

# Concurrent GC Leveraging Transactional Memory

Phil McGachey<sup>1</sup>   Ali-Reza Adl-Tabatabai<sup>2</sup>   Richard L. Hudson<sup>2</sup>   Vijay Menon<sup>2</sup>   Bratin Saha<sup>2</sup>  
Tatiana Shpeisman<sup>2</sup>

<sup>1</sup>Department of Computer Science  
Purdue University  
West Lafayette, IN 47907

<sup>2</sup>Programming Systems Lab  
Intel Corporation  
Santa Clara, CA 95054

phil@cs.purdue.edu

{ali-reza.adl-tabatabai,rick.hudson,vijay.s.menon,bratin.saha,tatiana.shpeisman}@intel.com

## Abstract

We predict that the ever-growing number of cores on our desktops will require a re-examination of concurrent programming. Two technologies are likely to become mainstream in response: Transactional memory provides a superior programming model to traditional lock-based concurrency, while Concurrent GC can take advantage of multiple cores to eliminate perceptible pauses in desktop applications such as games or Internet telephony. This paper proposes a combination of the two technologies, producing a synergy that improves scalability while eliminating the annoyance of user-perceivable pauses.

Specifically, we show how concurrent GC can share some of the mechanisms required for transactional memory. Thus as transactional memory becomes more efficient, so too will concurrent GC. We demonstrate how, using a state of the art software transactional memory system, we can build a state of the art concurrent collector. Our goal was to reduce 90% of pause times to under one millisecond. Of the remainder, we aim for 90% to be under 10ms, and 90% of those left to be under 100ms. Our performance results show that we were able to achieve these targets, with pause times between one or two orders of magnitude lower than mainstream technologies.

**Categories and Subject Descriptors** D.3.4 [Programming Languages]: Processors—Memory management (garbage collection)

**General Terms** Algorithms, Performance, Design, Experimentation, Languages

**Keywords** Transactional Memory, Strong Atomicity, Concurrent Garbage Collection, Compiler Optimizations, Virtual Machines, Transactional Integrity

## 1. Introduction

The computing industry has long been aware of Moore's law. Historically, transistor densities have doubled at roughly eighteen-month intervals; a trend allowing processor speeds to grow at a phenomenal rate. However, additional transistors are beginning to provide diminishing returns on the speed of monolithic processors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.  
Copyright © 2008 ACM 978-1-59593-960-9/08/0002...\$5.00

Additionally, power consumption is becoming a real concern, with heat dissipation growing with chip complexity. It is clear that future processor designs will take advantage of Moore's law by adding cores, rather than by ramping up clock speeds.

This is already occurring. Multiple cores are ubiquitous in the server market, and virtually all high volume desktops shipping today contain at least two cores. If historical trends continue, we can expect the number of cores in an average desktop machine to double every few years. It is not hard to imagine a future desktop machine with hundreds or thousands of cores, and hardware threads outnumbering software threads. Unfortunately, exploiting these processor improvements will not be as simple as in the past. Performance gains will come from application parallelism, rather than CPU horsepower. This will lead to an evolution in programming, as programmers develop means of introducing concurrency.

In this paper we discuss two major aspects of modern application development: concurrency control and garbage collection. Both will be affected by the rise of concurrent programming. However, we propose a synergistic combination of the two that will allow simple, high performance applications to be developed on future architectures.

### 1.1 Concurrency

As the number of available CPU cores increases, application developers will use an increased degree of concurrency to exploit the hardware's potential. However, as the number of threads increases, so too do the complexities of their interactions. We believe that current lock-based concurrency control mechanisms will prove to be inadequate when reasoning about large applications.

Transactional memory (TM) has been proposed as a simplified model for shared-state threaded programming. Developers can reason about their concurrent code as though it were running in isolation of all other threads, with its effects performed atomically. TM eliminates many of the common errors introduced by improper locking, such as deadlock or data races. Strongly atomic transactional memory[5] goes further, by making all memory access transactional it eliminates many of the semantic problems observed in weakly atomic systems.

We believe transactional memory to be a vital part of large-scale concurrent application development. We assume that it will eventually be commonplace, with hardware support under future architectures. And we assert that strong atomicity provides a superior programming model to weak atomicity.

### 1.2 Garbage Collection

A second vital area of modern application development is garbage collection. Until now, traditional stop-the-world techniques have

been effective in providing garbage collection to desktop applications. Such algorithms are simple to design and build, and can be optimized to minimize disruption. However, as the number of cores in a system grow, pausing all threads in order to garbage collect is no longer feasible. Pausing hundreds or thousands of threads has an overhead of its own, which must be paid before any collection work can begin. Parts of the garbage collector cannot be parallelized, leading to a small number of threads working while hundreds sit idle. And trends in memory sizes suggest that application heaps are likely to grow, leading to longer pauses while the garbage collector scans for references.

There exists an opportunity to exploit the large number of cores on our target platforms for the benefit of the garbage collector. We propose a concurrent GC algorithm that requires that application threads be paused individually and only briefly, and performs the majority of the work concurrently with the application.

### 1.3 Synergy

At face value, garbage collection and transactional memory appear to be disparate concepts. However, on closer inspection we can see that they share underlying mechanisms. A concurrent garbage collector must be aware of the interactions between application threads and objects undergoing collection. Similarly, a transactional memory system must track interactions between application threads and objects to detect violations of transactional integrity, when multiple threads access a single object. We can leverage this overlap to reduce the costs of concurrent garbage collection in a system with strongly atomic transactional memory.

### 1.4 Targets

In this work, we target future desktop architectures. Such processors will have tens, hundreds, or thousands of cores, and may have support for transactional memory built into the hardware. We expect these future machines to have strongly atomic transactional memory mechanisms embedded throughout the software stack, from the operating system to the applications.

We target desktop applications. While the garbage collection algorithm we present runs concurrently with user threads and offers low pause times, we do not make hard real-time guarantees. Our goal in this work is to provide soft real-time, where 90% of application pauses take less than a millisecond. Of the remainder, we aim for 90% to take less than ten milliseconds and 90% of those left to take less than 100 milliseconds. While this is not sufficient for hard real-time applications such as sensor monitoring or military uses, it is more than adequate for standard desktop programs such as games, telephony, or multimedia players.

### 1.5 Contributions

The major contribution we present in this paper is the synergistic combination of two rapidly evolving technologies. Additionally, we combine classic tri-color garbage collection marking algorithms, low pause time algorithms and termination guarantees with a novel transactional copying and read barrier algorithm.

We have implemented our system using a high-performance Software Transactional Memory (STM) infrastructure. We discuss some of the optimizations performed by the JIT compiler, and present the results of an extensive performance study.

## 2. Transactions As Concurrency Control

As programmers turn their attention to exploiting multi-cored architectures, the issue of concurrency control will come to the forefront. Designing robust, correct and efficient concurrent applications is a notoriously difficult problem. Critical sections of code and data structures must be guarded to prevent race conditions, while

over-synchronization can stifle performance; misplaced locks can force the program to execute serially, losing all advantages of the architecture. Lock acquisition must be carefully controlled to avoid deadlock. And all the locking policies and behaviors must be codified in such a way that the application can be safely modified years later, possibly by completely different teams of developers.

We believe that the concurrency control mechanisms now available to programmers will prove insufficient for the widespread development of highly concurrent applications. As an alternative, transactional memory offers a simplified mechanism through which concurrency can be expressed. Programmers need simply declare a section of code to be atomic, and the runtime system ensures that the operations performed within occur in isolation from all other transactions.

### 2.1 Transactional Integrity

We define *transactional integrity* to mean that transactions are atomic, consistent, and run in isolation. With weak atomicity, the programmer must explicitly mark potentially racy memory accesses as transactional to maintain transactional integrity. The alternative is strong atomicity, where it is the responsibility of the system to maintain transactional integrity perhaps by considering all memory accesses to take place transactionally. It is easy to compare this with referential integrity; in a malloc/free system it is the responsibility of the programmer to maintain referential integrity while in a garbage collected system it is the responsibility of the runtime.

## 3. The Concurrent Garbage Collector

Our concurrent garbage collection algorithm is based upon the Sapphire collector [17]. However, we are able to improve upon this algorithm through use of our transactional memory infrastructure.

### 3.1 Memory Organization

We view the application's memory space as containing three regions:

- **Uncollected.** A region of memory that will not be collected in a given garbage collection cycle. This space is also used to allocate metadata required by the VM. Allocation during concurrent garbage collection occurs in this region.
- **Collected.** The portion of the heap that is subject to collection in a given garbage collection. All data in the collected area is guaranteed to take the form of objects, including all application objects. The collected area is further broken down into fixed-sized blocks, which can be collected individually.
- **Stacks.** Each thread has a private stack, which contains no objects. Thread local structures such as read and write sets are also in this region, though they may not be on the actual runtime stack.

### 3.2 The Garbage Collector Phases

The garbage collection algorithm is divided into four major phases, each of which runs concurrently with the application.

#### 3.2.1 The Mark Scan Phase

The mark scan phase implements a concurrent marking algorithm. We describe the algorithm in terms of the traditional tri-color abstraction [11]; *white* objects have yet to be seen by the marker, *gray* objects have been seen but not yet scanned, and *black* objects have been fully scanned. We maintain the invariant that a black object may not point to a white object; the act of blackening an object ensures that all referenced objects are at least gray.

Before the mark phase commences, all objects in the Collected region are white. Prior to any marking, we install the mark phase write barrier. This barrier ensures that the mark phase invariant is maintained; should an application thread attempt to install a pointer to a white object into a field of a black object, the white object is first set to gray. An additional effect of this barrier is that newly allocated objects are considered to be black. Any pointers written to a new object (including initialization data) are caught by the write barrier, and referenced objects are marked gray.

During the mark phase, each thread is paused individually while its Stack region is scanned. All objects in the Collected region that are reachable from a given Stack region are colored gray, and the thread is restarted. At this point we consider the slots in the newly scanned Stack to be black. By blackening all gray objects, we perform a full traversal of the live objects in the Collected region.

The final step is to ensure that no white objects have been stored in the Stack region during copying. Since there is no read barrier in place during this phase, it is possible that a concurrently running application thread could have loaded a reference to a white object which could then be missed by the marking. We check for this by pausing each thread individually, and scanning its Stack region, graying any white objects we find. We iterate over this process until no white objects exist on any thread's Stack. Section 3.3 demonstrates the correctness of this method.

### 3.2.2 The Copy Phase

The copy phase evacuates live objects from selected blocks in the Collected region, allowing these blocks to be reclaimed. Not all blocks are selected for copying at every garbage collection; in the worst case all objects in the heap may be reachable, so it is necessary that space be allocated to store all objects that may be copied. To minimize the space overhead of this copy reserve, we copy objects from only a small part of the heap at once. We designate the set of blocks to be copied as the Collected region. Objects are copied to the copy reserve, which resides in the Uncollected region.

When an object has been copied, a forwarding pointer is installed in its header (see Section 4.2.1). Read and write barriers are required to ensure that the concurrently-running application threads observe only the most recent version of the object, whether in the Collected or the Uncollected region. The read barrier simply checks for the existence of a forwarding pointer, following it if necessary. The write barrier operates on both pointers and scalars. In either case it follows forwarding pointers prior to writing, but on pointer writes it also updates the reference being written if it points to a forwarded object.

The copy operation itself poses some difficulty in a concurrent collector. There exists the possibility of a race between the collector and an application thread, leading to a lost update. Our solution to this problem is discussed in Section 4.2.2.

### 3.2.3 The Flip Phase

The third stage of a collection is the flip phase, where references to Collected objects are systematically replaced with their newly-copied equivalents. Since the application threads may still see references that have not yet been flipped, it is necessary for the read and write barriers from the copy phase to remain in place.

The flip phase follows a similar protocol to the mark phase; first the Stack regions of each application thread is flipped, then all references in the Collected and Uncollected regions are flipped, and finally the Stack regions are flipped again to ensure that any remaining unflipped references are caught.

Once the flip phase terminates, the barriers are no longer required. All references in the system point to valid objects and, after some maintenance tasks, the garbage collection cycle is complete.

## 3.3 Ensuring Mark and Flip Phase Termination

We must ensure that all live objects are reached during the tracing phases of the garbage collector. This may be difficult in the face of concurrent application threads; standard tracing techniques do not account for references being copied and overwritten during the scan. Our collector traces the heap in both the mark and the flip phases. Since the mechanism used to ensure that all references are seen is equivalent in both phases, we describe here the mark phase.

The correctness issue arises because there are no read barriers on Stack accesses. It would be possible for a thread to read a reference to a white object after its stack was scanned. If this was the only reference to a live object, it would be missed by the mark phase and so be a candidate for collection. We avoid this problem by scanning each thread's Stack region several times, until we can guarantee that all live objects have been marked.

Each application thread is paused individually, and its Stack region is scanned. If no unmarked references are found during an iteration of a complete iteration of the threads, we conclude that there are no reachable white objects, and the phase ends. If there are any references to white objects, we restart the thread and scan those objects. We repeat this process until no unmarked references are found during a complete iteration of all the threads. For a proof that this termination algorithm is correct and complete see [17].

## 3.4 Real Time

As research into real-time systems gains importance, it is important to clearly state the level of real-time guarantees a concurrent garbage collector is prepared to make. The concurrent garbage collector described here implements soft real-time; we aim to ensure that 90% of pauses take less than a millisecond. However, we do not guarantee absolute predictability, making the collector unsuitable for hard real time applications.

We also do not make guarantees about the timing of the pauses. Indeed, pauses in our system occur at irregular intervals, since application threads are paused only when the garbage collector is running. For much of the application's run time, it will experience no garbage collection pauses at all. Additionally, our garbage collector causes some variability in application thread performance due to the barriers executed at various points in the collection.

# 4. Implementation

## 4.1 Software Transactional Memory

This work is built upon a high-performance software transactional memory infrastructure for Java. Our base system [1, 24] extends Java with an `atomic` language construct for declaring a code block as transactional. To support this, the modified just-in-time (JIT) compiler specially handles transactional control flow, inserts barriers for transactional and non-transactional memory accesses, and optimizes these operations to reduce overheads. Additionally, the modified virtual machine adds a transaction record field to the header of every object in the heap and creates separate transactional and non-transactional versions for each method. The base system uses the McRT-STM [22] runtime underneath, which implements optimistic concurrency control using versioning [19] for reads and strict two-phase locking [10] and eager versioning for writes.

### 4.1.1 Data Structures

An object in our base transactional memory system can be in one of two states; a shared object can be read by any transaction, while an exclusive object is available only to a single transaction. The status of an object is determined by the transaction record field in its header. A shared object's transaction record contains a version number (ending with a 1). An exclusive object's field contains a reference to the transaction descriptor data structure of

the owning transaction. In this way, whether the object is shared can be determined by inspecting a single bit. All objects are shared by default.

An object becomes exclusive as the result of a transactional write operation, and remains in that state until the transaction completes. When a write instruction appears in a transaction, it first checks the status of the target object. If the object is in the exclusive state and the current transaction is the owner, no further action is necessary. If another transaction is the owner, a conflict has occurred and the transaction must fall back to the contention manager. If the object is in the shared state, the transaction atomically inserts a reference to its transaction descriptor. It then logs the object's address and current version number in its write log.

The object remains in its exclusive state until the owner transaction completes, either through committing or aborting. If the transaction commits, any modifications to the object become permanent, and the object's version number is increased. Otherwise, all modifications are undone, and the version number remains the same. The object is released back to a shared state by replacing the transaction record pointer with the appropriate version number.

This version number protocol allows the TM system to determine whether an object has been modified in a given interval. When a shared object is first read by a transaction, its address and version number is stored in that transaction's read log. When the transaction attempts to commit, the read log is scanned and all version numbers are checked against the logged values. If an object has been modified since it was read or if it is in exclusive state, the version numbers do not match, and so the transaction cannot commit.

#### 4.1.2 Strong Atomicity

The version number protocol provides a means of ensuring that two transactions do not conflict on a given object. However, it makes no guarantees regarding non-transactional threads. If a thread does not observe the correct protocol when performing writes, it would be possible for it to modify an exclusive-mode object, causing the transactional code to read an incorrect value. Similarly, a non-transactional thread can read a value that has been modified but not committed by ignoring the exclusive-mode lock.

Strong atomicity solves this problem by ensuring that all memory accesses maintain transactional integrity. Unlike transactions, read and write barriers used outside of transactions do not constrain compiler reordering. However, at runtime they effectively act as single instruction transactions and are properly serialized with respect to program transactions. Since they consist of a fixed code sequence they can be heavily optimized by the JIT compiler.

Before writing to a field, a thread must place the relevant object in exclusive mode. As described above, this involves replacing the version number in the transaction record field. However, since a strong atomicity transaction consists of a single write, it would be wasteful to create a transaction record. Instead, we use an anonymous lock which does not require a full transaction record. Similarly, it is not necessary to maintain a write log for a strongly atomic transaction.

The invariants required to manage transactional memory are maintained through the use of barriers inserted at every memory access. There are two sets of barriers; those used for standard transactional code, and those inserted around single instructions to provide strong atomicity. These barriers are discussed in detail in Section 5.

#### 4.2 Concurrent Garbage Collection

Our current implementation is of concurrent, but not parallel, garbage collection. That is to say that while the garbage collector runs simultaneously with the application threads, only a single garbage collection thread runs at any given time. During develop-

ment, we found that a single garbage collection thread is sufficient to provide collection for the cores in today's machines. However, this will clearly not be the case in the future. To this end, our algorithm was designed to handle multiple garbage collection threads, and our implementation can easily be extended to make use of them.

##### 4.2.1 Forwarding Pointers

When an object is copied, a record must be maintained of its new destination. In a stop-the-world collector, it is acceptable for this forwarding pointer to overwrite part of the object body; no application thread will see the invalidated object before all references to it have been updated. This is not acceptable in a concurrent collector, where application threads may observe either the original or the new copy.

To the states described in Section 4.1.1, exclusive and shared, we add a third: forwarded. All accesses to an object must be aware of the possibility that the version being observed is not the canonical state, and take corrective action.

The encoding of these three states in the header of an object poses some difficulty. The states must be changed in an atomic action, to avoid races. For example, should two threads want to place an object in exclusive mode, it is essential that one fail. As a result, all three states must be encoded in a single word, allowing modification through an atomic compare and exchange operation.

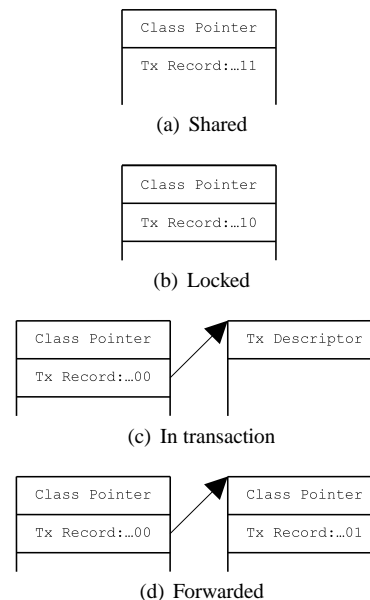


Figure 1. Transaction record states

The transaction record word is encoded as shown in Figure 1. The transaction record of an object in shared mode contains the object's version number, as in Figure 1(a). This is the most common case, and is the only bit-pattern in which the final bit is set, so tests for shared mode objects are cheap. An anonymously locked object, as in Figure 1(b), is in exclusive mode, but the contents of the transaction record cannot be mistaken for an aligned pointer. The final two cases, Figures 1(c) and 1(d), cannot be distinguished by inspecting only the transaction record word. We can recognize either of these cases if the lowest two bits of the transaction record are clear. Since both transaction descriptor and forwarding pointers refer to valid memory addresses, the contents of the transaction record word can be safely dereferenced.

Figure 1(c) shows an object in exclusive mode with a transaction descriptor pointer in its transaction record. The transaction descriptor is not a heap-allocated object, and so does not have a standard header. It is allocated in the Stack region, and is not under the control of the garbage collector. This allows the layout of the record to be contrived to distinguish it from heap objects; the first word of a transaction descriptor was designed to contain non-aligned data which cannot be confused for a pointer. This is in contrast to the first header word of an object on the heap, which is an aligned pointer (Figure 1(d)). In this way, transaction descriptor and forwarding pointers are distinguished.

#### 4.2.2 Atomic object copying

A major difficulty in implementing concurrent copying garbage collection arises when moving objects in the face of concurrent updates. Figure 2 demonstrates how an update can be lost when an object is copied.

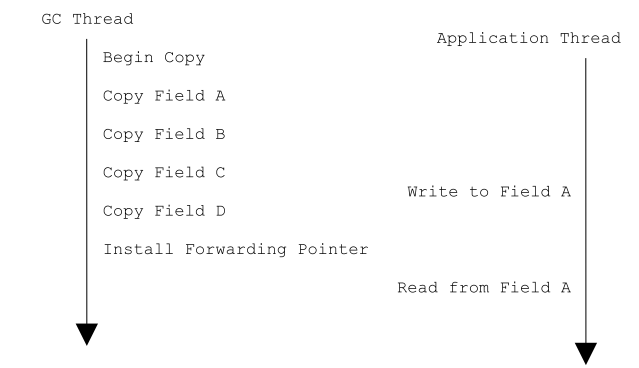


Figure 2. Lost update problem

When copying an object, the GC must copy one word at a time. Until all fields of the object have been copied, the new version is not complete, and a forwarding pointer cannot be installed. When the application thread attempts to write Field A, the modification is performed on the old version. However, at this point field A has already been copied, so the update is not reflected in the copy. When the GC thread installs a forwarding pointer, the application's subsequent attempt to read Field A is redirected to the new location of the object. The value read for Field A is the value as it stood during the object copy, missing the update.

Using our transactional memory infrastructure, we are able to perform the object copy atomically. Each object copy operation can be viewed as a transaction; a concurrent modification as described above will cause the copy transaction to abort and to be retried later.

While performing object copying transactionally solves the lost update problem, the overhead of a full transaction per object copied would be prohibitive. Alternatively, were the copy phase to be wrapped in one or more long transactions, the likelihood of conflicts would greatly increase. Fortunately, we are able to tailor our garbage collector implementation to make use of the transactional memory infrastructure without paying the overhead associated with full transactions.

Rather than creating a transaction per object copy, we can make use of the version number stored in the object header. Before the copy begins, we make a local copy of the contents of the transaction record field. Should the lock bit in the transaction record be set, we know that the object is currently in a transaction. In this case, we fall back to the garbage collector's conflict resolution strategy, discussed in Section 4.2.3. Otherwise, the contents of the field is known to be a version number, and the copy can continue.

The object is then copied field-by-field to the to-space. Since the object is not locked during the copy, it is possible for another thread

in the system to modify the original during the copying process. If this occurs, the two copies of the object will not match, and an update will be in danger of being lost. However, since we insist on strong atomicity, we will be aware of this modification. When we have completed the copy, the final step is to use an atomic compare and exchange instruction to ensure that the version number at the end of the copy operation matches our saved version number. If this is the case, we install a forwarding pointer, referring to the new, canonical, version of the object. If the version numbers differ, there has been a write to the object during the copy operation, and the copy is aborted. The copy may be retried, based upon the conflict resolution strategy.

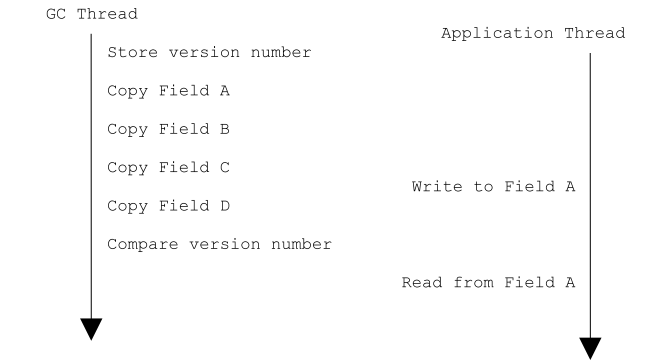


Figure 3. Lost update solution

Figure 3 shows the result of this system on the same sequence of events as before. This time, when the application writes to Field A, the transactional barrier increments the object's version number. When the GC thread has finished copying the fields, it compares the current version number with that it has stored. Since the two do not match, the forwarding pointer is not installed, and the copy is aborted. Finally, the application is able to correctly read the new value of Field A.

This mechanism for transactional object copying introduces minimal overheads. There is no need to acquire locks, and the metadata associated with a full transaction is not necessary. In the majority of cases, where there is no conflict during a copy, the only overhead beyond the copy routine of a stop-the-world collector is the compare and exchange operation needed to check the version number. Additionally, by pushing the resolution of conflicts onto the garbage collector rather than the mutator code, we do not trigger application aborts. The garbage collector never locks an object or increments a version number, ensuring that it will never cause a conflict during a transaction's commit phase.

#### 4.2.3 Conflict Resolution

There are two sources of conflicts within the copy phase of the concurrent garbage collector. An object can be locked before a transaction begins, or an object can be modified during the copy operation. In each case, we make the conscious effort to favor the mutator over the garbage collector. The rationale is that it is preferable for the garbage collector, running in the background of a process, to perform additional work than it is for the application to suffer performance degradation based on the layout of memory.

The currently implemented strategy for dealing with conflicts is to simply spin until the conflict is resolved. This technique is simple to implement and reason about and has proven effective in our investigations to date. This is simply because there are very few conflicts to be resolved, so this basic mechanism does not produce a noticeable slowdown.

It is likely, however, that this will not always be the case. As the number of threads, both software and hardware increase, the

probability of conflicts will grow. In that case, such a simplistic strategy will no longer suffice. By spinning for a limited time, then pinning objects or pages if conflicts persist, the goal of preferring the mutator can be maintained, while allowing the garbage collector to make progress.

#### 4.2.4 Pausing the Mutator

Since we aim to support highly responsive applications, it is important to limit the length of any pauses to application threads. Additionally, since we target systems with many concurrent threads, we aim to avoid stopping all threads simultaneously. We have designed all application pauses with these goals in mind.

There are five circumstances in which threads must be paused: during changes of garbage collector phase, at the start and end of the mark phase, and at the start and end of the flip phase. We now examine each of these pauses.

**Phase Changes** Since different barriers are required for each phase of the garbage collector, it is necessary to ensure that the correct barrier is active before it is required. We control this using a global switch; barriers behave according to the status of a volatile variable. To ensure that all threads are aware of the change in phase, we pause and immediately restart each thread. This ensures that every thread has passed a GC safepoint, and so we can guarantee that the correct barrier is in place.

Each thread is paused four times during a garbage collection cycle, but since no work is performed during the pause, its cost is negligible. The GC only stops one thread at a time and restarts it before moving on the next thread.

**Mark Phase** At the start of the mark phase, it is necessary to gather all references to heap objects from the Stack regions of each thread. Each thread is paused individually while the garbage collector scans the relevant data structures. In order to minimize the pause time, we do not perform any processing on the referred objects until the application thread has been restarted. We simply record all pointers in the Stack region to be processed later. This pause occurs once per GC cycle.

At the end of the mark phase, each thread must be paused again in order to ensure that no unmarked objects remain in the system. Each pause follows the same structure as the first mark phase pause; a thread is suspended, its program stack and transactional data structures scanned, and then restarted. While this pause may occur several times in a given garbage collection cycle, the count is bounded and, in practice, tends to be low (see Section 7.3).

**Flip Phase** The pauses in the flip phase follow a similar pattern to those in the mark phase; each thread is first paused to determine the objects reachable from its Stack region, and then paused again at the end until no unflipped pointers remain. However, more work is required in this phase, as the pointers in the Stack region must be flipped while the thread is paused.

Since flipping a slot involves examining the referred object, threads are generally paused for longer during the flip phase than during the mark phase. However, we aim to keep even flip phase pauses below the 1ms target. We refer the reader to [2] for techniques that bound the amount of the stack work done at each pause.

#### 4.2.5 Triggering the GC

While the garbage collector is designed to minimize pauses to the mutator threads, its running can still affect application performance. Barriers of varying complexity must be executed on all memory accesses while the garbage collector is running, causing some slow-down to application threads. We aim to minimize the time during which the garbage collector is active, while still maintaining sufficient available memory to fulfill allocation requests.

The simplest mechanism to ensure that memory is always available would be to have the garbage collector run continuously; when one collection cycle ends, the next will begin. This will maximize the memory available to the application at the expense of application performance, which will suffer from the additional barriers.

A less costly heuristic, and the one used in our implementation, is to trigger a collection when the available memory falls below a given threshold. We have found that a reasonably conservative threshold ensures that the application does not run out of memory, while substantially cutting down on the number of garbage collections required. Additionally, while a simple threshold has proved sufficient for our purposes, it is possible that an adaptive scheme that monitors the application's allocation rate could provide similar availability qualities while further reducing the number of collections required.

We also take advantage of the fact that the whole heap is not subject to copying during a garbage collection. We are able to dynamically select which blocks will make up the Collected region at each collection, deciding after the mark phase, but before the copy phase. This allows the garbage collector to make an informed choice on which blocks will yield the highest ratio of garbage objects.

## 5. Read and Write Barriers

Both transactional memory and our concurrent GC require read and write barriers. The transactional memory system uses read and write barriers to enforce specific memory access protocols. Weakly atomic STMs isolate transactions only with respect to other transactions and, thus, require barriers only in transactional code. Strongly atomic STMs provide full isolation of transactions - both from transactional and non-transactional code, and, thus, require barriers in both transactional and non-transactional code. Concurrent GC needs barriers during copy and flip phases to find the correct version of the object through the forwarding pointer. Our system combines the barriers and, thus, imposes little additional overhead to support concurrent GC compared to the underlying strongly atomic STM without such support.

The combined barrier exploits similarities between transactional and GC barriers and eliminates duplication of the operations common to both barriers. For example, a standard transactional read barrier for reads inside a transaction locates transaction record for the object, checks that it is a version number (indicating that object is in shared state and can be read) and then records the address of transaction record and its value in the read set. If it detects that transaction record is in exclusive state it invokes a contention manager that either waits for the transaction record to become available (potentially by asking the transaction that owns the transaction record to abort) or aborts the current transaction. Note, that if the object is forwarded its transactional record is located in the forwarded rather than original object, so transactional barrier has to follow forwarding pointer to locate the transaction record. A GC read barrier also follows the forwarding pointers; it locates the forwarded object and then reads the field from that object. The combined barrier eliminates duplication of following the forwarding pointers. It finds the forwarded object when looking for the transaction record and then uses that object for the read. This optimization is legal even if object gets forwarded in between of locating transaction record and reading the object field. During validation the transaction will detect that the object was forwarded and use the transaction record in the forwarded object to determine if the transaction validates. The combined barriers for transactional writes and non-transactional reads and writes operate in a similar way. They locate the forwarded object during the transactional part of the barrier and then use that object for read or write. In accor-

dance with the requirements of our GC algorithm, the barriers for pointer writes also follow forwarding pointers.

Our base transactional memory system uses aggressive compiler optimizations to reduce the overhead of the read and write barriers. It explicitly represents the barriers in the program intermediate representation and then applies both traditional and novel optimization to eliminate redundant barriers. For example, it eliminates redundant barriers generated due to multiple accesses to the same object (the barrier representation in the IR is designed in such a way that this elimination is handled by standard common subexpression elimination). It also eliminates barriers on the fields that it finds immutable (such as final fields) or local (such as fields of objects allocated in a transaction). We extend this approach to combined barriers - we represent them in the IR as special `openCopyForRead` and `openCopyForWrite` operations that return the forwarded object and let the compiler optimizations take care of redundancies. In some situations, the rules for eliminating combined barriers are exactly the same as the rules for eliminating transactional barriers. For example, we do not need either barriers for reads from the final fields because these fields cannot be written by another thread and have the same value in the original and forwarded object. But there are notable differences. For example, memory accesses to thread local objects do not require transactional barriers, but still need GC barriers. In such cases, instead of barrier elimination we do barrier simplification, where we replace the combined barrier by a lighter weight GC barrier.

Both our concurrent GC system and the base transactional memory system inline the fast paths of the read and write barriers into compiler-generated code. We carefully engineered the transaction record encoding so that the fast paths of transactional and combined barriers are identical for all accesses except for reference writes. A standard transactional barrier starts from checking if the transaction record is in the shared state, that is if its value is an odd number. In our system this check also detects the fact that the object is not forwarded (forwarding pointer is properly aligned and is, thus, even). In this case, the existing fast path of the transactional barrier provides the correct functionality because the memory access to non-forwarded object does not require GC barrier. If the object is forwarded, the execution follows the non-inlined slow path.

## 6. Related work

Herlihy and Moss [15] coined the term transactional memory and proposed hardware to implement it. Knight [18] proposed hardware to implement futures that had many of the same characteristics.

Shavit et. al. [23] introduced the term software transactional memory where the user statically marked memory location involved in transactions. Herlihy et. al. [14] and Marathe et. al. [20] provided object-based STMs for Java while Fraser [9] provided an object based STM for C. Like us, these system create object clones during transactions and install the new copy when upon commit. A crucial difference is that our copy operations does not modify the object and is thus lighter weight.

Earlier STMs for managed environments treated garbage collection in various ways. Harris et. al. [12] use a stop-the world garbage collector to prevent version number overflow but otherwise aborts transactions interrupt by garbage collection. Adl-Tabatabai et. al [1] presented an STM that allowed transactions to persists through garbage collection. Harris et. al. [13] presented a C# STM system that further extends GC to compact STM log structures by removing entries that are redundant or represent unreachable objects thereby enabling larger transactions. Shpeisman et. al. [24] presented strong atomicity extensions to our base STM and demonstrated garbage-collection style traversal over objects to implement dynamic escape analysis. In contrast, we explore synergies between

STM and GC in the opposite direction. As far as we are aware, we are the first to present a garbage collector that leverages STM.

Concurrent GC has a long and distinguished history. Dijkstra [8] and Steele [11] presented the basic concurrent marking techniques we now call the tri-color algorithm. Blackburn [4] showed that most distributed GC algorithms can be decomposed into a GC algorithm and a termination algorithm. The algorithms we present in our concurrent setting have the same characteristic, and the termination algorithms we use can be replaced without loss of generality.

Moss and Herlihy [16] in 1990, while also working on transactions, proposed a collector that used multiple version of objects in ways similar to what we do here. Private conversations with Moss makes it clear that the invention of transactional memory was at least at part a response to how to do concurrent GC.

We are most closely related to the Sapphire algorithm designed by Hudson and Moss [17]. The mark, flip, and sweep phases borrow heavily from this algorithm and therefore leverage the termination arguments presented in their papers. The copy phase is different and instead of maintaining dynamic consistency between different versions of the same object we use the transaction read and write barriers to locate the definitive object. The largest difference between the algorithms is that Sapphire relied only on write barriers while we rely on both read and write barriers.

Azul's Pauseless GC by Click et. al. [7] has many of the same goals as our collector. Their read barrier is implemented in hardware, while we piggyback on the software transaction mechanisms. Such hardware support if coordinated with transactional hardware support would likely improve the performance of our collector also. The Pauseless collector is fully parallelized and nothing in our algorithms prevents full parallelization. Once CMP desktops supports tens or hundreds of cores, paralleling the GC will be appropriate.

Brooks [6] used a forwarding pointer to implement an efficient read barrier that avoided conditional branches. Branch prediction and out of order execution in current hardware date this engineering trade-off somewhat but reduction of cache pressure is still a concern that we addressed using similar techniques.

Bacon et. al. [3] designed the Metronome Collector as a hard real time GC and uses similar read barrier data structures as both Brooks and ourselves. Our algorithm is targeted at human interactive desktop applications where soft real-time is a more appropriate trade-off. The biggest difference is that Metronome either runs on a single processor or apparently requires a stop the world protocol to achieve their goals while we only stop a single mutator thread at a time. The techniques we present could be merged with theirs and a hard real time GC could be realized.

Pizlo et. al. [21] present an algorithm that uses of existing hardware features to achieve lockless real time behavior. Our algorithm differs, leveraging the TM infrastructure to do object copying, and potentially ameliorating mutator performance issues.

## 7. Experiments

### 7.1 Methodology

We aim to show that our garbage collector can improve the responsiveness of typical desktop applications. To this end, we have selected a range of programs to demonstrate responsiveness.

The SPEC JVM98 suite consists of several single-threaded, short-running applications which offer a reasonable approximation to simple user tasks. We show results for all seven SPEC JVM98 benchmarks. All SPEC JVM98 benchmarks were run with a heap size of 32Mb, which was judged to be appropriate and realistic for applications of their size.

SPECjbb represents a heavier workload, with multiple user-level threads. It uses a lock-based synchronization model. The results presented were gathered during the two-minute timing mea-

surement period of a three-warehouse execution. The heap size was set to 1024Mb.

Atomicjbb is a modified version of SPECjbb which uses transactional synchronization rather than locks. We ran Atomicjbb in the same manner as SPECjbb, using three warehouses and a heap size of 1024Mb. Again, results were gathered during the two-minute timing run.

Finally, AtomicTSP and AtomicOO7 are small transactional workloads. AtomicTSP implements a Traveling Salesman Problem solver, while AtomicOO7 is a synthetic design database benchmark. These applications provide an approximation of simple transactional user tasks. All numbers were gathered on a Dell Precision\* 490 with 2 x 3.0 GHz Xeon®5160 (total of 4 cores) with 2 GB of RAM running Microsoft Windows\* Server 2003.

## 7.2 Pause Times

The primary goal of this garbage collector is to leverage transactional memory and multi-cored processors to minimize disruption to user applications. We measure this by pause times; the shorter the pause, the less disruption to the application. We define the length of each pause to last from just before a given thread is suspended to just after it is resumed. Each pause interval applies to exactly one thread; our system does not pause multiple threads simultaneously, as doing so would increase the work required and so the pause time.

In Figure 4 we show the distribution of pause times for each benchmark. Each bar in a histogram represents a 10ns bucket. The x-axis scale only covers pauses up to 2ms; representing all pauses would require a higher measurement granularity, obscuring detail at the low end. Figure 5 indicates the distribution of outlying values.

In general, the histograms demonstrate the properties we aimed to achieve. Each histogram has a sharp peak for pauses between zero and 10ns; this is expected, as the garbage collection algorithm requires that each thread be paused briefly to transition between phases (as described in Section 4.2.4). While these pauses are small, they do account for disruption to the application thread, and so are relevant to this metric.

The SPEC JVM98 benchmarks (Figures 4(d) to 4(j)) are representative of the many short-running applications that a desktop user will run. In six of these applications, all pauses take less than 1ms; the remaining application, `_213_javac`, had two pauses over this limit, both of which were less than 1.2ms. This demonstrates the level of responsiveness expected, even in trivial applications.

Figure 4(k) shows the pause times for SPECjbb. This data was gathered during the timing run of the benchmark, representing the steady state performance of a large, multi-threaded application. Such applications are also seen on the desktop, in the form of games or multimedia players. As can be seen, the performance characteristics for large applications are similar to those for small, with 98.8% of pauses lasting for less than half a millisecond.

Figure 4(l) shows the breakdown of all pauses over all benchmarks. This figure is not derived from the other graphs; it simply totals the pauses from the raw output of the benchmarks. As a result, it is representative of the overall expected performance on a desktop machine. As can be seen, virtually all pauses (98.9%) take less than one millisecond. Further, 96.9% of all pauses are of less than half a millisecond, significantly beating our goal.

Aside from keeping the majority of pauses below 1ms, we also aim to minimize the outlying pauses. Keeping 99% of pauses within a millisecond will offer little consolation if the remaining 1% cause applications to stall for a noticeable time. While we do not aim to provide hard real-time, we do want our garbage collector to be usable under reasonable circumstances. Figure 5 indicates the breakdown of pauses by length category. As mentioned, only one of the SPEC JVM98 benchmark applications has any pauses greater than a millisecond. SPECjbb, despite being a larger and longer-

| Benchmark                   | <1ms     | 1..10ms | 10..100ms | >100ms |
|-----------------------------|----------|---------|-----------|--------|
| <code>_201_compress</code>  | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| <code>_202_jess</code>      | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| <code>_209_db</code>        | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| <code>_213_javac</code>     | 99.43%   | 0.57%   | 0.00%     | 0.00%  |
| <code>_222_mpegaudio</code> | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| <code>_227_mtrt</code>      | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| <code>_228_jack</code>      | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| SPECjbb                     | 99.72%   | 0.14%   | 0.14%     | 0.00%  |
| AtomicOO7                   | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| AtomicTSP                   | 100.0%   | 0.00%   | 0.00%     | 0.00%  |
| Atomicjbb                   | 85.00%   | 12.50%  | 2.14%     | 0.36%  |
| Total                       | 98.92%   | 0.85%   | 0.21%     | 0.02%  |
| Goal                        | ≥ 90.00% | ≤ 9.00% | ≤ 0.90%   | ≤ 0.1% |

Figure 5. Pause time percentages

running application, has no pauses of more than 100ms, and only very few in the 1 – 100ms range.

Atomicjbb is the only application that does not meet our pause time goals. It has one pause of greater than 100ms, while the number of pauses in other categories are outwith the target ranges. This is most likely attributable to the number of long-running transactions. As discussed in Section 4.2.4, both the mark and flip stages require that each thread's Stack region is scanned. In non-transactional applications, this refers simply to the objects in the Java stack, as well as some metadata. However, transactional code must also provide references from the read and write logs that reside in the Stack region. As transactions touch memory, these data structures grow and so take longer to be scanned. It is likely that future redistribution of transactional memory responsibility from software to hardware will greatly improve the efficiency of this scanning, and so reduce these pause times.

## 7.3 Pause Frequency

During the discussion of the garbage collection algorithm in Section 3.3 there was the question of ensuring that all reachable objects had been marked or flipped, depending on the phase. The solution involved pausing each thread repeatedly until no white references remained. This introduces an element of non-determinism into the number of pauses in each garbage collection cycle.

Clearly, it is desirable that the application threads experience as few pauses as possible. Figure 6 provides the average number of pauses per application thread in each phase of the collection. These numbers do not include the necessary brief pause required when changing phase. However, they do include every other pause within the phase, including the initial scanning of the root set.

Generally, more pauses are required during the mark phase than during the flip phase. This may be because the copy and flip phase barriers automatically update unflipped references in the heap, meaning that few unflipped references are stored in the Stack regions. A general trend that can be seen in Figure 6 is that applications with more threads take longer to converge than those with few threads. This is to be expected, since the act of scanning multiple threads allows more time for unmarked or unflipped references to arrive on the stack. However, it can be seen that the average number of pauses is low, meaning that the termination algorithm is not likely to significantly disrupt application performance.

## 7.4 Barriers

The garbage collection algorithm requires that special barriers be present during the various collection phases. As these barriers are more heavyweight than those required by the transactional memory system alone, it is desirable to minimize the time during which

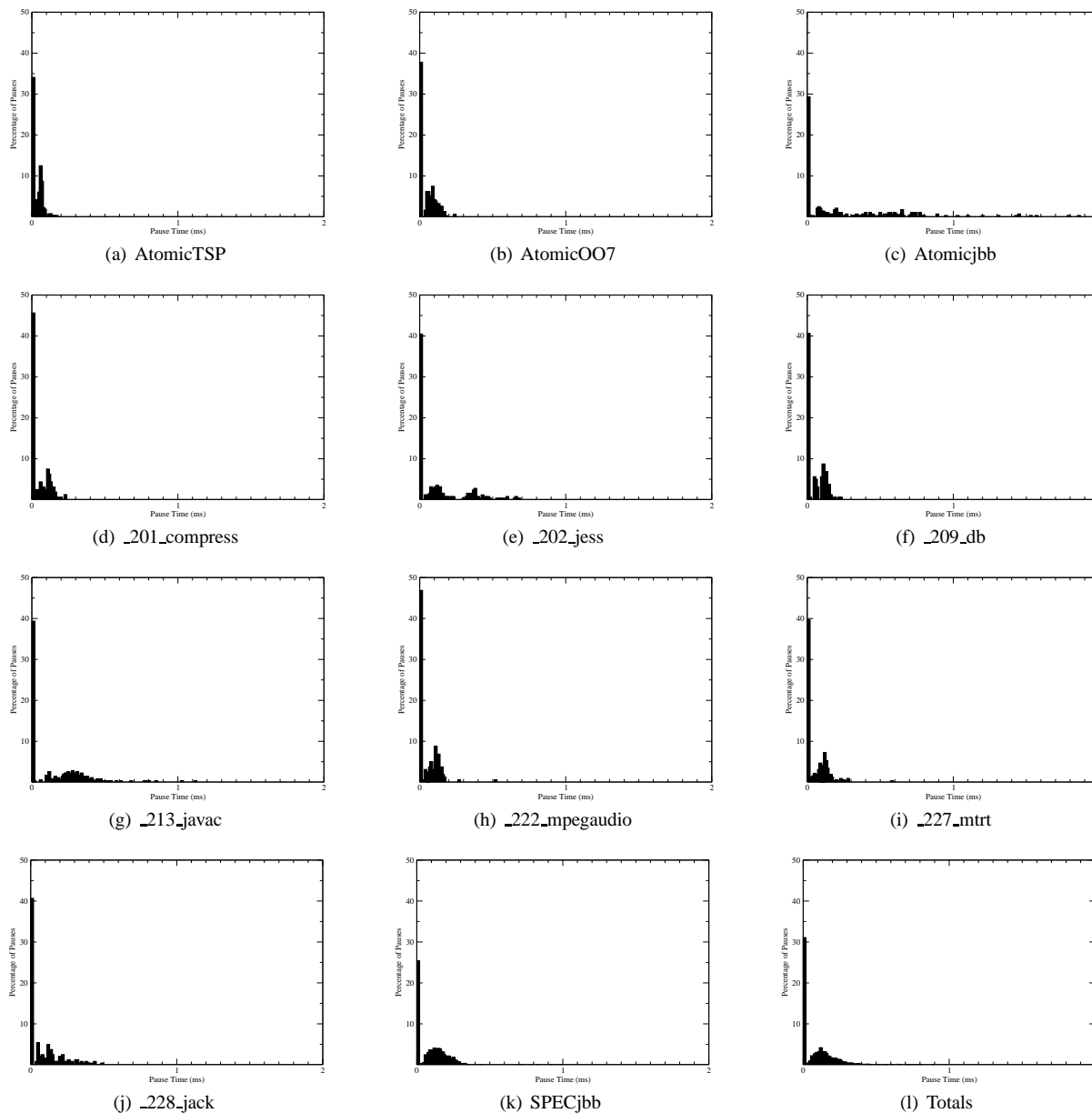


Figure 4. Pause times

they are in effect. Figure 7 shows the percentage of time that the various barriers are in effect. The interval is measured from when the first thread is paused to observe the phase change until the first thread is paused for the subsequent phase change. There is some small overlap of barriers as each thread is paused individually. However, this overlap time, when considered as a percentage of the application’s run time, is insignificant.

It can be seen that, even for applications with short run times, the percentage of time in which garbage collection barriers are enabled is very small. The copy phase barrier is in place for the shortest time, due to the policy of collecting only a small fraction of the heap in any garbage collection cycle. As a result, the copying overhead is small. The mark and flip barriers must be enabled for a

longer period, since these phases must touch every reference in the heap in order to ensure that no reachable objects are missed.

It should be noted that the garbage collection triggering strategy described in Section 4.2.5 is relatively naïve. By tuning this algorithm, as well as the sections of the heap to be collected, it is likely that the barrier usage could be decreased further.

## 8. Conclusion

In this paper we have presented a synergistic combination of strongly atomic transactional memory and concurrent garbage collection. Our GC algorithm makes use of the transactional memory mechanisms to avoid errors when copying objects, and the JIT compiler combines the GC and TM barriers to improve efficiency.

| Benchmark      | Mark Phase | Flip Phase | Total |
|----------------|------------|------------|-------|
| _201_compress  | 2.00       | 2.00       | 4.00  |
| _202_jess      | 2.60       | 2.00       | 4.60  |
| _209_db        | 2.00       | 2.00       | 4.00  |
| _213_javac     | 2.67       | 2.00       | 4.67  |
| _222_mpegaudio | 2.00       | 2.00       | 4.00  |
| _227_mtrt      | 3.70       | 2.90       | 6.60  |
| _228_jack      | 2.05       | 2.00       | 4.05  |
| SPECjbb        | 5.64       | 2.67       | 8.31  |
| AtomicOO7      | 3.58       | 2.00       | 5.58  |
| AtomicTSP      | 2.00       | 2.00       | 4.00  |
| Atomicjbb      | 4.00       | 2.00       | 6.00  |
| Average        | 2.93       | 2.14       | 5.07  |

**Figure 6.** Average number of pauses per GC

| Benchmark      | Mark  | Copy  | Flip  | Total |
|----------------|-------|-------|-------|-------|
| _201_compress  | 0.19% | 0.08% | 0.26% | 0.53% |
| _202_jess      | 0.91% | 0.37% | 1.18% | 2.45% |
| _209_db        | 0.43% | 0.14% | 0.44% | 1.00% |
| _213_javac     | 0.82% | 0.17% | 0.82% | 1.81% |
| _222_mpegaudio | 0.22% | 0.08% | 0.27% | 0.57% |
| _227_mtrt      | 1.81% | 0.61% | 2.02% | 4.44% |
| _228_jack      | 0.77% | 0.36% | 0.58% | 1.71% |
| SPECjbb        | 1.27% | 0.27% | 1.02% | 2.56% |
| AtomicOO7      | 0.04% | 0.01% | 0.03% | 0.08% |
| AtomicTSP      | 0.51% | 0.00% | 0.00% | 0.51% |
| Atomicjbb      | 1.37% | 0.52% | 2.39% | 4.28% |
| Average        | 0.76% | 0.24% | 0.82% | 1.81% |

**Figure 7.** Percentage of execution with barriers enabled

We aimed to demonstrate that 90% of pauses caused by our garbage collector lasted for less than a millisecond. Of the remainder, we looked for 90% to be under 10ms, 90% of those remaining to be under 100ms, and so forth. Our experimental results showed that, over all workloads, we met those targets. Indeed, we far exceeded them, with 98.82% of all pauses taking less than a millisecond and 96.9% of all pauses taking less than half a millisecond.

## Acknowledgments

This work is partially supported by the National Science Foundation under grants Nos. CNS-0720505, CCF-0702240, and CNS-0551658. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

## References

- [1] A.-R. Adl-Tabatabai, B. T. Lewis, V. S. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.
- [2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 11–20, New York, NY, USA, 1988. ACM Press.
- [3] D. Bacon, P. Cheng, and V. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *In Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems*, 2003.
- [4] S. M. Blackburn, R. L. Hudson, R. Morrison, J. E. B. Moss, D. S. Munro, and J. Zigman. Starting with termination: a methodology for building distributed garbage collection algorithms. In *ACSC '01: Proceedings of the 24th Australasian conference on Computer science*, pages 20–28, Washington, DC, USA, 2001. IEEE Computer Society.
- [5] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.
- [6] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM Press.
- [7] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2005. ACM Press.
- [8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.
- [9] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Technical Report UCAM-CL-TR-579.
- [10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [11] J. Guy L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.
- [12] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*.
- [13] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI 2006*.
- [14] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC 2003*.
- [15] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993*.
- [16] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. In *SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 229–236, New York, NY, USA, 1991. ACM Press.
- [17] R. L. Hudson and J. E. B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, 2001.
- [18] T. Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, 1986. ACM Press.
- [19] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), 1981.
- [20] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *LCR 2004: Languages, Compilers, and Run-time Support for Scalable Systems*.
- [21] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th International Symposium on Memory Management*, pages 159–172, New York, NY, USA, 2007. ACM.
- [22] B. Saha, A.-R. Adl-Tabatabai, R. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006*.
- [23] N. Shavit and D. Touitou. Software transactional memory. In *PODC 1995*.
- [24] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. In J. Ferrante and K. S. McKinley, editors, *PLDI 2007: Programming Language Design and Implementation*, pages 78–88. ACM, 2007.