Nested pessimistic transactions for both atomicity and synchronization in concurrent software

by

Tarek Chammah

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Existing atomic section interface proposals, thus far, have tended to only isolate transactions from each other. Less considered is the coordination of threads performing transactions with respect to one another. Synchronization of nested sections is typically relegated to outside of and among the top-level flattened sections. However existing models do not permit the composition of even simple synchronization constructs such as barriers. The proposed model integrates synchronization as a first-class construct in a truly nested atomic block implementation. The implementation is evaluated on quantitative benchmarks, with qualitative examples of the atomic section interface's expressive power compared with conventional transactional memory implementations.

Acknowledgements

This work would not be possible without the support and dedication of the people around me.

As the hardest working secretary in the school, Margaret Towell is constantly on top of all computer science graduate related issues. Margaret stays in the office later than many graduate students, and has helped me in many ways - academic and personal.

Prabhakar Ragde is the best educator in the school and his dedication to teaching the undergraduates shows in every class he prepares and the wealth of knowledge he endows his lectures with. Thanks for motivating me to explore further with the functional paradigm.

William Durocher and Nancy Soontiens. Running a dojo is a tough selfless job and the dedication and effort put into every Karate practice with us is much appreciated. You do it purely for the love of the art and it shows in the camaraderie and esprit de corps around the dojo.

As the senior researcher in our group, Nomair Naeem has given me advice and support in my work and academic life. When necessary he has also given me a dose of tough love to get moving!

Peter Buhr is the director of our lab and the doyen of concurrent programming practice and experience on the faculty. An entertaining lecturer, Peter's enthusiasm for the subject matter is contagious. As a reader of my thesis Peter has set a high bar for quality. Thanks for making this a better thesis Peter.

Jonathan Rodriguez and I started in our respective programs in the area of concurrency a term apart. Though our paths have since diverged, I fondly remember the many late night chats we had early on as colleagues in the lab on technical issues pertaining to concurrency models, and their implications. As well, it was often enlightening and always entertaining discussing general topics varying from economics to history and culture, among countless other endeavors we shared.

Ondrej Lhotak as my advisor and friend allowed me wide latitude in pursuing many disparate interests on campus. I have attended a multitude of talks of varying subject matter. In

addition I have pursued extracurricular team endeavors and volunteer activities, and sat in many a class lecture from different faculties. With freedom comes responsibility and in retrospect, due to my inability to stay focused, this has not always been beneficial to me. Yet I appreciate his acceptance of means of enrichment beyond the mere academic. During my thesis revision process, Ondrej helped me improve my technical writing for more precise academic communication. I wish we had communicated as effectively in person towards my research goals. The process of creating this thesis has been a painful birth of sorts, and I am very grateful for Ondrej's patience and support over the course of my Masters thesis work. Ondrej has always been respectful and professional; his dedication to the field and our group has been a constant source of inspiration. Thank you for everything.

To my parents

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Concurrent programming has traditionally been a bane of developers for decades. Ever since shared-memory locks and condition variables were implemented in the context of operating systems in the 1970s, programmers have been grappling with concurrency bugs such as race conditions, deadlocks, and related ills [LPSZ08]. What was once a curiosity for the average programmer - being the domain of systems developers and parallel computing centres - concurrent computing has now forced itself into the mainstream of computing consciousness [SL05].

Now that single-core performance improvements have largely halted due to a multitude of factors - a confluence of power dissipation, wire scaling, and instruction level parallelism limits - the burden for increasing performance has fallen onto the typical developer's shoulders to program for greater numbers of computing cores, which have become ubiquitous even on consumer devices [OH05].

The average programmer has to navigate the myriad language memory-models and concurrency libraries on a given platform - usually based on locks and condition variables - in order to construct and attempt to reason about concurrent programs, which are likely to contain subtle bugs that may manifest themselves only years down the line. While there have been effective dynamic race-condition detectors developed, the instrumented programs' performance can be degraded by a factor of ten or more and the false positive rates are excessively high [SBNSA97] [Nish04].

Ensuring mutual exclusion for concurrently accessible data can be done through coarse or fine-grained locking schemes. Employing a few coarse-grained locks to protect access to program modules is straightforward, in that it simplifies the problem. However, parallel performance suffers due to insufficient granularity of data able to be accessed in parallel. For example, a big lock protecting a hash table prevents concurrent access to distinct buckets that a finer-grained locking scheme allows. Yet ensuring a program is deadlock free is a difficult

task to achieve in the presence of fine-grained locking, which is necessary to improve the performance of concurrent code. However, unlike race conditions, deadlocks [CES71] are easy to detect once threads are intertwined in a deadly embrace, as they manifest themselves in a program or component hang.

Bugs inherent in traditional shared-memory programming are only one facet of the problems plaguing concurrent programs. Another issue concerns the composability of software in the presence of explicit fine-grained locking. Disparate modules and objects may need to be accessed whilst holding unrelated locks. Thus, arbitrarily code cannot be refactored and the program expected to work in the presence of an increasingly complex maze of locks, due to intertwined concerns of mutexes and the code they protect. Therefore, the inner workings of sub-modules and the data accessed therein must be known and understood in order to properly refactor such programs. Hence, a program composed of nominally separate modules cannot be reasoned about one module at a time. Changes must necessarily take into account all code and data accessed within the scope of the program. Software engineering best practices must often be broken when these separate concerns are intertwined.

1.1 Transactional Memory

As a response to this dilemma, in recent years a concurrent programming abstraction known as transactional memory [HM93] has gained prominence. Transactional memory provides for mutual exclusion for all data accessed within the confines of a transactional block of code. Originating from the database world, the notion of a transaction [Lome77] provides the properties of Atomicity, Consistency, Isolation, and Durability (ACID). Atomicity provides the guarantee that a sequence of statements executes indivisibly. In essence, its effects are observed to occur all at once, or not at all. Consistency ensures the effects of a transaction transform the program's state such that its logic is not violated and its invariants are maintained. Code inside a transaction is nominally isolated from the effects of statements in separate concurrent transactions. Were concurrent transactions to access the same data with at least one of them mutating it - a conflict – the transactions must be

serialized, using an abort and retry mechanism. This method, serialization, ensures the effects of one transaction are isolated from those of another. Durability is a property not present in transactional memory, though in the context of databases, assures successfully executed transactions are saved to a stable store, such that state is not lost in the event of a malfunction.

Programs constructed with such transactions exhibit atomicity of statements within atomic sections. In addition, statements within disparate atomic sections are isolated from one another. Transactional memory does not suffer from the deadlock and race condition problems plaguing traditional programmer specified lock-based concurrent code. Another benefit of transactional memory is the all or nothing nature of transactions that commit or abort, due to its atomicity property. The state of the program is never left in a half mutated state, but is consistent with the full updated results from a completed transaction, or rolled back to the state that existed before memory was mutated by an aborted transaction.

In transactional memory, an atomic section's read and write sets constitute the sets of memory cells read and written respectively by statements executed within the section at runtime. Statements within atomic sections execute optimistically assuming exclusive access to their portion of the program state. If a section's read and write sets conflict with another's then all but one of the transactions aborts, meaning their effects are rolled back to the prior state before the transaction(s) executed. Otherwise, a transaction succeeds upon commit, meaning its effects are made visible to other threads.

Transactional memory suffers from the cardinal problem of not handling irreversible operations well. Such operations include operating system calls and I/O routines which cannot in general be reversed through a rollback of state due to interactions with the physical world. Typical implementations execute only one transaction containing irreversible operations at a time - a singular master transaction - that can abort any transactions it comes into conflict with. Obviously the lack of concurrency for such transactions limits the potential of transactional memory in I/O heavy applications.

Transactional memory can be implemented purely in software [ST95]. Software transactional memory must record all memory locations read and written to as well as all values mutated in the course of a transaction. The extra bookkeeping costs typically result in significant performance degradation due to the overhead of tracking of read and write sets and storing prior values in the runtime implementation. In practice, some of the overhead can be mitigated through dataflow analysis, as well as dynamic filtering to remove redundant bookkeeping [Har09].

1.2 Pessimistic Transactions

An alternative to software transactional memory is to utilize pessimistic atomic sections [MZGB06] to implement the transactional memory semantics. Such a solution involves performing static program analysis to infer a set of locks for each atomic section. These locksets correspond to the abstract memory locations affected during the execution of the section at run time. The inferred locks ensure an atomic section has exclusive access to update its portion of the program's state; therefore, by definition, its effects cannot conflict with those of another section. The original programs are transformed to utilize locking libraries in the implementation to ensure mutual exclusion. Pessimistic atomic sections can permit concurrency of irreversible and I/O operations because actions in disparate sections are predetermined not to conflict. Therefore, arbitrary irreversible operations are permitted since nothing is rolled back.

However, such implementations also suffer from performance degradation due to conservative static program analysis necessary to predetermine the program execution at runtime is faithful to the semantics of transactional memory. An important analysis employed is that of determining the sets of memory locations a program pointer points to. Statements can affect memory indirectly through pointers and a determination of which locations are aliased is necessary in order to guarantee atomicity of shared-memory locations. The locksets inferred from these sets of shared-memory locations are necessarily imprecise as the runtime states of arbitrarily complex programs cannot be computed statically; it is in general an undecidable problem. Such conservative assumptions do not typically permit as

large a number of concurrent transactions to execute in parallel even when they can actually do so at run time without conflict, as implemented in software transactional memory.

1.3 Condition Synchronization

A general problem with transactional concurrency models is the lack of support for condition synchronization. Condition synchronization is a mechanism that enables threads to communicate in order to wait for each other, enforcing an order of execution between them. For example, a consumer thread, when it encounters an empty queue, can be made to wait until a producer thread fills the queue and notifies the consumer.

Condition synchronization has traditionally been implemented using condition variables, or built into monitors. A condition variable supports two operations: signal (or notify) and wait. The wait operation blocks the thread that calls it until another thread invokes the signal operation on the same condition variable. Synchronizing atomic operations on certain conditions determines what can occur before and after certain program states, which is necessary to partially order concurrent operations.

Consider the case of a thread arriving at a barrier and only proceeding when all threads have arrived:

```
void enterbarrier() {
    atomic {
          count+=1;
    }
    atomic (count == thread_count) {
          ...
    }
}
```

This procedure waits until the number of threads that have reached the barrier is equal to the thread count. Upon entry into the first atomic section, each thread atomically increments the count of the total number of threads entered thus far. The thread then evaluates the predicate "`count == thread_count`", which becomes true when all threads have

reached this point in the routine. If a predicate evaluates to false conventional implementations would retry the (top level) atomic section.

If a developer calls `enterbarrier()` from within another (conventional) transaction:

```
atomic {
    ...
    enterbarrier();
    ...
}
```

The conventional transactional memory implementation would never allow the incremented count values from other threads to be made visible to a thread executing the barrier, due to the isolation property maintained by the outermost transaction. Thus, the evaluated predicate would never become true, due to the rollback of the increment. Therefore, in a conventional optimistic implementation the transaction never completes, resulting in a live-lock situation. It is the responsibility of the programmer to ensure such transactions are not composed, which would result in this error.

In this thesis, inserting a new parent transaction around the previously written transactions allows the newly composed system to function as a whole, promoting modularity. Thus, in the barrier example, the value of the incremented count variable is immediately made visible to the nested waiting transactions, and hence, the predicate eventually is satisfied when all threads reach the barrier, permitting them to pass.

A new atomic section interface model is presented that enables certain types of composition. However, due to the model's expressive power, the developer is not prevented from crafting erroneous predicate code that prevents forward progress. Nevertheless, it is desirable to coordinate threads with condition synchronization, and predicates - whether inserted into conditional statements in transactions lacking composability or promoted to first class status as part of the atomic section interface - cannot (in general) be evaluated statically to guarantee forward progress of the program at runtime.

1.4 Thesis Contributions

With conventional nested atomic section semantics, a nested transaction ensconced in another cannot make its effects visible to other threads upon commit. Only the successful commit of its top-level parent transaction ensures the new state becomes visible. Therefore, reusing the example of the barrier mechanism as part of a higher-level parent transaction [SKBY07] prevents state changes in the nested transaction from being made visible to other threads entering the barrier because aborting and retrying the top level transaction upon a failed inner predicate is not a plausible solution. As a result, there can be no progress in a thread's execution as it rolls back its increment of the count variable; therefore the predicate can never be satisfied.

The outcome whereby code reuse necessitates reasoning about the inner workings of callees and refactoring the program to guard against breakage is clearly unsatisfactory as again the notion of proper abstraction and composability of code within concurrent software is compromised, similar to explicit locking. Basic concurrent solutions such as barriers and producer/consumer patterns cannot be composed with other transactions. Attempting to reuse formerly top-level transactions within other transactions unfortunately renders these patterns unusable, thus breaking the modularity promise of transactional memory.

However, synchronization among threads entering the barrier under a parent transaction can be ensured by employing a new kind of synchronization variable for the `count` value evaluated in the predicate to guard entry to an atomic section. Sync variables allow for nested atomic sections to engage in condition synchronization by perforating the transaction's atomicity in order to permit restricted communication of state changes among transactions. This novel mechanism enables certain classes of composability, but can also limit potential concurrency among atomic sections directly accessing the same sync variable. However, the parent sections of these atomic sections can nevertheless execute concurrently.

This work presents an attempt to craft a semantics and develop an implementation for pessimistic atomic sections that promotes local program reasoning and easy refactoring in order to ensure modularity is maintained. Piercing atomic sections in this manner is shown

to support expression of condition synchronization in a composable fashion and reasoned about with relative ease. This work shall further endeavor to demonstrate these assertions through an efficient system and software crafted in order to assess the stated aims of this work. Both quantitative results and qualitative samples shall advance the robust claims contained herein.

1.5 Thesis Structure

The rest of this work is organized into the following sections: Chapter 2 surveys prior related works and provides the requisite background and terms of reference for the thesis. Chapter 3 discusses the language constructs introduced and their semantics. Chapter 4 delves into the implementation of the system and the design decisions considered. Chapter 5 presents quantitative benchmark evaluations of the system and includes qualitative results and their interpretation, as well as an implemented system featuring the model. Chapter 6 is the coda of this work and summarizes the thesis. It also contains a section on desiderata, potential extensions and possible avenues of exploration.

Chapter 2
Related Work

This chapter discusses transactional memory, which was originally introduced as a hardware extension; software transactional memory, which avoids the requirements for special hardware; pessimistic atomic sections, which maintain the interface and much of the semantics of software transactional memory yet do not undertake speculative operations; and finally condition synchronization, a method of ordering concurrent operations and atomic statements.

2.1 Transactional Memory

The inception of the field of transactional memory was initiated by Herlihy [HM93] based on some processor architectures with synchronization idioms in the form of a load-linked and store-conditional pair of instructions. Upon loading a datum from memory (with the load-linked instruction) the associated datum address is recorded. A sequence of computation instructions is allowed on the datum until it is ready for storing (with the store-conditional instruction) back to memory. If in the meantime the address has been written to by another processor, or an exception occurred from one of the computations on the datum, or an interrupt was signaled, then the store fails, and the sequence has to be retried.

In Herlihy's initial (hardware) transactional memory proposal, this instruction sequence was extended to permit multiple memory locations to be tracked. Thus, an atomic sequence featuring multiple variables could be attempted within a transaction. Support for maintaining transactional coherence requires the use of existing multiprocessor coherence protocols, and a buffer or part of the cache to maintain speculative state. As the initial proposal required dedicated hardware support, some sought to achieve similar semantics purely from the software system, without specialized processor structures that often have very limited sizes and may never be supported by industry.

2.2 Software Transactional Memory

Software transactional memory (STM) became popular as a method to support transactions, requiring separate per thread speculative heap state and conflict detection among ongoing transactions. Software transactional memory implementations must record all memory locations read and written to as well as all values mutated in the course of a transaction. The extra bookkeeping costs typically result in significant performance degradation - more than an order of magnitude over an equivalent sequential program - due to the overhead of tracking of read and write sets and mutated values in the runtime implementation. In the initial work of Shavit [ST95], transactions were static in that they were limited to fixed data declared as concurrently accessible and thus permitted to be mutated inside an atomic section.

Today's STMs are fully general with dynamically initiated transactions and arbitrary transactional state, due to advances by Herlihy [HLMS03]. Though Herlihy pioneered effective contention management for transactions to permit higher throughput, the resulting systems manifested order of magnitude overheads, or more, as compared to non-transactional execution.

Unlike conventional closed-nested transactions [Moss82], where the effects of a nested section are invisible to other threads until the top level section completes, open-nested transactional memory as pioneered by Moss [Moss06] permits the effects of a nested section to be made visible as soon as the section commits, and thus, punctuates the top-level transaction, similar to this work. The difference is this thesis work is pessimistic and thus never needs to rollback. Contrast to an open-nested top-level transaction that is optimistic, and hence when the transaction aborts, compensating actions must often be specified by the programmer to undo the effects of a committed nested section, for those operations that can be undone.

For example, open-nested transactional implementations can buffer output in certain circumstances so that it is not immediately visible. Upon a nested transaction abort, the developer can specify a compensating action through a registered abort handler to nullify the

buffered output, thus properly rolling back the transaction. Upon a nested transaction commit, the reads and write sets of the inner transaction are cleared from that of the parent. Thus, a memory conflict does not occur between the parent and child transaction if the parent accessed the memory committed by the child. However, this has implications for the programming model: if an inner transaction needed to abort upon detection of such a conflict - in order to maintain the program's logical semantics - then such a course of action is no longer possible. Therefore, the notion of abstract locks is introduced in the open-nested transactional model for child transactions to acquire. Acquiring such a lock ensures detection of high-level (non-memory) conflicting actions with the parent transaction.

Both closed and open nesting can improve performance since if a nested transaction aborts, it can be rolled back and re-executed without aborting the outer transaction. However, open nesting admits more concurrency than closed nesting due to a greater set of allowable schedules, yet it can exhibit loss of serializability [ALS06] in the most popular implementations if the effects of aborted transactions either remain visible or are allowed to be reified as part of optimistic retry constructs made available to the programmer. Similar to this work, communication among transactions involving waiting within an optimistic framework has most recently been formulated [LM11] as part of a model tracking dependencies among transactions, and either aborting or committing all mutually dependent transactions together, without requiring compensating actions.

Although STMs are more scalable than software utilizing locks, in that they have potentially higher throughput, they have been found to be slower than lock-based concurrent codes in the contended case, when transactions experience frequent aborts due to conflicts. This result, as well as the fact that optimistic transactional systems cannot by their nature roll back irreversible actions, led to the development of pessimistic transactional systems, which by default assume the conflicting case and thus do not roll back.

2.3 Pessimistic Atomic Sections

Pessimistic atomic section implementations typically utilize a collection of static program analyses to infer a set of locks for shared data accessed within atomic sections. These

locksets correspond to the abstract memory locations affected during the execution of the section at run time. The inferred locks ensure an atomic section has exclusive access to update its portion of the program's state; therefore, by definition, its effects cannot conflict with those of another section. Fine-grained lock inference allows for higher levels of concurrency, yet the analysis required is prohibitive to calculate, and it is typically impossible to obtain the necessary precision. Coarser locks increase thread contention of accessed data, preventing otherwise distinct concurrent accesses from being performed, and hence, reducing performance. The original programs are transformed to utilize locking libraries in the implementation to ensure mutual exclusion. Traditional locks from the underlying platform are utilized and lock acquisitions and releases are inserted to ensure atomicity.

Brewer [MZGB06] first proposed annotating shared data that is to feature in an atomic section for transformation into an underlying set of locks. His Autolocker framework could infer a set of lock acquisitions and releases for the program and guarantee a deadlock free ordering of lock acquisitions. Autolocker allows dynamically allocated shared data to be marked, thus ensuring fine-grained instance-based lock inference. Autolocker begins its transformation process by merging all files for a program and extracting the atomic sections. A dependency graph is generated containing all the locks acquired within the sections. Then, a topological sort is performed to obtain a global deadlock free ordering of lock acquisitions. If a deadlock free transform cannot be obtained (due to a cycle in a dependency graph) for the given input, Autolocker signals an error at compile time.

Unfortunately, this approach places an annotation burden on the developer as compared to conventional transactional memory: concurrently accessed variables need to be marked with the locks protecting them. A consequential problem is that the annotations might be incorrect. Nonetheless, the programmer is freed from manually acquiring and releasing locks in order to access certain data.

Subsequent work has improved on Autolocker by eliding this annotation burden placed on the programmer, moving to provide an atomic interface substantially similar to transactional

memory systems. Hicks [HFP06] utilizes static whole program analysis to infer coarse-grained locks directly from the data accessed in atomic sections. A lock is associated with a set of memory locations; acquiring a lock thus guarantees mutual exclusion to a thread entering an atomic section with respect to all variables accessed within the section. Another improvement in their work is to reduce the number of locks needed by merging locks that are always present together in the locksets of atomic sections throughout the program. Depending on the implementation of locks (e.g., kernel versus user locks), lock acquisition and release can be costly in terms of time to perform a system call and time spent in the kernel. Further work by Emmi [EFJM07] attempts to minimize the set of locks needed for the atomic blocks by formulating lock allocation as a Binary Integer Programming (BIP) problem. Obtaining optimal solutions surprisingly did not take an exorbitant amount of time (less than a second for most programs), though transforming the atomic sections into BIP formulations did, taking the better part of an hour in one case.

The work of Cherem [CCG08] infers fine-grained per data structure instance locks by performing a backwards analysis for each heap location accessed within an atomic section. Dereferenced pointer expressions corresponding to heap accesses that are in scope at the beginning of the atomic section are locked. To ensure analysis termination, derived expression locks are inferred up to a specified expression size limit, at which point coarse-grained locks are utilized. Furthermore, a multi-granularity locking library and assignment scheme is utilized such that deadlock is avoided at runtime, for the most precise compile time assignment of locks to date. Cunningham [CGE08] [CDE08] implemented a lock inference framework for Java that supported unbounded atomic section accesses - necessary when accessing the nodes of a recursive structure such as a linked list for example - through a formulation into regular expressions. However, their framework does not prevent deadlocks resulting from inserted locks, but detects them at runtime and attempts to roll back state.

Zhang, [ZSZSG07] [ZSZSG08] and Halpert [HPV07] aim to support the existing concurrent interfaces of OpenMP and Java monitors respectively, without requiring the programmer to supply locking information. Their works cover programs containing such concurrent constructs, though they disregard existing locking information and attempt to

infer the set of locks required while adhering to the respective concurrent interface contracts. They both use a May-Happen-in-Parallel analysis to aid in building an atomic section interference graph. Whereas other works infer the locks for a section from the aliasing information of the data accessed within, these two works utilize the concurrency interference graph to discern conflicting atomic sections and assign the same lock to them. A graph containing nodes representing critical sections has edges between nodes if they interfere in their accesses of a specific variable. They also formulate heuristics to minimize the number of locks allocated.

In order to fully support their respective concurrent interfaces, both of these works also allow condition synchronization, in that a thread inside a critical section or monitor may wait on another thread until a specified condition becomes true. However, Zhang et al. forbid nested sections; while Halpert et al. do support nesting there is no notion of atomicity as it pertains to the enveloping parent sections. Two-phased locking is a locking policy initially utilized in databases to guarantee the serializability of transactions. Two-phased locking mandates a lock acquisition phase followed by a lock release phase; once any locks are released, no locks can be further acquired within a transaction [EGLT76]. Since Halpert et al. only support Java's critical sections semantics, they do not need to implement two-phased locking to guarantee the outermost transaction is atomic.

In contrast to conventional operation-centric transactions, which are essentially blocks of code delimited by transaction begin and end statements, Vaziri [VTD06] attempts to formulate a data-centric synchronization framework that automatically locks sets of data as a byproduct of accessing objects. Developers annotate fields within classes whose objects must be synchronized together as an atomic set, and the compiler infers atomic sections to satisfy these consistency constraints. In essence, the specified higher-level data constraints replace the more voluminous operation-centric synchronized blocks, reducing the chances of data races due to programmer error in properly delimiting transactions.

2.4 Hybrid Transactional Memory

There have been attempts at combinations of pessimistic and optimistic atomic section implementations. Pessimistic atomic section implementations suffer from performance degradation due to conservative static program analysis necessary to prove that the program execution at runtime is faithful to the semantics of transactional memory. Such conservative assumptions do not typically permit as large a number of concurrent transactions to execute in parallel even when they can actually do so at run time without conflict, as implemented in software transactional memory. As a result, pessimistic implementations can take up to a factor of eight times that of optimistic implementations to execute certain microbenchmarks in the high contention configuration [CCG08]. On the other hand, the extra bookkeeping costs in software transactional memory result in significant performance degradation due to the overhead of tracking read and write sets and mutated values in the runtime implementation. This cost is typically ameliorated by hashing the addresses of accessed shared data words to a smaller number of runtime metadata objects, which are utilized to track the read and write sets of transactions. However, this can result in false conflicts, not unlike the approximation inherent in a static analysis of an aliased datum's abstract set of memory locations. Realizing this, Mannarswamy's [MCRS10] work aims to statically infer the mapping of a subset of the shared data within a program to distinct runtime metadata objects. At runtime, the STM implementation (TL2) allocates these mappings to their own metadata objects (ensuring false conflicts are not experienced), and meanwhile maps the rest of the accessed shared data as it otherwise would.

The work of Usui [UBES09] collects runtime statistics on aborts as well as commits and adaptively executes critical sections with either locks or optimistically as transactions. Sections that have experienced high contention are automatically switched to acquiring locks. However, sections have to be marked with programmer annotated coarse-grained locks, which detracts from the composability and deadlock-freedom properties of conventional atomic sections. The work of Dalessandro [DDSSS10] supports a restricted hybrid transactional model in which atomic sections that write are executed pessimistically and

cannot abort, while sections that only perform reads are executed optimistically and concurrently with other such sections, but can still abort.

2.5 Condition Synchronization

Whereas most atomic section implementations concern themselves with atomicity, there have been attempts at supporting condition synchronization mechanisms directly as a first class construct in the atomic section interface. Harris [HF03] adapts C.A.R. Hoare's Conditional Critical Regions (CCR) construct to an atomic section interface for use by an STM implementation. A CCR permits entry into a delimited region of code upon the evaluation of a predicate to true. As adapted by Harris, a predicate is permitted just after the atomic keyword, delineating a conditionally executed atomic section. However, within Harris' work, all conditions within nested atomic sections are effectively evaluated at the top-level transaction (hence, nested atomic sections are flattened into the parent sections), and therefore, no mechanism is provided to communicate among atomic sections. Essentially, any condition evaluating to false aborts the top-level transaction.

The work of Smaragdakis [SKBY07] attempts a rather complex atomic section programming model - with nine additional keywords in total - in support of communication among transactions. Transactions are permitted to observe the effects of other sections and conditionally execute through the use of a wait keyword followed by a predicate that can be placed in the middle of an atomic section. Transactions are executed and automatically commit upon either a wait statement whose condition evaluates to false or encountering an irreversible operation. Such an early commit splits the transaction into a finished part and future transaction that is awaiting execution. The programmer must manually reestablish program invariants upon resumption of the rest of the new transaction once the condition is evaluated to true, or the irreversible operation finishes execution. Every procedure transitively containing such a suspending transaction must be annotated as such. This annotation is to aid the programmer via type system enforced warnings in the model.

This chapter has discussed the history of transactional memory and atomic sections, the quest to make them flexible and expressive in terms of synchronization as compared to traditional locking constructs, while maintaining their ease of use.

Chapter 3
Language Constructs and Semantics

The atomic section implementation consists of syntactic additions to a base language and their accompanying semantic modifications. These constructs are discussed and elaborated upon.

3.1 Syntax

The proposed atomic section interface consists of the addition of two new keywords to the C99 language.

3.1.1 Atomic Sections

The `atomic` keyword specifies an atomic section compound block. The section occurs wherever a statement is allowed, and it may contain an optional parenthesized predicate guard:

```
atomic [(predicate)] {
}
```

An ANTLR grammar [PARR06] for C would incorporate the following rule:

```
atomic_statement
       : 'atomic' (options: '(' expression ')') statement
       ;
```

Where the added statement is part of the statement non-terminal production:

```
statement
       : labeled_statement
       | compound_statement
       | expression_statement
       | selection_statement
       | iteration_statement
       | jump_statement
       | atomic_statement
       ;
```

3.1.2 Sync Variables

The `sync` keyword consists of a declaration qualifier for a static or global variable:

```
sync int var = 0;
```

An ANTLR grammar would incorporate the aforementioned construct as part of the following rule:

```
type_qualifier
       : 'const'
       | 'volatile'
       | 'restrict'
       | 'sync'
       ;
```

Sync variables may only be accessed within the body or guard predicate of an atomic section. Furthermore, a given sync variable cannot be accessed within multiple nested transaction levels along a given path of execution within a transaction. Such a program construction is considered erroneous and is detected at compile time, resulting in a compiler error. The atomic block syntax with optional predicate is similar to that of Harris style conditional critical regions [HF03] as implemented for the Java language. The sync modifier is unique to this design, though in the current implementation sync variables are limited to static and global integers.

3.2 Semantics

3.2.1 Atomic Sections

Execution of an atomic section entails execution of the statements within it. Other threads observe the effects of the executed statements atomically - all at once - after all the statements have completed, except for effects on sync variables. Threads execute atomic sections one at a time when exclusively accessing the same variables. Variables accessed exclusively may only be accessed by one thread at a time.

3.2.2 Atomic Section Nesting without Predicates

A transaction syntactically nested within a parent transaction is executed as an atomic block of statements as part of the outer transaction, which itself is a larger atomic block. Variables accessed within a nested section are automatically accessed exclusively in the parent section. In addition, variable accesses in nested sections contained within conditional statements are also accessed exclusively in the parent transaction, as the implementation is pessimistic. This is the closed nested model of transactions [Moss82], that is an atomic section model where the effects of a nested section are not visible to other threads until the top-level atomic section completes.

3.2.3 Predicates

The definition of an optional predicate expression for an atomic block permits an atomic section to block while waiting for a predicate to evaluate to true before executing the statements within. Predicates are composed of arbitrary side-effect free expressions containing ordinary as well as sync variables. An unsatisfiable predicate due to erroneous program construction results in an indefinitely blocked thread, i.e. synchronization deadlock. Predicates permit condition synchronization of programs utilizing atomic sections.

3.2.4 Sync Variables

The effects of mutations of variables declared with the sync modifier – sync vars – are not observed atomically but rather perforate transactions in that they are observed when the directly accessing atomic section completes. In the execution model - as opposed to the implementation, which is pessimistic by default - the enclosing transaction is normally executed pessimistically when it contains a sync variable. A sync variable within the predicate expression of an atomic section is considered part of the set of variables accessed within the section.

3.2.5 Predicate Nesting

Nested transactions may wait on predicate expressions to become true before executing. However, ordinary non-sync variables accessed within the section are accessed exclusively,

and thus, are not of much use in a predicate expression, whose state must change in order to eventually become true and allow entry to the nested section. Therefore, sync variables must be utilized as part of a predicate expression, and as sync variables are not atomic with respect to the enclosing parent transaction, the resulting predicate expression is able to change state.

An atomic section directly (that is, not transitively) accessing a sync var syntactically present within the section is guaranteed exclusive access to the variable for the duration of the section. Upon termination of the directly accessing section, the sync var is no longer atomic, irrespective of the nesting depth of the section, and any changes performed to it are immediately visible to all threads, unlike ordinary non-sync variables accessed within a nested section.

Thus, the perforation of transactions that sync vars provide permits some measure of limited communication among threads in an otherwise closed nested model of transactions. The earlier aforementioned erroneous program construction from section 3.1.2 results in a compiler error because only one nested level or depth of an atomic section is permitted to directly access a sync var. A nested transaction further accessing a sync var in addition to its parent is nonsensical, as the parent transaction directly accessing the sync var would not be guaranteed an atomic view of it.

Figure 3-1 displays a diagram showing the nesting depth of nested transactions and the exclusivity and subsequent visibility of the accessed ordinary variable c and sync variables a and b within, as specified in this example code block:

```
atomic ( /* a synchronized */ ) {
    atomic ( /* b synchronized */ ) {
          /* c accessed */
    }
    atomic ( /* b synchronized */ ) {
    }
    /* a accessed */
```

```
}
```

Sync variable a is accessed in the top-level section and is exclusive to said section until completion and cannot be accessed in its nested sections. Sync variable b is accessed in the two nested sections and is exclusive to them until they complete and cannot be accessed in the parent section. Ordinary variable c happens to be accessed in the first nested section and is held exclusively for the entirety of the top-level section.

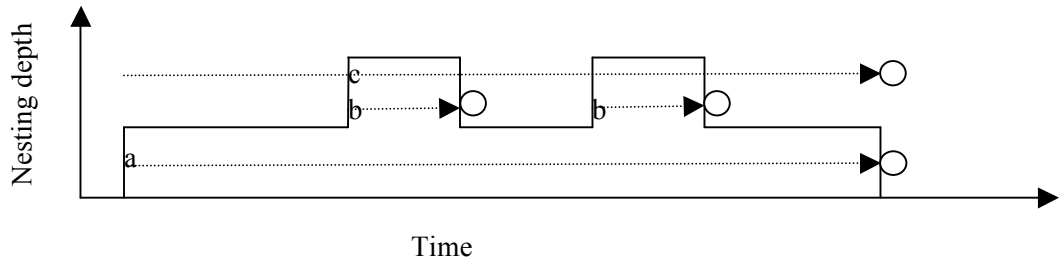

Figure 3-1 Nested transactions accessing sync variables a and b, in addition to ordinary variable c

3.2.6 Example

This is illustrated with an example of a simple thread barrier. Threads reaching a barrier typically each increment a counter and wait, with the last arriving thread permitting all threads to proceed past the barrier when the condition becomes true. Barriers permit multiple threads to synchronize actions in stages. A barrier in the presented transactional model can be implemented as such:

```
void enterbarrier() {
    sync int static count = 0;
    atomic {
          count+=1;
    }
    atomic(count == thread_count) {
          ...
    }
}
```

The barrier code - which may be implemented as part of a library module - may be called and thus composed within the logic of a program already containing an atomic section like so:

```
atomic {
    ...
    enterbarrier();
    ...
}
```

If the counter is not declared as a sync variable then the program threads executing the enclosing atomic section containing the barrier code could not observe the increment of the counter by other threads and the barrier would not function properly. The enclosing section never completes as the nested barrier section waits indefinitely for the counter variable to change.

If an atomic section specification is an otherwise closed nested model though permitting the programmer to have a limited form of communication of mutated values outside of the transaction, then atomic sections can be composable. This is the motivation for the introduction of sync variables in the present work. They are meant to be included in predicate expressions guarding entry to atomic sections and, when mutated inside, their updated values are visible outside of the immediate transaction upon its conclusion.

In the barrier example, assuming the variable named count is a sync variable then its incremented value is visible to the second nested transaction. Two or more atomic sections attempting to directly access a particular sync variable have to do so one at a time, that is, the variable is accessed exclusively by the directly accessing section. Upon exiting an atomic section where a sync variable is accessed, the variable is free to be tested as part of the predicate expression of other waiting transactions, including the next nested transaction in the example given.

As a sync variable is externally visible, the action of mutating it is considered irreversible and thus a transaction undertaking such an operation must be executed pessimistically. In this atomic section model, transactions containing external operations must be executed

pessimistically. While this simplifies the programming model for the developer, it has certain important implications.

3.2.7 Subtlety of Sync Variables and Nesting

It is generally considered ill-advised from a performance perspective to prevent independent concurrent operations from proceeding while a thread blocks on an operation. Yet that is what pessimistic transactions containing deeply nested waiting transactions can do to other threads. This is not a formidable obstacle as the depth of nesting in typical transactional code has been measured to be low [CCMM06].

A further implication for the model is that certain program constructions can suffer from a situation akin to the nested monitor lockout problem [List77] found in concurrent monitor based code. This problem is manifested when a thread holding a resource(s) is waiting to be signaled by another thread which requires the resource(s) to signal the waiting thread. This is distinct from a deadlock condition in that the threads in question may well have acquired the resource(s) according to a common total order, though it still results in a program not making forward progress. Nested monitor lockout may be resolved through a change in the affected code to remove the problem. It is important to note that concurrent languages featuring monitors such as Java and uC++ have chosen to allow for the possibility of the nested monitor lockout problem to occur as an alternative to the programmer having to manually establish program invariants upon monitor entry, which requires global program reasoning. Developer diligence and awareness of this issue is a requirement in this atomic section model.

Chapter 4
Implementation and Design Decisions

Overall the system assigns locks to atomic sections so that when they access the same variable, the sections are assigned the same lock, which guarantees exclusive access to the variable at runtime. Determining the set of variables accessed - and thus memory locations accessed - is a necessary prerequisite before locks are mapped to sections. Thus, tallying the set of memory objects accessed within each routine in a program's call graph and deducing the set of corresponding locks needed is the task of the analysis performed within the implemented system. The implementation details, including modifications to the front end and analysis and transformation phases of the augmented compiler infrastructure, are elaborated.

4.1 LLVM

The added constructs were implemented for the C99 language as supported in the Low Level Virtual Machine (LLVM) 2.6 compiler infrastructure [LA04] augmented with the Data Structure Analysis (DSA) module [LLA07]. LLVM is a relatively new compiler framework and typed intermediate representation based on static single-assignment form [CFRWZ91]. Its modern modular design facilitates new whole program optimizations and robust clean extensions for research and experimentation. Note optimizations applied across the entire program are possible at link time where code exists for both the application program and any libraries utilized. New in the 2.6 version of LLVM is a modular front end, clang, to natively parse many C-like languages. LLVM's clang front end was modified to accept the new constructs.

4.1.1 DSA

DSA is a fast mostly O(n*log(n)) time complexity points-to analysis. It is mostly context sensitive in that the calling procedure context is taken into account in the resulting analysis in order to yield further precision.

DSA calculates a data structure graph (DSG) for each function in the input program such that distinct nodes represent disjoint sets of dynamic memory objects and edges correspond to pointers from the fields of the memory objects (nodes) to other such objects. A DSG may also contain call nodes to other such graphs representing calls in the program control-flow graph. The structures provided by DSA can readily be used to ascertain whether two pointers within the same function (corresponding to a DSG) may alias. DSA does not directly support aliasing queries - queries determining whether the set of memory objects pointed-to by a pair of pointers intersect - involving pointers corresponding to nodes in different functions, though DSA provides the aforementioned structures and relations that can be processed to determine whether functions accessing data through a pointer alias the same memory object. Determining whether sets of accesses alias is necessary in order to guarantee independence of statements and their runtime effects.

DSA operates in three phases: local, bottom up, and top down. In the first stage of DSA, a DSG is created for each function in the input program using only intraprocedural information. The second ("bottom-up" postorder traversal) stage incorporates information from the callee DSGs into the caller DSGs by cloning the former into the latter, eliminating incomplete information due to call nodes in a function, and thus, completing the construction of the call graph. The third ("top-down" reverse postorder traversal) stage merges the caller DSGs into each of their callee DSGs.

After the third stage, two distinct functions both calling a third have the memory object nodes corresponding to their arguments merged together into the called function's DSG, losing context sensitivity. Context sensitivity is also lost in self and mutually recursive procedures. Maintaining context sensitivity during this stage by splitting the called function's DSG for each distinct callsite significantly expands the memory utilization of DSA, and thus, splitting is not performed. After DSA finishes, complete information is attained for every input function's DSG and nodes contained therein, except for memory objects that may be accessed by code external to the analyzed input program.

4.2 Analysis Pass

This work features a lock inference analysis that, broadly speaking, forms global equivalence classes of memory objects, assigns said classes for each atomic section in each function corresponding to memory transitively accessed through contained called functions, and forms equivalence classes for encountered sync variables.

The analysis pass uses the results of the top-down DSA phase to generate the points-to sets. The analysis first matches the DSG nodes of the arguments and formal parameters across function calls in different DSGs corresponding to the caller and callee. The matched nodes across the functions of the input program are subsequently put into global equivalence classes (ECs). This step facilitates determining whether two pointers across the whole input program may alias by simply checking if their corresponding matched nodes are placed in the same equivalence class.

After forming the global equivalence classes, the lock inference analysis pass collects the ECs corresponding to accesses at every program point in order to tally the set of equivalence classes of memory locations accessed inside each function. Input programs' call graphs are then cleaved into strongly connected components (SCCs). The SCCs of the call graph are traversed bottom up in order to union the ECs of the callees up into the calling functions. This step is performed so the ECs corresponding to the functions that are transitively called inside any atomic section are accounted for.

A non-trivial SCC containing more than one procedure assigns the ECs of the atomic sections inside each procedure within the tally of all the ECs corresponding to all the memory locations accessed by any function within the SCC, thus losing precision, but efficiently handling recursive calls. Atomic sections traversed during the pass are annotated with the ECs of the accesses they guard.

The sync variables (enforced statically not to be aliased) encountered during this phase are placed into their own equivalence classes - one variable per class. Like other memory objects, their accesses inside atomic sections need to be tallied. Sync variables are not aliased with themselves or any other, therefore separate transactions accessing sync variables

– while serialized – still permit concurrency within distinct parent transactions. Therefore in the following example, while the first thread's parent atomic section is executing in the nested section within the barrier routine, the second thread's parent atomic section may concurrently be executing just before or just after its own call to the barrier routine:

```
atomic {  // Thread 1 parent atomic section
    enterbarrier();  // currently executing within nested section
}
atomic {  // Thread 2 parent atomic section
    // May be executing just before call to nested atomic section
    enterbarrier();
    // May be executing just after call to nested atomic section
}
```

However, if a developer were to later insert and access a common variable within the parent transactions then concurrency is no longer possible. Sync variables are currently limited to static and global integers and may not have their address taken in order to ensure disjointedness with other memory objects due to the necessarily conservative nature of the analysis.

More formidable analyses exist than presented here that attempt to discern locksets for path expressions [EFJM07] rather than abstract memory locations, or alternatively, expression locks for any program point. The analysis by Cherem at al. utilizing expression locks is able to reason regarding recursive structure accesses up to a specified limit, and can utilize fine-grained locks to protect per instance allocated structures. Yet the resulting additional benefits from more thorough analyses have been found to be miniscule [CCG08], thus the analysis performed in this work is considered to be suitable.

4.2.1 Transformations

Nested atomic sections are traversed top down in the call graph to insert lock/unlock code of inferred locks in a total order, according to a two-phase locking discipline [EGLT76].

Since each EC corresponds to a lock of abstract memory locations, the set of all such locations must be accessed exclusively upon entering an atomic section. Therefore, each lock set is reified into a Pthreads mutex.

For the transformation, all the ECs are sorted into an arbitrary total order such that deadlock at runtime is prevented. A classic two-phase locking (2PL) policy is implemented to ensure atomicity for the overall (arbitrary nested) atomic section. As an optimization, late locking is performed such that a lock is delayed from being acquired (not necessarily at the beginning of a top-level atomic section) until the beginning of the nested section where the first access is performed corresponding to the lock.

The order of accesses performed at runtime is conservatively approximated in the SCC of the call graph through the intersection of a given function's directly accessed memory objects with the result of the union of its accessed objects and its called functions' accessed objects. Any memory objects in the intersection are necessarily accessed after the start of said function. Calls to Pthreads mutex unlock routines are inserted such that all locks corresponding to ECs are released at the end of the top-level atomic section. The SCCs of the call graph are then traversed top down through the atomic sections. In this fashion, nested atomic sections across functions are evaluated from outer section to inner section. A diagram follows showing late locking employed with three variables a, b, and c accessed from SCC nodes alpha and beta. Though the SCC node alpha transitively accesses variables b and c, locks on them are not acquired upon entry to this SCC node, but upon entry to the first node where they are accessed, SCC node beta.

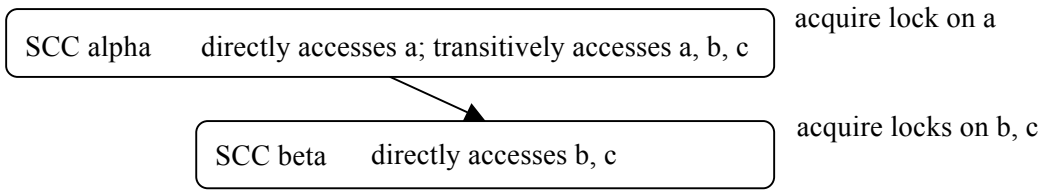

Figure 4-1 Late locking with variables a, b, and c accessed from SCC nodes alpha and beta

Every atomic section encountered is transformed such that the ECs directly guarded within are represented as locks that must be held upon entry. The locks representing ECs directly guarded in the section are a subset of the ECs guarded by the section transitively. A global array of Pthreads mutex structures corresponding to the ECs of the program is inserted into the program text such that the Pthreads mutex calls access the necessary structures at runtime.

Given that the locks representing ECs are acquired in a total order, delayed locking may be a vacuous optimization if locks which may safely be acquired later in the call graph happen to be ordered before locks which are necessary to be acquired earlier in the call graph, and therefore, the former end up being acquired prior to the latter. This approach guarantees each atomic section acquires a (super) set of the locks necessary to protect data accesses transitively performed from within. Furthermore, the fact that each atomic section nominally attempts to acquire only the locks protecting its direct accesses – late locking – allows for more potential concurrency to be exhibited. However all locks previously acquired are still released at the end of the top-level atomic section. This is in contrast to some implementations that acquire all locks transitively required at the beginning of the top-level atomic section and release them all at its end. A simple heuristic combines multiple Pthreads locks always acquired together inside atomic sections into a single lock. This step reduces the lock acquisition overhead of the underlying library.

4.2.2 Sync Variables Implementation

Given the semantics of sync variables, different implementations are possible. For example, continuous polling of the variable(s) associated with an atomic section's predicate may be used. Alternatively, a change in any of the variables results in a notification and reevaluation of the predicate. In the current implementation, the latter approach is chosen as the former method is considered to unnecessarily penalize threads that wait. Notification is implemented using Pthreads condition variables (which also require an associated lock), and is triggered upon the exit of an atomic section that modifies a sync variable. Upon notification, an attempt is made to acquire the lock(s) of the sync variable(s) within a

predicate expression. If the predicate evaluates to true, the associated atomic section is executed. Otherwise the lock(s) of the sync variable(s) are released and the predicate is retested at a later time, upon further notification. One limitation of Pthreads is that a thread may wait on only one condition variable at a time. Thus, the current implementation only allows one sync variable within a predicate.

4.2.3 Possible Analysis Optimization

A given total order imposed on lock acquisitions may penalize certain program executions. A lock corresponding to a memory location access within a deeply nested atomic section may be ordered early in a total order. Thus, it might have to be acquired early within the overall atomic section of a given thread's execution, preventing other threads from accessing it within their own atomic sections if the other accesses within their sections do not conflict with said thread. In the current implementation, the chosen order is arbitrary. A different approach to ordering could utilize acquired statistics from program runs to attempt to order the locks in a suitable arrangement so as to improve performance. Information collected on program execution determines the atomic sections and locks that are most frequently executed or waited upon.

A further transformation may then make use of this information such that distinct overall atomic sections containing nested sections experiencing heavy contention over a small intersecting set of locks would have their ordering changed such that entry into an overall atomic section does not acquire a lock whose corresponding memory locations are accessed late within a deeply nested section that are contended by other overall atomic sections. LLVM does not account for threads in its relatively immature profile guided information collection and optimization infrastructure however; therefore, this technique was not attempted.

4.2.4 Transformation Example Featuring Sync Variable

Given the discussion of the analysis and transformation of the implementation, a running example is presented featuring condition synchronization among threads as introduced with barriers and featuring a sync variable, and the step by step transformation of the program as it

is run through the implemented system. Some of the automatically generated code and outputs resulting from the transformations are elided, as they do not pertain to the discussion.

The example program creates NUM_THREADS threads that are then run with the same function containing a call to a barrier within it. Note, the preprocessor variable NUM_THREADS is changed to the appropriate number of processors between configuration runs. All threads must synchronize with the barrier after executing the first half of the function, before they can proceed to the second half. Upon finishing execution of the function, the threads are terminated by being joined with the main program thread.

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 8

void simplebarrier()
{
  sync static int count = 0;
  atomic {
    count++;
  }
  atomic (count == NUM_THREADS) {
  }
}

void *TaskCode(void *argument)
{
  int tid = *((int *) argument);
  for(volatile int i=0;i<100000000;i++) ; // work delay
  // printf("Thread %d completing first half of task.\n", tid);
  simplebarrier();
  for(volatile int i=0;i<100000000;i++) ; // work delay
  // printf("Thread %d completing second half of task.\n", tid);
```

```
}

int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];
  int rc;

  /* create all threads */
  for (int i=0; i<NUM_THREADS; ++i) {
    thread_args[i] = i;
    // printf("creating thread %d\n", i);
    rc = pthread_create(&threads[i], (void *) NULL, TaskCode,
(void *) &thread_args[i]);
  }

  /* wait for all threads to complete */
  for (int i=0; i<NUM_THREADS; ++i) {
    // printf("joining thread %d\n", i);
    rc = pthread_join(threads[i], NULL);
  }
}
```

The analysis phase first tallies the points-to Equivalence Classes (ECs) for each function in the input program from the results of the post processing of the Data Structure Analysis (DSA) pass. Thus, the ECs for the three procedures of the extended example are:

{sync.simplebarrier.count} for the simplebarrier procedure that reflects the sync variable accessed in its atomic sections, {} for the TaskCode procedure, and {} for the main procedure.

The call graph for the input program is main -> TaskCode -> simplebarrier. Therefore, a postorder traversal of the SCCs of the callgraph of the input program yields the following ECs for the three procedure of the extended example:

{sync.simplebarrier.count} for the simplebarrier, {} for the TaskCode procedure, and {} for the main procedure.

Note that sync variables are not propagated up into the calling procedures. The transformation phase begins by a reverse postorder traversal of the SCCs of the callgraph and inserts lock/unlock code for the atomic sections in the simplebarrier procedure for the Pthreads mutex associated with the accessed sync variable. Therefore, the LLVM specific basic block disassembly output for the entry to the first atomic section is transformed to insert locking code, from:

```
  atomic.begin:                                     ; preds = %entry
    br i1 true, label %atomic.body, label %atomic.end
```

to:

```
  atomic.begin:                                     ; preds = %entry
    %sync.simplebarrier.count = call i32 @pthread_mutex_lock([40 x
i8]* @0) ; <i32> [#uses=0]
    br i1 true, label %atomic.body, label %atomic.end
```

In generic LLVM assembly, the branch mnemonic as shown features the condition as the first operand (in this instance it is the constant true) and the second operand as the basic block to branch to upon a true condition (which is what is taken in this instance), and the third basic block to branch upon a false condition.

The exit from the atomic section is transformed to insert unlocking code, from:

```
  atomic.end:             ; preds = %atomic.body, %atomic.begin
    br label %atomic.begin1
```

to:

```
  atomic.end:             ; preds = %atomic.body, %atomic.begin
    %sync.simplebarrier.count1 = call i32
@pthread_cond_broadcast([48 x i8]* @"0") ; <i32> [#uses=0]
    %sync.simplebarrier.count2 = call i32 @pthread_mutex_unlock([40
x i8]* @0) ; <i32> [#uses=0]
    br label %atomic.begin1
```

Note the inserted call to the Pthreads specific sync variable mapped condition variable broadcast call. The second atomic section entry is likewise transformed from:

```
atomic.begin1:                                    ; preds = %atomic.end
  %tmp2 = load i32* @sync.simplebarrier.count ; <i32> [#uses=1]
  %cmp = icmp eq i32 %tmp2, 5                     ; <i1> [#uses=1]
  br i1 %cmp, label %atomic.body3, label %atomic.end4
```

to:

```
atomic.begin1:                                    ; preds = %atomic.end
  %sync.simplebarrier.count3 = call i32 @pthread_mutex_lock([40 x
i8]* @0) ; <i32> [#uses=0]
  %tmp2 = load i32* @sync.simplebarrier.count ; <i32> [#uses=1]
  %cmp = icmp eq i32 %tmp2, 5                     ; <i1> [#uses=1]
  br i1 %cmp, label %atomic.body3, label %atomic.end4
```

Note the test for entry into the atomic section if the condition is satisfied. In the next step, the transformation phase splits the aforementioned basic block into a first block that locks the generated mutex associated with the sync variable:

```
atomic.begin1:                                    ; preds = %atomic.end
  %sync.simplebarrier.count3 = call i32 @pthread_mutex_lock([40 x
i8]* @0) ; <i32> [#uses=0]
  br label %atomic.cond
```

And a second block that tests if the condition is satisfied:

```
atomic.cond:                ; preds = %atomic.cv, %atomic.begin1
  %tmp2 = load i32* @sync.simplebarrier.count ; <i32> [#uses=1]
  %cmp = icmp eq i32 %tmp2, 5                     ; <i1> [#uses=1]
  br i1 %cmp, label %atomic.body3, label %atomic.cv
```

If the condition is satisfied, the thread may proceed into the body of the atomic section. However, if the condition is not satisfied the thread is directed to a newly generated block that calls the Pthreads specific sync variable mapped condition variable wait call:

```
atomic.cv:                                       ; preds = %atomic.cond
  %sync.simplebarrier.count5 = call i32 @pthread_cond_wait([48 x
i8]* @"0", [40 x i8]* @0) ; <i32> [#uses=0]
```

```
  br label %atomic.cond
```

Upon being woken up, the thread is directed to the second of the split block outlined earlier to test the predicate condition again. The specification for the Pthreads API is that the lock associated with the condition variable is atomically released upon a thread waiting, and reacquired upon waking up after being signaled, respectively. The exit of the atomic section is transformed to insert unlocking code, from:

```
atomic.end4:          ; preds = %atomic.body3, %atomic.begin1
  ret void
```

to:

```
atomic.end4:          ; preds = %atomic.body3, %atomic.begin1
  %sync.simplebarrier.count4 = call i32 @pthread_mutex_unlock([40
x i8]* @0) ; <i32> [#uses=0]
  ret void
```

The sync variable is included in the text of the program:

```
@sync.simplebarrier.count = internal global i32 0 ; <i32*>
[#uses=3]
```

And the generated Pthreads mutex lock and condition variable are also included in the program:

```
@0 = global [40 x i8] zeroinitializer   ; <[40 x i8]*> [#uses=5]
@"0" = global [48 x i8] zeroinitializer ; <[48 x i8]*> [#uses=2]
```

4.2.5 Transformation Featuring Inferred Locks and Sync Variable

The previous running transformation example was rather simple, as the framework did not need to infer locks. In this subsection, an implementation of the producer/consumer pattern is presented which utilizes the generic shared queue insertion and removal operations illustrated in section 5.2 as examples of the condition synchronization facility possible with the model. The intermediate representation of the program is transformed, step-by-step, as it is run through the implemented system. Some of the automatically generated code and outputs resulting from the transformations are elided, as they do not pertain to the discussion.

The extended example creates producer and consumer threads each executing their own respective actions of putting and taking items from a shared queue. A fixed number of items are produced, so the amount of work is fixed regardless of the number of threads. The underlying queue code is generic, utilizing pointers to heap allocated elements. Thus, this implementation requires locks to be inferred to protect the shared memory structure from inadvertent accesses. The put and get procedures also utilize a size sync variable to ascertain and mutate the number of elements contained in the queue. All threads must synchronize with the queue such that only one thread may put or take items from the queue. Upon finishing execution of their respective tasks, the producer and consumer threads are terminated by being joined with the main program thread. As in the previous example, the preprocessor variable `NUM_THREADS` is eight in this example but is changed to the appropriate number of processors between configuration runs.

```
#include <pthread.h>
#include <stdio.h>

typedef int TItem;
#define NUM_THREADS 8
#define NUM_ITEMS 1048576

void *prod(void *arg)
{
  int tid = *((int *) arg);
  for(int i=0;i<(NUM_ITEMS/NUM_THREADS);i++) {
    TItem* elp=(TItem*)malloc(sizeof(TItem));
    *elp=i*NUM_THREADS+tid;
    put(elp);
  }
}

void *cons(void *arg)
{
```

```
  int tid = *((int *) arg);
  for(int i=0;i<(NUM_ITEMS/NUM_THREADS);i++) {
    TItem* elp=get();
    free((void*)elp);
  }
}

void *prodcons(void *arg)
{
  int tid = *((int *) arg);
  for(int i=0;i<(NUM_ITEMS/NUM_THREADS);i++) {
    TItem* elp=(TItem*)malloc(sizeof(TItem));
    *elp=i*NUM_THREADS+tid;
    put(elp);
    elp=get();
    free((void*)elp);
  }
}
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];
  int rc;

  /* create all threads */
  if (NUM_THREADS==1) {
    thread_args[0] = 0;
    rc = pthread_create(&threads[0], (void *) NULL, prodcons,
(void *) &thread_args[0]);
  }
  else
  for (int i=0; i<NUM_THREADS; ++i) {
    thread_args[i] = i;
```

```
      if (i % 2) {
        rc = pthread_create(&threads[i], (void *) NULL, cons, (void
*) &thread_args[i]);
      }
      else {
        rc = pthread_create(&threads[i], (void *) NULL, prod, (void
*) &thread_args[i]);
      }
    }

    /* wait for all threads to complete */
    for (int i=0; i<NUM_THREADS; ++i) {
      rc = pthread_join(threads[i], NULL);
    }
  }
```

The analysis phase first tallies the points-to Equivalence Classes (ECs) for each function in the input program from the results of the post processing of the Data Structure Analysis (DSA) pass. Thus, the ECs for the six procedures of this extended example are:

{sync.num_entries, 0xfecb18, 0xff28a8, 0x103b7b0, 0x103b8c0} for the get procedure that reflects four disjoint equivalence classes of memory locations identified and the sync variable sync.num_entries accessed in its atomic section, {sync.num_entries, 0xfecb18, 0xff2818, 0x103b7b0, 0x103b8c0} for the put procedure with the first element representing the sync variable accessed in its atomic section and the last four reflecting the distinct equivalence classes of memory locations identified, {0x103b8c0} for the prod procedure, {0x103b8c0} for the cons procedure, {0x103b8c0} for the prodcons procedure, and {} for the main procedure. The get and put procedures share many of the equivalence classes as they access the many of the same abstract locations of memory objects.

The call graph for the input program is main -> prod -> put, main -> cons -> put, main -> prodcons, prodcons -> put, prodcons -> get. Therefore, a postorder traversal of the SCCs of the callgraph of the input program yields the following ECs for the six procedure of the extended example:

{sync.num_entries, 0xfecb18, 0xff28a8, 0x103b7b0, 0x103b8c0} for the get procedure, {sync.num_entries, 0xfecb18, 0xff2818, 0x103b7b0, 0x103b8c0} for the put procedure, {0xfecb18, 0xff2818, 0x103b7b0, 0x103b8c0} for the prod procedure, {0xfecb18, 0xff28a8, 0x103b7b0, 0x103b8c0} for the cons procedure, {0xfecb18, 0xff2818, 0xff28a8, 0x103b7b0, 0x103b8c0} for the prodcons procedure, and {0xfecb18, 0xff2818, 0xff28a8, 0x103b7b0, 0x103b8c0} for the main procedure.

The transformation phase begins by a reverse postorder traversal of the SCCs of the callgraph and inserts lock/unlock code for the atomic sections in the put and get procedures for the Pthreads mutex associated with the accessed sync variable, as well as the four inferred locks corresponding to the four equivalence classes of identified memory locations. Therefore, the LLVM specific basic block disassembly output for the entry to the atomic section in the get procedure is transformed to insert locking code, from:

```
  atomic.begin:                                     ; preds = %entry
    %tmp = load i32* @sync.num_entries     ; <i32> [#uses=1]
    %cmp = icmp sgt i32 %tmp, 0                 ; <i1> [#uses=1]
    br i1 %cmp, label %atomic.body, label %atomic.end
```

to:

```
  atomic.begin:                                     ; preds = %entry
    %sync.num_entries = call i32 @pthread_mutex_lock([40 x i8]* @0)
; <i32> [#uses=0]
    %"1046a10" = call i32 @pthread_mutex_lock([40 x i8]* @1) ; <i32>
[#uses=0]
    %"1049d60" = call i32 @pthread_mutex_lock([40 x i8]* @2) ; <i32>
[#uses=0]
    %"1046d60" = call i32 @pthread_mutex_lock([40 x i8]* @3) ; <i32>
[#uses=0]
    %"1046d90" = call i32 @pthread_mutex_lock([40 x i8]* @4) ; <i32>
[#uses=0]
    %tmp = load i32* @sync.num_entries     ; <i32> [#uses=1]
    %cmp = icmp sgt i32 %tmp, 0                 ; <i1> [#uses=1]
    br i1 %cmp, label %atomic.body, label %atomic.end
```

Note the test for entry into the atomic section if the condition is satisfied. The entry to the transformed atomic section contains five locks being acquired. One of which is the lock corresponding to the sync variable, the rest are locks representing the equivalence classes.

In the next step, the transformation phase splits the aforementioned basic block into a first block that locks the generated mutex associated with the sync variable as well as the inferred locks:

```
  atomic.begin:                                     ; preds = %entry
    %sync.num_entries = call i32 @pthread_mutex_lock([40 x i8]* @0)
; <i32> [#uses=0]
    %"1046a10" = call i32 @pthread_mutex_lock([40 x i8]* @1) ; <i32>
[#uses=0]
    %"1049d60" = call i32 @pthread_mutex_lock([40 x i8]* @2) ; <i32>
[#uses=0]
    %"1046d60" = call i32 @pthread_mutex_lock([40 x i8]* @3) ; <i32>
[#uses=0]
    %"1046d90" = call i32 @pthread_mutex_lock([40 x i8]* @4) ; <i32>
[#uses=0]
    br label %atomic.cond
```

And a second block that tests if the condition is satisfied:

```
  atomic.cond:                ; preds = %atomic.cv, %atomic.begin
    %tmp = load i32* @sync.num_entries     ; <i32> [#uses=1]
    %cmp = icmp sgt i32 %tmp, 0                 ; <i1> [#uses=1]
    br i1 %cmp, label %atomic.body, label %atomic.cv
```

If the condition is satisfied, the thread may proceed into the body of the atomic section. However, if the condition is not satisfied the thread is directed to a newly generated block that unlocks the inferred locks, calls the Pthreads specific sync variable mapped condition variable wait call, and then locks the inferred locks again (upon being woken up):

```
  atomic.cv:                                  ; preds = %atomic.cond
    %"1046d911" = call i32 @pthread_mutex_unlock([40 x i8]* @4) ;
<i32> [#uses=0]
    %"1046d612" = call i32 @pthread_mutex_unlock([40 x i8]* @3) ;
<i32> [#uses=0]
```

```
    %"1049d613" = call i32 @pthread_mutex_unlock([40 x i8]* @2) ;
<i32> [#uses=0]
    %"1046a114" = call i32 @pthread_mutex_unlock([40 x i8]* @1) ;
<i32> [#uses=0]
    %sync.num_entries7 = call i32 @pthread_cond_wait([48 x i8]*
@"0", [40 x i8]* @0) ; <i32> [#uses=0]
    %"1046a11" = call i32 @pthread_mutex_lock([40 x i8]* @1) ; <i32>
[#uses=0]
    %"1049d61" = call i32 @pthread_mutex_lock([40 x i8]* @2) ; <i32>
[#uses=0]
    %"1046d61" = call i32 @pthread_mutex_lock([40 x i8]* @3) ; <i32>
[#uses=0]
    %"1046d91" = call i32 @pthread_mutex_lock([40 x i8]* @4) ; <i32>
[#uses=0]
    br label %atomic.cond
```

The exit from the atomic section is transformed to insert unlocking code, from:

```
  atomic.end:             ; preds = %atomic.body, %atomic.begin
    %tmp5 = load i32** %item                  ; <i32*> [#uses=1]
    store i32* %tmp5, i32** %retval
    %0 = load i32** %retval                   ; <i32*> [#uses=1]
    ret i32* %0
```

to:

```
  atomic.end:             ; preds = %atomic.body, %atomic.begin
    %"1046d901" = call i32 @pthread_mutex_unlock([40 x i8]* @4) ;
<i32> [#uses=0]
    %"1046d602" = call i32 @pthread_mutex_unlock([40 x i8]* @3) ;
<i32> [#uses=0]
    %"1049d603" = call i32 @pthread_mutex_unlock([40 x i8]* @2) ;
<i32> [#uses=0]
    %"1046a104" = call i32 @pthread_mutex_unlock([40 x i8]* @1) ;
<i32> [#uses=0]
    %sync.num_entries5 = call i32 @pthread_cond_broadcast([48 x i8]*
@"0") ; <i32> [#uses=0]
    %sync.num_entries6 = call i32 @pthread_mutex_unlock([40 x i8]*
@0) ; <i32> [#uses=0]
    %tmp5 = load i32** %item                  ; <i32*> [#uses=1]
```

```
  store i32* %tmp5, i32** %retval
  %0 = load i32** %retval                    ; <i32*> [#uses=1]
  ret i32* %0
```

Note the inserted call to the Pthreads-specific condition variable broadcast call that is mapped to the sync variable.

The sync variable is included in the text of the program:

```
  @sync.num_entries = global i32 0, align 4         ; <i32*>
[#uses=6]
```

And the generated Pthreads mutex locks and condition variable are also included in the program:

```
  @0 = global [40 x i8] zeroinitializer   ; <[40 x i8]*> [#uses=6]
  @1 = global [40 x i8] zeroinitializer   ; <[40 x i8]*> [#uses=4]
  @2 = global [40 x i8] zeroinitializer   ; <[40 x i8]*> [#uses=4]
  @3 = global [40 x i8] zeroinitializer   ; <[40 x i8]*> [#uses=4]
  @4 = global [40 x i8] zeroinitializer   ; <[40 x i8]*> [#uses=4]
  @"0" = global [48 x i8] zeroinitializer ; <[48 x i8]*> [#uses=4]
```

The transformation steps for the put procedure and resulting intermediate representations and inferred locks are very similar.

Chapter 5
Evaluation

The evaluation performed shows the suitability of the developed atomic section programming model and implementation for constructing robust, performant concurrent programs. The model is shown to provide increased programmability and expressiveness when compared to conventional STM interfaces.

5.1 Quantitative Results

Quantitative results are presented for two programs illustrating common concurrency patterns including producers/consumers and thread barriers.

5.1.1 Experimental Methodology

Benchmark configurations include: the particular benchmark programs; the implementation framework the programs are run under; the choice of one, two, four, or eight processors; and possibly data sets and settings such as the level of contention. All configuration instances were run five times, the order of runs randomized with other configurations, with the average of the five runs per configuration recorded in the graphs. All programs were run on an eight processor four socket dual-core Linux machine with 16 GB of RAM.

5.1.2 Microbenchmark Evaluations

Microbenchmark evaluation results are presented for the producer/consumer pattern of code discussed in the transformation subsection 4.2.5, utilizing the condition synchronization construct involving the put and get shared queue operations from section 5.2. Figure 5-1 displays the average runtimes in seconds for the evaluated configurations of up to eight processors. It shows that though runtime decreases with up to four processors, it increases with eight, due to overhead.

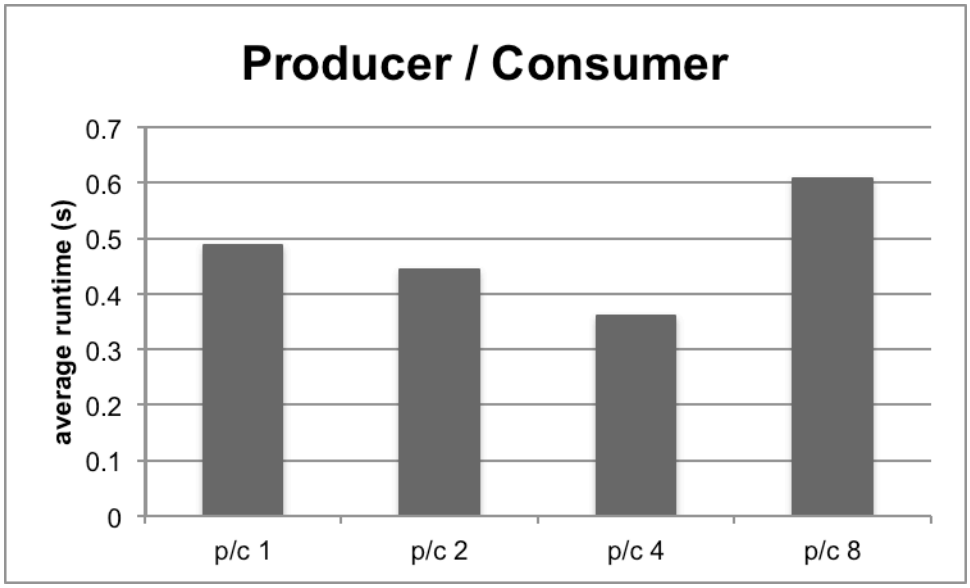

Figure 5-1 Producer / consumer PTM implementation (lower is better)

The thread barrier example in the transformation subsection 4.2.4 was executed as a microbenchmark on up to eight processors. Figure 5-2 displays the average runtimes in seconds for the evaluated configurations. The same amount of work is performed by each thread regardless of configuration since even though the work delays are executed in parallel, they are the same for all threads so the times cannot decrease. The small amount of contention that exists - due to the barrier - increases with more threads, along with the variability of scheduling inherent with greater numbers of threads, the displayed runtimes thus increase with more threads attempting to enter the barrier.

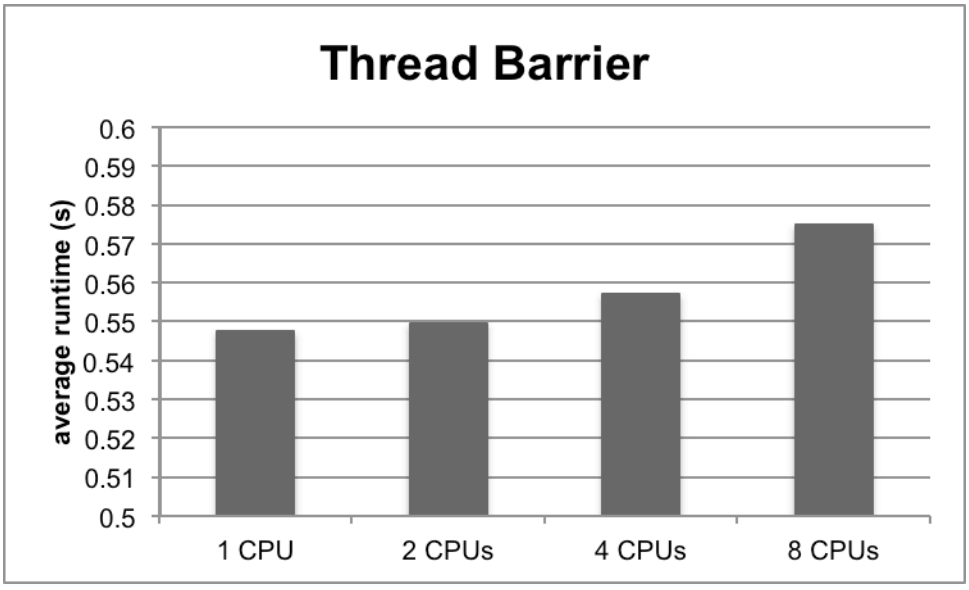

Figure 5-2 Thread barrier PTM implementation (lower is better)

5.2 Qualitative Examples

Qualitative results from the limited experience implementing programs in this model have thus far been positive. The new approach preserves the benefits of traditional transactional semantics, while permitting disciplined multithreaded cooperation.

Thread coordination is essential in concurrent applications. One class of thread coordination problems concerns producer/consumer patterns, which are prevalent in multithreaded applications. A producer/consumer pattern involves a producer thread (or multiple threads) creating a unit of work at a variable - and usually different - rate than the unit of work can be processed by a consuming thread (or multiple threads.) The varying rate in processing items of work in a chain of processing steps can sometimes be explained by the different speeds and latencies of particular levels of the memory hierarchy.

Processing a unit of work defined in terms of a memory frame versus a disk block versus a network packet can require processing times that differ by an order of magnitude or more, depending on the latency of the devices in question. This difference is one of the motivations for the producer and consumer threads being decoupled as part of the producer/consumer pattern. Another motivation for the loose coupling of item processing may be the priority of

different threads coordinating operations. A user interface thread is latency sensitive and may hand off further processing of an identified work item to a lower priority consuming thread.

An example of a generic producer/consumer design utilizing the transactional model follows. In it, the `sync` variable `num_entries` is incremented by the producer thread placing a work item into the shared buffer. The producer waits at the start of the transaction if the queue is full. The in and out index variables correspond to the ends of the buffer the producer and consumer place and take items, respectively, and are modulo incremented. The consumer thread decrements `num_entries` upon taking a work item from the queue. If upon entry to the atomic section the queue is tested to be empty, the consumer waits for an item to be placed into the queue by the producer. The type of the items placed into the queue is left for the implementation to define.

```
sync int num_entries = 0;
#define MAX_ENTRIES 1048577
TItem* buf[MAX_ENTRIES];
int in = 0, out = 0;
void put(TItem* item) {
    atomic (num_entries < MAX_ENTRIES) {
          buf[in] = item;
          in = (in + 1) % MAX_ENTRIES;
          num_entries++;
    }
}
TItem* get(void) {
    TItem* item;
    atomic (num_entries > 0) {
          item = buf[out];
          out = (out + 1) % MAX_ENTRIES;
          num_entries--;
    }
    return item;
```

```
}
```

The details of the producer and consumer thread implementations are highly specific to the operation objective and are customized depending on the application. Note that the processing of the item and the handoff with the shared buffer may well be small details as part of an overall larger atomic operation according to the logic of the application.

Examples of producer/consumer patterns invoked as part of program library-based solutions that can be composed with concurrent application transactions include file copy, application logging, document search, plugin filters, and web server data aggregation and statistics.

Threads orchestrating the copying of files and directories can decouple the process of locating individual files (according to a specified criterion) from actually copying them. A program performing this operation as part of a transaction allows the file locator and file copying threaded codes (which may be factored into their own callable module) to coordinate with each other. In this instance, the file locator is higher priority than the file copying thread as it is identifying files to be copied for the file copier to start its operation.

An application logger module can asynchronously accept input messages while the application it is composed with conducts its own operations, which may be performed with transactions, and the log processor thread can write the log entry messages to a permanent medium.

A text processor may be composed with a document search and indexing service, which consist of document text crawling and indexing threads that are independent of the main program threads. The application can permit a user search of a document to be accelerated by utilizing an index of the document as part of a program transaction.

An extendable application can feature third-party plugins that may filter input cooperatively as though in a pipeline. The producing and consuming plugin stages may themselves be consumers and producers, respectively, of further plugin stages. The application may conduct transformation operations on data filtered through certain plugins as part of program transactions.

A web server application may aggregate statistical data on the requests served by sending metadata of information sent to a reducer thread that collates the statistical data and may be located in its own analytics module. The application threads that serve the data as part of their transactions identify and send the metadata to the aggregator, which runs independently yet still coordinates in consuming the metadata.

5.3 Example System Realization and Evaluation

The constructs and implementation were utilized to develop one of the examples from the previous section. An application logger is constructed that features a pipeline of logging threads. The first stage filters messages – informative (light) or trace (heavy) – onto distinct queues for separate logging threads to handle. Application threads generate messages modeling low or high logging message dispatch and arrival for the application loggers onto the initial filter queue of the pipeline. The application threads feature an atomic section, corresponding to a chunk of code within an application that is intended to be executed as one unit, such as a relational database subquery. From within the section the application threads are synchronized at a barrier to start together and then generate the messages. The called barrier routine contains a nested atomic section. Sync variables are utilized in the initial stage of the pipeline, to control access to the filter queue, as well as in the two downstream logging queues, and in the thread barrier procedure. Each pipeline stage's sync variable counts the number of elements in its corresponding queue. Threads cannot take elements from the queue if it is empty, nor can they put elements onto the queue if it is full, as accessed using the queue's sync variable. The application logger source is in the appendix.

Each message features an allocated character string buffer, a length field, and a stamp for the thread id of the generating application thread. Light (informative) messages were thirty-two bytes in length, while heavy (e.g. stack trace) were sixty-four kilobytes. Heavy message generation is eight times greater than the light generation case. Figures 5-3 through 5-6 display the average runtimes in seconds for one through eight processors for the four possible configurations of the two dimensions concerning the message type and message generation rate.

For the light weight messages generated at a low message generation rate, the performance improves with two threads, though deteriorates with a higher thread count, due to overhead of threading as well as synchronized access to a shared resource, since higher thread counts increase contention and serialize access to the queues, as well as increase scheduling delays. For light messages dispatched at a high rate, performance in terms of runtime is more than an order magnitude worse than the low rate of dispatch, and does not improve beyond two threads. For heavy weight messages, performance does not noticeably improve with more threads (except for the slight decrease in run time for the light dispatch rate at two threads.) However, performance does not degrade nearly as severely with more threads than in the cases with light weight message dispatch.

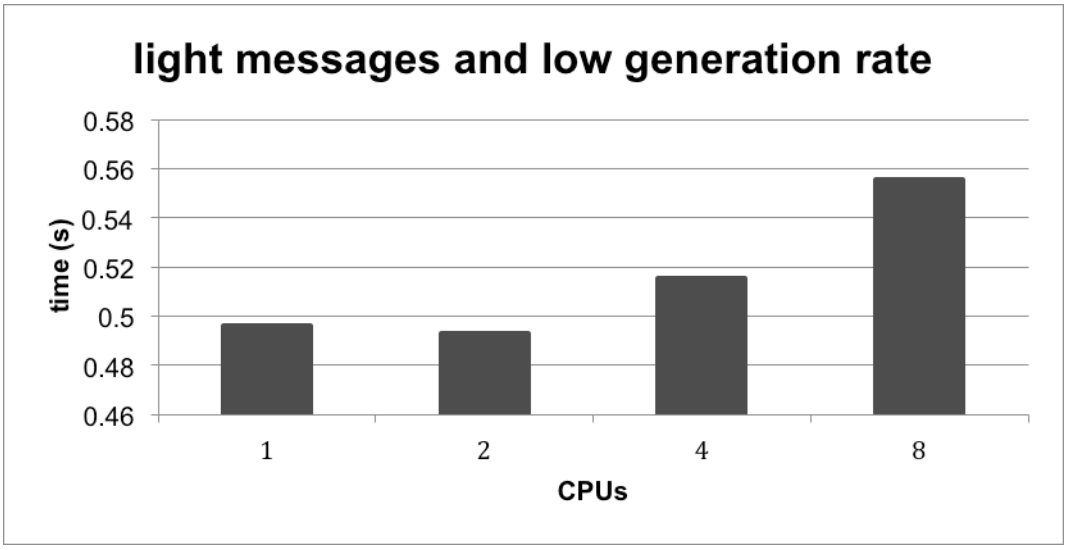

Figure 5-3 Light messages and low generation rate (lower is better)

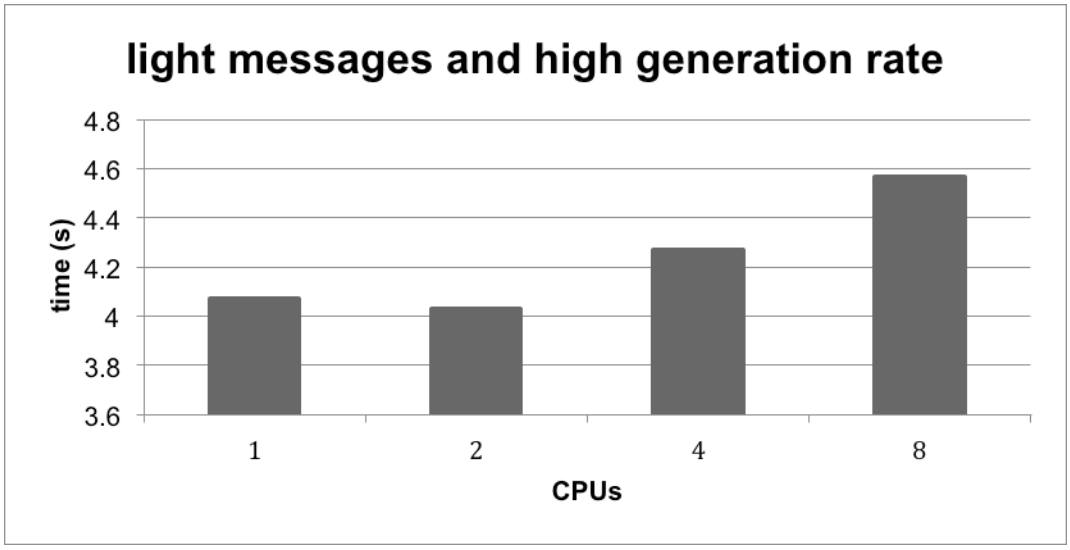

Figure 5-4 Light messages and high generation rate (lower is better)

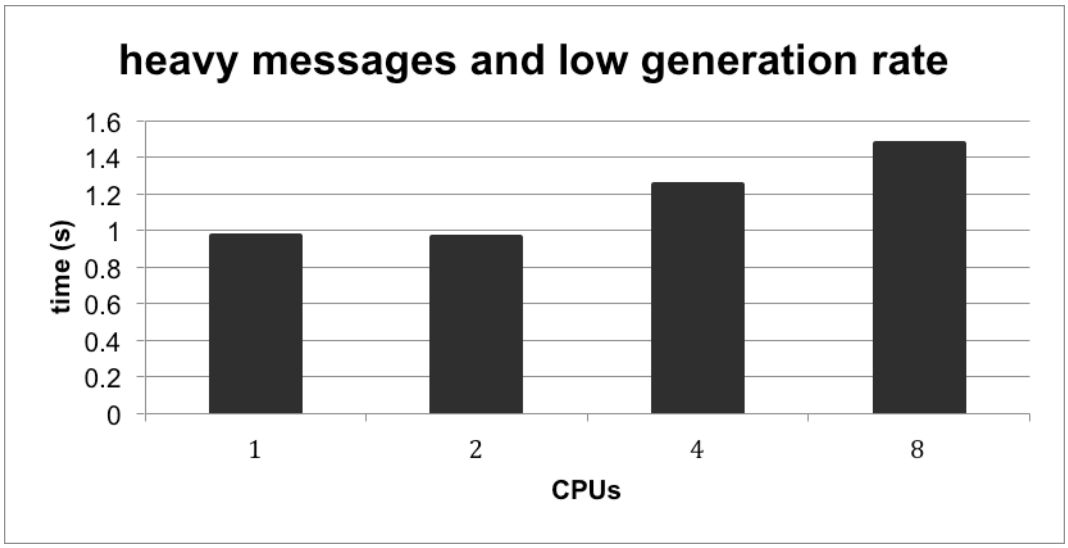

Figure 5-5 Heavy messages and low generation rate (lower is better)

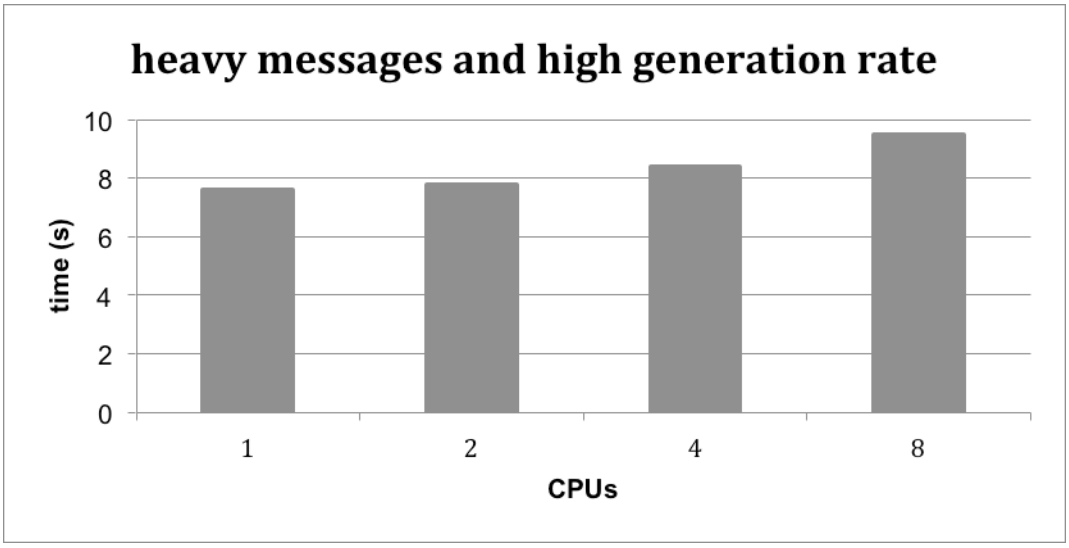

Figure 5-6 Heavy messages and high generation rate (lower is better)

Chapter 6
Conclusion and Future Work

The presented atomic section programming model features first class support for condition synchronization with truly nested transactional semantics. The benefits include composability of software, reuse of code and libraries featuring idioms such as many producer/consumer patterns, and barrier synchronization. Simplicity of the model is also a virtue as it decreases the cognitive load on the programmer.

6.1 Coda

The need for condition synchronization is outlined and the inadequacies of existing transactional memory systems detailed. A model and syntax for implementing condition synchronization in the context of atomic sections is presented along with its compositional properties. The sync variable construct enables nested transactions to block a thread until a predicate expression becomes true before the thread starts the transaction.

Notable findings of the model in regards to expressivity is that the atomic section interface is able to capture concurrent patterns and use cases featuring condition synchronization such as barriers and producer/consumer scenarios.

Open questions regarding the model are:

Is the transactional model as implemented general enough for constructing a broad cross section of concurrent software?

Are the constructs for condition synchronization sufficiently expressive to capture most use cases of conventional conditional variables?

What are the implications for conditions and transactions when embedded in a language with exception handling?

Are transactions in and of themselves a proper construct to reason about and craft concurrent distributed software that may need to coordinate across large distances without the strict synchronization requirements and overheads inherent to transactions?

6.2 Desiderata

Further quantitative evaluation of the implementation on the STAMP benchmark suite [MCKO08] against an STM competitor is in order. More extensive heuristics for assigning lock sets to atomic blocks are planned to be implemented as future work, in order to improve performance. In addition, exploring a hybrid combination of pessimistic and optimistic transactional memory [LM11] with support for condition synchronization is a fruitful endeavor. Finally, evaluation of the programming model on further benchmarks and larger software projects would be beneficial in gaining confidence with regards to the applicability and generality of the proposed atomic section interface.

Bibliography

[ALS06] Kunal Agrawal, Charles E. Leiserson, Jim Sukha. Memory Models for Open-Nested Transactions. MSPC 2006.

[Boeh09] H. J. Boehm. Transactional Memory Should Be an Implementation Technique, Not a Programming Interface. HotPar 2009.

[CCCMM05] Brian D. Carlstrom, et al. Transactional Execution of Java Programs. SCOOL 2005.

[CCG08] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. PLDI 2008.

[CCMM06] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, Kunle Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. HPCA 2006.

[CDE08] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Lock inference proven correct. FTfJP 2008.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. ACM Computing Surveys, June 1971.

[CFRWZ91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. TOPLAS 1991.

[CGE08] D. Cunningham, K. Gudka, and S. Eisenbach. Keep off the grass: Lock inference for atomicity. CC 2008.

[DDSSS10] Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, Michael Spear. Transactional Mutex Locks. Euro-Par 2010.

[DS09] P. Dudnik, M. M. Swift. Condition Variables and Transactional Memory: Problem or Opportunity. TRANSACT 2009.

[EFJM07] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, Rupak Majumdar. Lock allocation. POPL 2007.

[EGLT76] K. P. Eswaran, Jim N. Gray, R. A. Lorie, I. L. Traiger. The notions of consistency and predicate locks in a database system. Communications of the ACM, November 1976.

[Har09] T. Harris. Invited talk: Language Constructs for Transactional Memory. POPL 2009.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. OOPSLA 2003.

[HFP06] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock Inference for Atomic Sections. TRANSACT 2006.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III. Software transactional memory for dynamic-sized data structures. PODC 2003.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. ISCA 1993.

[HMPH05] Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. Composable memory transactions. PPoPP 2005.

[HPV07] R. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. PACT 2007.

[LA04] Chris Lattner, and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO 2004.

[List77] Andrew Lister. The problem of nested monitor calls. SIGOPS Operating Systems Review 1977.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context- sensitive points-to analysis with heap cloning practical for the real world. PLDI 2007.

[LM11] V. Luchangco, V. J. Marathe. Revisiting Condition Variables and Transactions. TRANSACT 2011.

[Lome77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. Conference on language design for reliable software 1977.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics. ASPLOS 2008.

[LR06] Larus, J.R. and Rajwar, R. Transactional Memory. Morgan & Claypool, 2006.

[MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. IISWC 2008.

[MCRS10] Sandya Mannarswamy, Dhruva R. Chakrabarti, Kaushik Rajan, Sujoy Saraswati. Compiler aided selective lock assignment for improving the performance of software. PPoPP 2010.

[Moss06] J.E.B. Moss. Open nested transactions: Semantics and support. Workshop on Memory Performance Issues 2006.

[Moss82] J.E.B. Moss. Nested transactions: An approach to reliable distributed computing. Symposium on Reliability in Distributed Software and Database Systems 1982.

[MQ08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay, Eric Brewer. Autolocker: synchronization inference for atomic sections. POPL 2006.

[Ni07] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, Tatiana Shpeisman. Open Nesting in Software Transactional Memory. PPoPP 2007.

[Nish04] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. VM 2004.

[OH05] K. Olukotun, L. Hammond. The Future of Microprocessors. ACM Queue September 2005.

[Parr06] Terence Parr. ANSI C grammar for ANTLR v3. 2006. www.antlr.org/grammar/1153358328744/C.g

[SBNSA97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. SOSP 1997.

[SKBY07] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, Michal Young. Transactions with isolation and cooperation. OOPSLA 2007.

[SL05] H. Sutter, J. Larus. Software and the Concurrency Revolution. ACM Queue September 2005.

[ST95] Nir Shavit and Dan Touitou. Software Transactional Memory. PODC 1995.

[Stee96] B. Steensgaard. Points-to analysis in almost linear time. POPL 1996.

[UBES09] Takayuki Usui, Reimer Behrends, Jacob Evans, Yannis Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. PACT 2009.

[VTD06] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-Oriented Language. POPL 2006.

[ZSZSG07] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, Guang R. Gao. Optimized lock assignment and allocation: a method for exploiting concurrency among critical sections. PPoPP 2007.

[ZSZSG08] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, Guang R. Gao. Minimum lock assignment: A Method for Exploiting Concurrency among Critical Sections. LCPC 2008.

Appendix A

Data for graphed quantitative results is presented in this appendix, as well as source code.

Table 6-1 Producer / consumer microbenchmark data

1	0.503	0.561	0.434	0.473	0.478
2	0.466	0.449	0.464	0.433	0.41
4	0.36	0.364	0.359	0.361	0.359
8	0.808	0.451	0.716	0.497	0.643

Table 6-2 Thread barrier microbenchmark data

1	0.547	0.549	0.548	0.546	0.547
2	0.549	0.548	0.551	0.549	0.551
4	0.559	0.552	0.564	0.559	0.551
8	0.574	0.564	0.573	0.582	0.581

Source code for application logger:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int NUM_APP_THREADS = -1;
int NUM_LOG_THREADS = -1;
int LOWHIGH = -1;
int LIGHTHEAVY = -1;
int NUM_ITEMS = -1;
int LIGHTLEN = 32;
int HEAVYLEN = 65536;
int LOWRATE = 128;
int HIGHRATE = 1024;
```

```
typedef struct {char* msg; int len, stamp} TItem;

const int MAX_ENTRIES = 1024;
TItem* fil_buf[MAX_ENTRIES];
sync int fil_num_entries = 0;
int fil_in = 0, fil_out = 0;

TItem* light_buf[MAX_ENTRIES];
sync int light_num_entries = 0;
int light_in = 0, light_out = 0;

TItem* heavy_buf[MAX_ENTRIES];
sync int heavy_num_entries = 0;
int heavy_in = 0, heavy_out = 0;

void barrier()
{
  sync static int count = 0;
  atomic {
    count++;
  }
  atomic (count == NUM_APP_THREADS) {
  }
}

void putfilter(TItem* item) {
      atomic (fil_num_entries < MAX_ENTRIES) {
            fil_buf[fil_in] = item;
            fil_in = (fil_in + 1) % MAX_ENTRIES;
            fil_num_entries++;
      }
}

TItem* getfilter(void) {
      TItem* item;
      atomic (fil_num_entries > 0) {
            item = fil_buf[fil_out];
            fil_out = (fil_out + 1) % MAX_ENTRIES;
            fil_num_entries--;
      }
      return item;
}

void putlight(TItem* item) {
```

```
      atomic (light_num_entries < MAX_ENTRIES) {
            light_buf[light_in] = item;
            light_in = (light_in + 1) % MAX_ENTRIES;
            light_num_entries++;
      }
}

TItem* getlight(void) {
      TItem* item;
      atomic (light_num_entries > 0) {
            item = light_buf[fil_out];
            light_out = (light_out + 1) % MAX_ENTRIES;
            light_num_entries--;
      }
      return item;
}

void putheavy(TItem* item) {
      atomic (heavy_num_entries < MAX_ENTRIES) {
            heavy_buf[heavy_in] = item;
            heavy_in = (heavy_in + 1) % MAX_ENTRIES;
            heavy_num_entries++;
      }
}

TItem* getheavy(void) {
      TItem* item;
      atomic (heavy_num_entries > 0) {
            item = heavy_buf[heavy_out];
            heavy_out = (heavy_out + 1) % MAX_ENTRIES;
            heavy_num_entries--;
      }
      return item;
}

void *app(void *arg)
{
  int tid = *((int *) arg);
  atomic {
    barrier();
    for(int
i=0;i<((NUM_LOG_THREADS/NUM_APP_THREADS)*NUM_ITEMS/NUM_APP_THREADS);
i++)   {
      TItem* elp=(TItem*)malloc(sizeof(TItem));
      elp->msg=malloc(sizeof(LIGHTLEN));
      elp->len=LIGHTLEN;
      elp->stamp=i*NUM_APP_THREADS+tid;
      putfilter(elp);
```

```
      printf("thread with id %d generated message with stamp %d\n",
tid, elp->stamp);
    }
  }
}

void *lightlogger(void *arg)
{
  int tid = *((int *) arg);
  for(int i=0;i<(NUM_ITEMS/NUM_LOG_THREADS);i++) {
    TItem* elp=getlight();
    printf("thread with id %d logged light message with stamp %d\n",
tid, elp->stamp);
    free(elp->msg);
    free((void*)elp);
  }
}

void *heavylogger(void *arg)
{
  int tid = *((int *) arg);
  for(int i=0;i<(NUM_ITEMS/NUM_LOG_THREADS);i++) {
    TItem* elp=getheavy();
    printf("thread with id %d logged light message with stamp %d\n",
tid, elp->stamp);
    free(elp->msg);
    free((void*)elp);
  }
}

void *filter(void *arg)
{
  int tid = *((int *) arg);
  for(int
i=0;i<(((NUM_LOG_THREADS/NUM_APP_THREADS)*NUM_ITEMS/NUM_APP_THREADS)
);i++) {
    TItem* elp=getfilter();
    if (elp->len == LIGHTLEN) {
      putlight(elp);
      printf("thread with id %d filtered light message with stamp %d
onto light queue\n", tid, elp->stamp);
    }
    else {
      putheavy(elp);
      printf("thread with id %d filtered heavy message with stamp %d
onto heavy queue\n", tid, elp->stamp);
    }
  }
```

```
}

int main (int argc, char *argv[])
{
  pthread_t logthreads[32];
  int logthread_args[32];
  pthread_t appthreads[16];
  int appthread_args[16];
  int rc;

  if (argc==1||argc!=5) {
    printf("Usage: <# app threads> <# logging threads> (0=='low
rate' | 1=='high rate') (0=='light msg' | 1=='heavy msg')\n");
    exit(0);
  }
  NUM_LOG_THREADS = atoi(argv[2]);
  NUM_APP_THREADS = atoi(argv[1]);
  LOWHIGH = atoi(argv[3]);
  LIGHTHEAVY = atoi(argv[4]);

  if (LOWHIGH)
    NUM_ITEMS=HIGHRATE;
  else
    NUM_ITEMS=LOWRATE;

  /* create all threads */
  for (int i=0; i<NUM_LOG_THREADS; ++i) {
    logthread_args[i] = i;
    if (i % 3 == 0) {
      printf("creating filter logger thread %d \n", i);
      rc = pthread_create(&logthreads[i], (void *) NULL, filter,
(void *) &logthread_args[i]);
    }
    else if (i % 3 == 1) {
      printf("creating light message consuming logger thread %d \n",
i);
      rc = pthread_create(&logthreads[i], (void *) NULL,
lightlogger, (void *) &logthread_args[i]);
    }
    else if (i % 3 == 2) {
      printf("creating heavy message consuming logger thread %d \n",
i);
      rc = pthread_create(&logthreads[i], (void *) NULL,
heavylogger, (void *) &logthread_args[i]);
    }
  }
  for (int i=0; i<NUM_APP_THREADS; ++i) {
    appthread_args[i] = i;
```

```
    printf("creating message generating app thread %d \n", i);
    rc = pthread_create(&appthreads[i], (void *) NULL, app, (void *)
&appthread_args[i]);
  }

  /* wait for all threads to complete */
  for (int i=0; i<NUM_APP_THREADS; ++i) {
    printf("joining app thread %d\n", i);
    rc = pthread_join(appthreads[i], NULL);
  }
  for (int i=0; i<NUM_LOG_THREADS; ++i) {
    printf("joining log thread %d\n", i);
    rc = pthread_join(logthreads[i], NULL);
  }
}
```