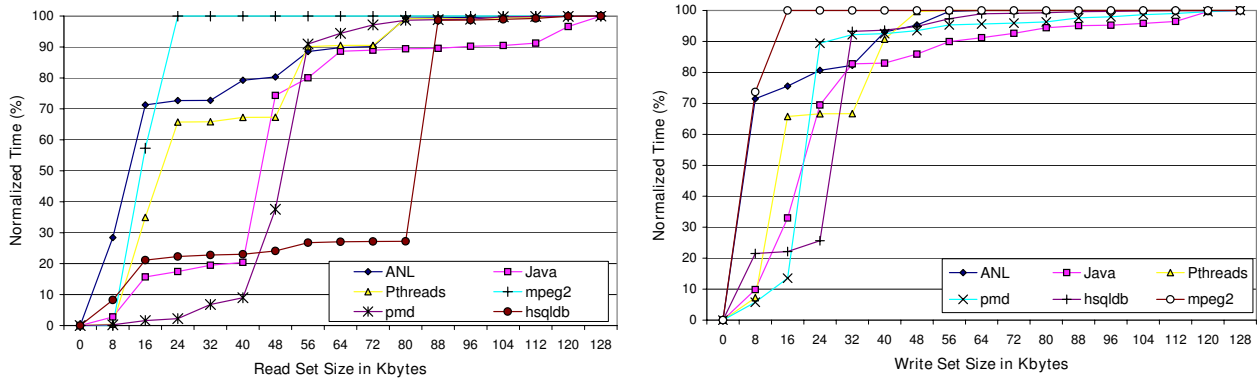

Figure 2: Normalized time spent in transactions with different read-set (left) and write-set (right) sizes.

sharing as well as logging (buffer space waste) or committing (badwidth waste) more than two times the amount of necessary data. Word granularity tracking can eliminate these issues [23]. Page granularity tracking is simply too wasteful to allow for good performance for most of applications.

4.3 Read-/Write- Set Size to Transaction Length Ratio

Hardware and software TM systems perform basic operations for each word in the read- or write- set: log an old value in the undo-log, commit the new value to shared memory, validate a read value before commit, restore old value at abort, etc. The overheads of such operations can be hidden if they can be amortized across a large number of instructions in the transaction. Figure 3 presents the distribution of write-set size (in words) to the transaction length (in instructions) ratio for critical transactions. A ratio of 50% means that the transaction stores to a unique new address every two instructions. The ratio is roughly 10% for most transactions but some of them go as high as 25%. The reason for the high ratios is that these applications perform all updates to shared data in critical sections. For non-critical

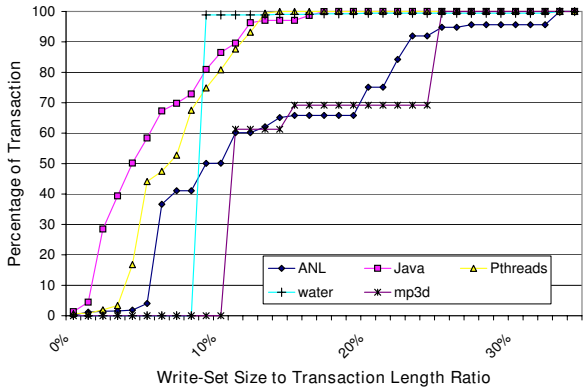

Figure 3: The write-set size (in words) to transaction length (in instructions) ratio for critical transactions.

transactions, the write-set size to transaction length ratio is typically below 10%. The reason is the lower frequency of stores altogether and higher temporal locality for stores. For the read-set size to transaction length ratio, we observed similar statistics. Many critical transactions exhibit ratios of 15% up to 30%. Non-critical transactions have substantially

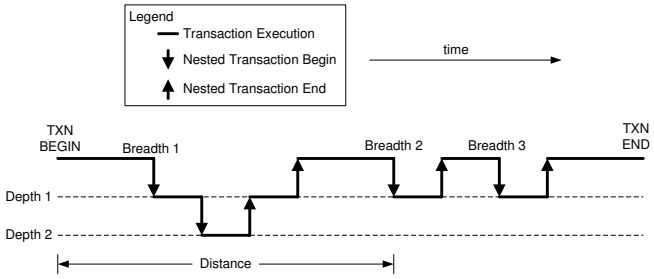

Figure 4: The definition of depth, breadth, and distance for nested transactions.

Application	% Trans. with IO	% Trans. with Wr IO	% Trans. with Rd IO	% Trans. with Rd & Wr IO
water-spatial	5.3	0.0	5.3	0.0
moldyn	0.4	0.4	0.0	0.0
raytracer	0.4	0.4	0.0	0.0
crypt	0.4	0.4	0.0	0.0
series	0.3	0.3	0.0	0.0
sor	0.4	0.4	0.0	0.0
pmd	0.5	0.5	0.0	0.0
kingate	1.3	0.0	1.3	0.0
mpeg2	20.0	20.0	0.0	0.0

Table 6: I/O frequency in critical transactions.

lower ratios.

Observations: The high read-/write- set to instruction length ratios signal potential problems for various TM implementations. For HTM systems using a store buffer, the latency of writing all stores to shared memory at commit time may be difficult to hide. Hence, double-buffering techniques may be necessary to avoid the slowdown [10]. For HTM systems using an undo-log, log updates can be frequent and may require a separate port into the cache hierarchy to avoid stalls. For STM systems, the overhead from lock acquisition for stores and locks or version validation for loads may be difficult to hide if read- and write- sets are tracked at fine granularity.

On the other hand, the above results suggest that systems that support continuous transactional execution do not suffer any additional inefficiencies because of non-critical transactions. The latency of mechanisms for logging, committing, validating, or rolling back transactional data is a bigger issue with critical rather than non-critical transactions.

4.4 Transaction Nesting

Nested transactions may occur in transactional programs when they call library code that uses transactions internally. Nested transactions also allow programmers to avoid expensive rollbacks when potential conflicts are limited within a small portion of a large transaction. A major question for HTM implementations is how much nesting support to provide through fast hardware. To explore this issue, we define four characteristics of nested transactions in Figure 4. Depth refers to the level of nesting at which a transaction executes (number of parents in the nested call graph). Breadth refers to the number of nested calls each transaction makes (number of immediate children in the nested call graph). Distance is the number of instructions between the beginning of a transaction and the beginning of one of its children.

Table 5 presents the nesting characteristics for applications that exhibit nesting in more than 1% of their transactions. It is immediately obvious that nesting is not widespread. Most programmers avoid nested synchronization either because it is difficult to code correctly with current models or because of the obscure performance implications. Most programs in Table 5 are Java applications, where nesting synchronization occurs in the Jikes RVM code to support class loading (tree-like class loader) and just-in-time compilation. Nesting depths of 1 and 2 are the most frequent and the average breadth is 2.2.

Observations: It is difficult to draw general conclusions on nesting from our analysis. For the specific applications, one may be able to eliminate the need for nesting support by recoding the Java virtual machine. For HTM systems that automatically flatten nested transactions, the penalty for a conflict in the inner transaction that leads to an outermost transaction rollback will be proportional to the transaction distance. For the Java programs we studied, the mean distance is quite high (140,000 instructions on the average). For implementations that provide full support for nested transactions, it seems that two levels of nesting support will be sufficient. Such nesting support includes the ability to track read-/write- sets and detect conflicts independently for three transactions. The same hardware resources can be used for double-buffering to hide commit overheads [23]

4.5 Transactions and I/O

I/O operations within a critical transaction can be problematic. Input data from external devices must be buffered in case the transaction rolls back. Output data must also be buffered until the transaction commits. If a single transaction performs both input and output operations, deadlocks can occur through the I/O system. Table 6 shows the percentage of critical transactions that include I/O operations. Most applications have few critical transactions with I/O, which is natural because a long I/O operation is unattractive, and usually unnecessary, within a application-level critical section[1]. Mpeg2 and Water_spatial are the exceptions. The Mpeg2 algorithm holds a lock while reading a slice from the

[1]We do not consider any critical sections in the operating system code necessary to implement I/O operations.

Application	% Trans with Nesting	Nesting Depth (% of Nested Trans)			Nesting Breadth (% of Total Trans)			Mean Distance
		1	2	>2	1	2	>2	
moldyn	22	16	42	41	13	7	3	291889
montecarlo	14	99	0	0	0	0	14	2784
raytracer	14	36	41	23	7	4	3	99262
crypt	18	45	37	19	11	3	4	56211
lufact	18	39	38	23	11	3	5	87913
series	14	40	51	8	7	2	4	68782
sor	16	48	4	48	9	3	4	75400
sparsematmult	13	87	11	2	6	1	6	10440
specjbb	9	63	35	2	1	4	4	58855
pmd	17	19	30	52	8	4	5	659871
hsqldb	1	3	97	0	0	0	1	6538826
bp-vision	4	100	0	0	4	0	0	165
localize	2	100	0	0	2	0	0	641

Table 5: The nesting characteristics of critical transactions.

video stream. Water_spatial's output operations use a lock to print to the console. No transactions attempt to execute both an input and an output operation.

With non-critical transactions, I/O handling is easy as we can split transactions in any way necessary (immediately before and after any I/O statement).

Observations: I/O is unlikely to be a serious roadblock to transactional memory. I/O is rare within critical transactions and the deadlock scenario that cannot be handled by buffering I/O does not occur in practice.

Application	Length in Instructions			
	Avg	50th %	95th %	Max
equake	244	9	1134	40750634
art	70062948	71978851	74117449	74824088
is	129	3	3	19844217
swim	62130	68467	91296	91296
cg	521	6	691	18949151
bt	40796	8106	35531	13091051

Table 7: Transaction length statistics for speculative parallelization.

5. Analysis for Speculative Parallelization

Apart from facilitating non-blocking synchronization, TM allows for speculative parallelization of sequential programs [9]. Speculative parallelization provides an attractive programming model as it eliminates the burden of developing a provably correct parallel program. Instead, the programmer merely identifies potentially parallel regions in sequential code. Hardware executes these regions optimistically in parallel and resolves dynamically any dependencies based on the sequential semantics of the original code. Nevertheless, using TM for speculative parallelism may lead to significantly different common case behavior than that presented in Section 4. To identify the major trends, we study the six OpenMP applications in Table 1 after remapping the iterations from parallel loops in OpenMP into critical transactions for speculative parallelization. In other words, we assume an OpenMP-like, directive-based model for speculative loop parallelization.

Table 7 presents the transaction length statistics. The applications fall into two categories: those with speculative parallelism in inner loops (Equake, Is, Cg) that leads to short transactions and those with speculative parallelism from outer loops (Art, Swim, Bt) that leads to significantly longer transactions. For Art in particular, transactions include tens of millions of instructions. Figure 5 presents the transaction read-set and write-set distributions. Applications with long transactions generate significantly larger read-sets and write-sets than those in Section 4 (200 Kbytes to 2 Mbytes). On the other hand, all speculatively parallelized applications exhibit low read-/write- set to transaction length ratios (10% or less) due to high temporal locality in large transactions. None of these applications include noticeable nesting or I/O operations within critical transactions.

Observations: The main conclusion is that using TM for speculative parallelization requires larger buffers to track read-sets and write-sets. L1-sized buffers can be too small for several applications and may lead to frequent overflows. If L1-sized buffers are used, fast TM virtualization techniques will be critical. On the other hand, L2-sized buffers

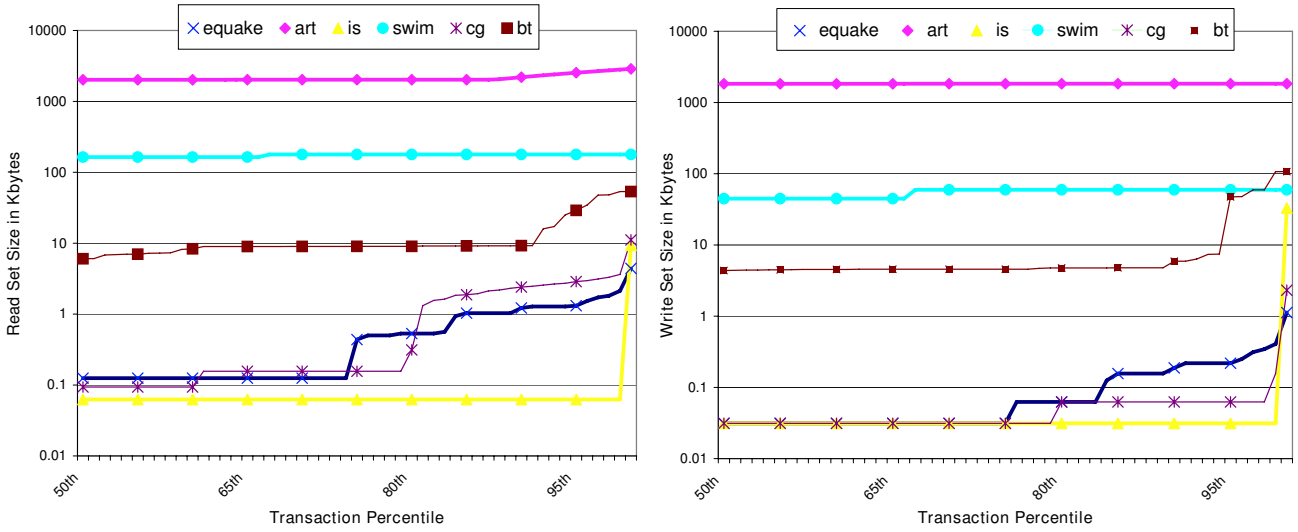

Figure 5: The cumulative distribution of read-set (left) and write-set (right) sizes for speculative parallelization.

are sufficient to eliminate overflows for most applications. Alternatively, one can conclude that speculative parallelization with TM suggests parallelizing the inner loops instead of the outer loops, as in the conventional wisdom with current systems. Since the read-/write- set to transaction length ratios are lower than with non-blocking synchronization, the use of TM for speculative parallelization does not place any additional requirements on the mechanisms for buffering, commit, and rollback.

6. Conclusions and Future Work

We studied a set of existing multithreaded applications in order to characterize their common case behavior with transactional memory systems. The analysis involves mapping parallelism and synchronization primitives in the source code to transaction boundaries. We measure basic characteristics such as the distribution of transaction lengths, read-set and write-set sizes, and the frequency of nested transactions and I/O operations. These characteristics provide key insights into the design of efficient TM systems for both non-blocking synchronization and speculative parallelization.

Our analysis indicates the following trends. Most transactions are small, hence the fixed overheads associated with starting and ending transactions must be minimized. Interrupts and contexts switches can be handled with simple software mechanisms. The read-sets and write-sets for most transactions fit in L1 caches. Long transactions, particularly from speculative parallelization, require read-/write-set buffering in the L2 cache as well. Since L2 overflows will be rare, complex hardware mechanisms for TM virtualization are unnecessary and should be replaced by software-only alternatives. Continuous transaction execution and speculative parallelization with transactions do not require lower overheads per address read or written than what is needed for non-blocking synchronization. Nested transactions occur mostly in system code and limited hardware support is likely to be sufficient. I/O operations within transactions are also rare. The observed I/O patterns are easy to handle through I/O buffering techniques.

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Additional support was also available through NSF grant 0444470.

References

[1] B. Alpern and S. Augart. The Jikes Research Virtual Machine Project: Buliding an Open-source Research Community. *IBM Systems Journal*, Nov. 2005.

[2] C. Ananian, K. Asanović, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded Transactional Memory. In *the 11th Intl. Symposium on High-Performance Computer Architecture*, Feb. 2005.

[3] The Apache HTTP Server Project. http://httpd.apache.org/.

[4] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 3rd Edition*. Addison-Wesley, 2002.

[5] C. Blundell, E. Lewis, and M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *The 4th Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[6] The DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.

[7] P. Felzenszwalb and D. Huttenlocher. Effi cient Belief Propagation for Early Vision. In *the IEEE Conference on Computer Vision and Pattern Recognition*, June 2004.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, , and K. Olukotun. Programming with transactional coherence and consistency. In *the 11th Intl. Conference on Arch. Support for Programming Languages and Operating Systems*, Oct. 2004.

[10] 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 *the 31st International Symposium on Computer Architecture*, June 2004.

[11] T. Harris. Exceptions and Side-effects in Atomic Blocks. In *the Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[12] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *the 22nd Symposium on Principles of Distributed Computing*, July 2003.

[14] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the 20th Intl. Symposium on Computer Architecture*, May 1993.

[15] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous Multithreading Implementation in POWER5. In *the 15th Hot Chips 15 Symposium*, August 2003.

[16] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE MICRO*, 25(2), Mar. 2005.

[17] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[18] B. Kuzmaul and C. Leiserson. Transactions Everywhere. MIT LCS Research Abstract, 2003.

[19] B. Lewis and D. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1998.

[20] D. Lowell and P. Chen. Free Transactions with Rio Vista. In *the 16th symposium on Operating Systems Principles*, Oct. 1997.

[21] E. Lusk and R.Overbeek. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[22] V. Marathe, W. Scherer, and M. Scott. Adaptive Software Transactional Memory. In *the 19th Intl. Symposium on Distributed Computing*, Sept. 2005.

[23] A. McDonald, J. Chung, H. Chafi , C. Minh, B. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *the 14th Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2005.

[24] C. McNairy. Montecito: The Next Product in the Itanium Processor Family. In *the 16th Hot Chips Symposium*, Aug. 2004.

[25] K. Moore, J. Bobba, M. Morovan, M. Hill, and D. Wood. LogTM: Log Based Transactional Memory. In *the 12th Intl. Conference on High Performance Computer Architecture*, Feb. 2006.

[26] MPEG Software Simulation Group. http://www.mpeg.org/MSSG/.

[27] NASA Advanced Supercomputing Parallel Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[28] OpenMP Application Program Interface, Version 2.5. http://www.openmp.org/, May 2005.

[29] R. Rajwar and J. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *the 10th Intl. Conference on Arch. Support for Programming Languages and Operating Systems*, October 2002.

[30] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *the 32nd Intl. Symposium on Computer Architecture*, June 2005.

[31] M. Ringenburg and D. Grossman. AtomCaml: First-Class Atomicity via Rollback. In *the 10th Intl. Conference on Functional Programming*, Sept. 2005.

[32] B. Saha, A. Adl-Tabatabai, R. Hudson, C. Cao Minh, and B. Hertzberg. A High Performance Software Transactional Memory System For A Multi-Core Runtime. Technical report, Intel Inc, 2005.

[33] M. Satyanarayanan et al. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1), Feb. 1994.

[34] N. Shavit and S. Touitou. Software Transactional Memory. In *the 14th Symposium on Principles of Distributed Computing*, Aug. 1995.

[35] L. Smith, J. Bull, and J. Obdrzalek. A Parallel Java Grande Benchmark Suite. In *the Supercomputing Conference*, Nov. 2001.

[36] The SPEC jbb2000 Benchmark. http://www.spec.org/jbb2000/.

[37] SPEC OpenMP Benchmark Suite. http://www.spec.org/omp/.

[38] H. Sutter. The Concurrency Revolution. *C/C++ Users Journal*, 23(2), Feb. 2005.

[39] H. Sutter. The Trouble with Locks. *C/C++ Users Journal*, 23(3), Mar. 2005.

[40] S.Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *the 22nd Intl. Symposium on Computer Architecture*, June 1995.

[41] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust Monte Carlo Localization for Mobile Robots. *Artifi cial Intelligence*, 128(1-2):99–141, 2001.