Techniques for Improving the Performance of Software Transactional Memory

Srđan Stipić

Department of Computer Architecture

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Doctor of Philosophy in Computer Architecture

July, 2014

Advisor: Adrián Cristal
Co-Advisor: Osman S. Unsal
Tutor: Mateo Valero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Curso académico:

Acta de calificación de tesis doctoral

Nombre y apellidos

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada ____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a	(Nombre, apellidos y firma) Secretario/a

(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

______________________, ________ de __________________ de _______________

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretaria de la Comisión Permanente de la Escuela de Doctorado

Barcelona a ________ de ____________________ de __________

To my parents.

Acknowledgements

I am thankful to a lot of people without whom I would not have been able to complete my PhD studies. While it is not possible to make an exhaustive list of names, I would like to mention a few. Apologies if I forget to mention any name below.

I would like to thank my advisors Adrián Cristal and Osman Unsal for all the help and guidance they provided during my PhD studies. I would also like to acknowledge Ibrahim Hur for his help while he was part of Barcelona Supercomputing Center. I also thank Mateo Valero for his dedication and continuous effort in making the Barcelona Supercomputing Center such a great platform for research.

I would like to thank Tim Harris, who kindly mentored me during my three-month stay in Microsoft Research Cambridge. I had a great and productive time in Microsoft thanks to Tim's always positive attitude and enthusiasm.

I would like to thank Lukasz Skital, Rory Ward, and Tom Limoncelli from Google where I did internship in Google-Dublin/Ireland for three months in 2011. Internship in Google gave me the opportunity to work in industrial environment where I was working on their internal tool for managing network connections in Google's data-centers.

I would also like to acknowledge all my friends and colleagues from the office that helped me throughout my PhD; for their insights and expertise in technical matters, and for their unconditional support that has been crucial to keep me sane. Many thanks go to Adrià Armejach, Ana Jokanović, Azam Seyedi, Bojan Marić, Branimir Dickov, Chinmay Kulkarni, Cristian Perfumo, Daniel Nemirovsky, Ege Akpinar, Ferad

Zyulkyarov, Gokcen Kestor, Gülay Yalçın, Ivan Ratković, Javier Arias, Jelena Koldan, Maja Etinski, Milan Pavlović, Milan Stanić, Milovan Đurić, Miloš Milovanović, Nehir Sonmez, Nikola Marković, Nikola Vujić, Oriol Arcas, Paul Carpenter, Saša Tomić, Timothy Hayes, Vasilis Karakostas, Vesna Smiljković, Vladimir Subotić, Vladimir Gajinov, Vladimir Marjanović, and many others. I sincerely thank you all for your help and all the great moments we have had together.

I would like to thank my friends and family for supporting me during this endeavour. My deepest thanks to my wife Claudia for her love and for being there for me all the time. This dissertation would have not been possible without her.

Abstract

Transactional Memory (TM) provides software developers the opportunity to write concurrent programs more easily compared to any previous programming paradigms and promisses to give a performance comparable to lock-based synchronizations.

Current Software TM (STM) implementations have performance overheads that can be reduced by introducing new abstractions in Transactional Memory programming model.

In this thesis we present four new techniques for improving the performance of Software TM: (i) Abstract Nested Transactions (ANT), (ii) TagTM, (iii) profile-guided transaction coalescing, and (iv) dynamic transaction coalescing. ANT improves performance of transactional applications without breaking the semantics of the transactional paradigm, TagTM speeds up accesses to transactional metadata, profile-guided transaction coalescing lowers transactional overheads at compile time, and dynamic transaction coalescing lowers transactional overheads at runtime.

Our analysis shows that Abstract Nested Transactions, TagTM, profile-guided transaction coalescing, and dynamic transaction coalescing improve the performance of the original programs that use Software Transactional Memory.

Contents

Chapter 1

Introduction

1.1 Introduction to Transactional Memory

The multi-core era has already arrived. Currently, most of the new desktop or laptop computers have two or more cores. Intel and AMD are promising that in coming years we will have 32, 64 or more cores integrated in to a single chip. The new game consoles like XBOX ONE from Microsoft and PlayStation 4 from Sony have multi core CPUs (8 CPU cores with multi-core GPU). Still, the developers of new applications have hard time writing concurrent programs that utilize all of the available CPU cores. The reason for this is that the programmers are still using locks as their main building blocks for writing concurrent programs. The use of locks introduces problems like: dead lock and live lock, that are hard to detect, debug and reproduce. The transactional memory raises the level of abstraction for the programmers and elegantly eliminates the problems stated above.

The transactional memory (TM) technology borrows proven concurrency-control concepts from work done over the decades in the database field, and try to apply them in everyday programming languages (C, C++, Java, C#).

Transactional Memory (TM) systems can be subdivided into three flavors: Hardware TM (HTM), Software TM (STM) and Hybrid TM (HyTM) (the mix of hardware and software transactional memory).

1.1.1 Transactions in databases

In the context of databases, a single logical operation on the data is called a transaction. A set of properties that database transactions guarantee are: atomicity, consistency, isolation, and durability (ACID).

An example of a transaction is a transfer of funds from one account to another, even though it might consist of multiple individual operations, such as debiting one account and crediting another.

Atomicity

Atomicity refers to the ability of the database to guarantee that either all of the tasks of a transaction are performed or none of them are. For example, the transfer of funds can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account will not be debited if the other is not credited. Atomicity states that database modifications must follow an “all or nothing” rule. Each transaction is said to be “atomic.” If one part of the transaction fails, the entire transaction fails. It is critical that the database management system maintains the atomic nature of transactions in spite of any operating system or hardware failure.

Consistency

Consistency property ensures that the database remains in a valid state before the start of the transaction and after the transaction is over, whether successful or not.

Consistency states that only valid data will be written to the database. If, for some reason, a transaction is executed that violates the database’s consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules

Isolation

Isolation refers to the constraint that prevent other operations to access or see the data in the intermediate state during transaction. This constraint is required to maintain the consistency between transactions in a DBMS system.

Durability

Durability refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone. This means it will survive system failure, and that the database system has checked the integrity constraints and won't need to abort the transaction. Many databases implement durability by writing all transactions into a log that can be played back to recreate the system state right before the failure. A transaction can only be deemed after it is safely stored in the log.

1.1.2 Transactional memory

In computer science, transactional memory (TM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. It functions as an alternative to lock-based synchronization. A transaction in this context is a piece of code that executes a series of reads and writes to a shared memory. These reads and writes logically occur at a single instant in time (atomicity); intermediate states are not visible to other (successful) transactions (isolation). The idea of providing hardware support for transactions originated in paper by Tom Knight [Knight (1986)]. The idea was popularized by Maurice Herlihy and J. Eliot B. Moss [Herlihy & Moss (1993)] for hardware TM. In 1995 Nir Shavit and Dan Touitou extended this idea to software-only transactional memory (STM) [Shavit & Touitou (1995)].

Transactions in TM do not try to provide all ACID properties of database transactions. From the programming point of view only atomicity and isolation properties are required because TM as a concept tries to eliminate explicit use of locks (TM systems can use locks in their implementations but those locks are not exposed to the programmers).

```
// lock version                          // transactional version
while(1){                                atomic{
  lock(counter.lock);                      if(!(counter.value > 0)) retry;
  if(counter.value > 0) break;             --counter.value;
  unlock(counter.lock);                  }
}
// we are holding the lock
--counter.value;
unlock(counter.lock);
```

Figure 1.1: Decrementing a counter

Consistency property (as it is defined in databases) does not exist in TM, but most of TM systems provide a ‘retry’ operation that automatically restarts the transaction. The retry operation can be used to implement consistency rules that are specified in the program.

The example in Figure 1.1 shows how the consistency rules are programmed by using locks and by using transactions with retry operation. For example, assume that we want to decrement a shared counter only if it is greater than zero (consistency rule). In the version of the program that uses locks we have have to take special care (to release the lock) in order not to create a deadlock. We can see that the transactional version is much simpler and it does not suffer from the deadlock problem because the TM system guaranties correct execution.

Durability is not provided in TM because the programs which use TM are not durable by definition.

Version control

Depending on the moment when we update the shared memory locations we can identify two types of version control:

- *Eager versioning*: a write access within a transaction immediately writes to memory the new data version. The old version is buffered in an undo log.

- *Lazy versioning*: stores all new data versions in a write buffer until the transaction completes.

Conflict detection

Depending on the moment when we detect conflicts between transactions we can identify two types of version control:

- *Eager conflict detection*: TM detects a conflict when a transaction tries to access a memory reference.
- *Lazy conflict detection*: TM detects conflicts when the first of two or more conflicting transactions commits.

Read and Write set

During the execution of a transaction, the transaction speculatively reads and writes shared memory locations. In order to keep the track of read and write accesses to shared memory location, each transaction keeps the access information in its read and write sets. We define read and write set of a transaction in the following way:

- *Read set* is the set of shared memory locations that are read speculatively during the execution of the transaction.
- *Write set* is the set of shared memory locations that are written speculatively during the execution of the transaction.

Read and write set can overlap or they can be disjunctive. At the commit time, the transaction validates its read and write set and commits its write set (the transaction updates the shared memory locations with the speculative values from the write set).

Conceptual advantages and disadvantages of TM

TM greatly simplifies conceptual understanding of multithreaded programs and helps make programs more maintainable by working in harmony with existing high-level abstractions such as objects and modules. Lock-based programming has a number of well-known problems that frequently arise in practice:

- Locks require thinking about overlapping operations and partial operations in distantly separated and seemingly unrelated sections of code, a task which is very difficult and error-prone for programmers.
- The fine grained locking require programmers to adopt a locking policy to prevent deadlock, livelock, and other failures to make progress. Such policies are often informally enforced and fallible, and when these issues arise they are difficult to reproduce and debug.
- Locks can lead to priority inversion, a phenomenon where a high-priority thread is forced to wait on a low-priority thread holding exclusive access to a resource that it needs.

In contrast, the concept of a memory transaction is much simpler, because each transaction can be viewed in isolation as a single-threaded computation. Deadlock and livelock are either prevented entirely or handled by an external transaction manager. Priority inversion can still be an issue, but high-priority transactions can abort conflicting lower-priority transactions that have not already committed.

On the other hand, the need to abort failed transactions also places limitations on the behavior of transactions: they cannot perform any operation that cannot be undone, including most I/O.

1.1.3 Nested Transactions

Figure 1.2 shows an example of two functions `f()` and `g()` where each function contains one transaction. The function `g()` executes transaction that contains the call to function `f()`. Because function `f()` is marked as `inline`, the compiler inlines the body of function `f()` in the function `g()`. After inlining, the function

```
inline void f() {
  atomic { a++; }
}
```

```
void g() {
  atomic {
    b++;
    f();
  }
}
```

```
// after inlining
void g() {
  // outer transaction
  atomic {
    b++;
    // inner transaction
    atomic { a++; }
  }
}
```

Figure 1.2: Nested transactions

`g()` contains two transactions (`outer` and `inner`). TM system define 3 modes how these two transactions execute with respect to other transactions in the system: (i) flat nesting, (ii) closed nesting, and (iii) open nesting.

Flat nesting mode executes outer transaction as one big transaction. Flat nesting ignores the presence of the nested atomic block but executes the body of nested transaction as part of the outer transaction. If a conflict happens on `a` or `b`, the outer transaction is restarted. When the outer transaction finishes executing, the speculative values of `a` and `b` are committed to the memory.

Closed nesting mode executes outer transaction as one big transaction. If a conflict happens on `a`, the inner transaction is restarted; and if the conflict happens on `b`, the outer transaction is restarted. When the outer transaction finishes executing, the speculative values of `a` and `b` are committed to the memory.

Open nesting mode executes outer transaction as two separate transactions. If a conflict happens on `a` inner transaction is restarted and if conflict happens on `b` the outer transaction is restarted. When the inner transaction finishes executing, the speculative value of `a` is committed to the memory (even though the outer transaction did not finish). When the outer transaction finishes executing, the speculative value of `b` is committed to the memory.

1.1.4 Software Transactional Memory (STM)

Unlike the locking techniques used in most modern multithreaded applications, a thread speculatively access the shared memory without regard for what other threads might be doing, recording every read and write that it is performing in a log. Instead of placing the burden on the writer thread to make sure it does not adversely affect other operations in progress, it is placed on the reader thread, who after completing an entire transaction verifies that other threads have not concurrently made changes to memory that it accessed in the past. This final operation, in which the changes of a transaction are validated and, if validation is successful, made permanent, is called commit. A transaction may also abort at any time, causing all of its prior changes to be rolled back and undone. If a transaction cannot be committed due to conflicting changes, it is typically aborted and re-executed from the beginning until it succeeds.

However, in practice STM systems also experience a performance hit relative to fine-grained lock-based systems on small numbers of processors (1 to 4 depending on the application). This is primarily due to the overhead associated with maintaining the log and the time spent committing transactions.

1.1.5 Hardware Transactional Memory (HTM)

In HTM systems, the hardware ensures correct transaction execution. HTM – compared with STM – does not experience the performance penalties. Hardware implements all necessary transactional mechanisms in caches [Hammond *et al.* (2004a), Ananian *et al.* (2005)] or memory directory [Moore *et al.* (2006b)].

Currently proposed HTM systems fall into three regions of the design space:

- LL: lazy conflict detection, lazy version management [Hammond *et al.* (2004a)]
- EL: eager conflict detection, lazy version management [Ananian *et al.* (2005)]

- EE: eager conflict detection, eager version management [Moore *et al.* (2006b)]

In the paper [Bobba *et al.* (2007)], the authors show that lazy/lazy TM perform better than eager/eager TMs in general case.

The downsides of HTM is that none of the CPU manufactures (Intel, AMD, IBM) are planing to implement full HTM in their CPUs, because of its complexity. Sun Microsystems planed to release the Rock CPU that provide limited support for executing small transactions in hardware [Moir *et al.* (2008)] but in the end the project was canceled. Intel released Haswell CPU with best effort HTM system [Yoo *et al.* (2013)] that executes transaction as long as the transactional data fit in L1 cache. When the transactional data overflows the cache, the transaction use software fallback mechanism.

1.1.6 Hybrid Transactional Memory

Hybrid transactional memory promises best of the two worlds. Small transitions can be executed in hardware and larger ones in software. Most transactions have small size [Ananian *et al.* (2005)] and HTM system can execute them in hardware which as a consequence gives us the maximum performance for the majority of the transactions. The transactions that exceed hardware resources will fall back to software mode.

University of Rochester proposed an extension to the MESI protocol [Shriraman *et al.* (2006)] that provides acceleration of STM. In 2006. Intel proposed adding few additional bits to the cache lines of the CPU in order to follow the validity of the memory locations that are referenced by STM [Saha *et al.* (2006a)]. Sun Microsystems [Moir *et al.* (2008)] proposed a hybrid TM approach in which transactions execute in hardware and if they overflow, they are rolled back and restarted in software.

1.2 STAMP Benchmark Suite

STAMP benchmark suite [Minh *et al.* (2008)] is de facto standard in TM community for the evaluation of TM systems, thus, we use it for the evaluation of hardware and software techniques presented in this thesis [Chapters 3, 4,

Application	**Domain**	**Description**
bayes	machine learning	Learns structure of a Bayesian network
genome	bioinformatics	Performs gene sequencing
intruder	security	Detects network intrusions
kmeans	data mining	Implements K-means clustering
labyrinth	engineering	Routes paths in maze
ssca2	scientific	Creates efficient graph representation
vacation	online transaction processing	Emulates travel reservation system
yada	scientific	Refines a Delaunay mesh

Table 1.1: STAMP benchmark suite applications

and 5]. STAMP consist of eight different benchmarks (Table 1.1) that cover a wide range of transactional execution cases (e.g., small/large transactions, small/large conflict rate). The benchmarks are: **bayes**, **genome**, **intruder**, **kmeans**, **labyrinth**, **ssca2**, **vacation**, **yada**.

Bayes application implements algorithm for learning the structure of Bayesian networks from observed data. The Bayesian network itself is represented as a directed acyclic graph, with a node for each variable and an edge for each conditional between variables. Initially, the network hash no dependencies among variables, and the algorithm incrementally learns dependencies by analysing the observed data. On each iteration, each thread is given a variable to analyze, and as more dependencies are added to the network, connected subgraphs of dependent variables are formed.

A transaction is used to protect the calculation and addition of a new dependency, as the result depends on the extent of the subgraph that contains the variable being analyzed. Using transaction is much simpler than a lock-base approach as using locks would require manually orchestrating a two-phase locking scheme with deadlock detection and recovery to allow concurrent modification of the graph. Calculations of new dependencies take up most of the execution time, causing **bayes** to spend almost all its execution time in long transactions that have large a read and write set. Overall, this benchmark has high amount of contention as the subgraph changes frequently.

Genome application takes large number of DNA segments and tries to match them to reconstruct the original source genome. The application operates in two phases. In the first phase, **genome** uses hash set to remove the duplicate DNA segments, and in the second phase, tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments.

Transaction are used in each phase of the benchmark. Additions to the set of unique segments are enclosed by transactions to allow concurrent accesses, and accesses to the global pool of unmatched segments are also enclosed by transactions since threads may try to remove the same segment. Overall, the transaction in **genome** are of moderate length and have moderate read write set size. Additionally, almost all of the execution time is transactional, and there is little contention.

Intruder application uses signature-based network intrusion detection system to scan network packets for matches against a known set of intrusion signatures. Network packets are processed in parallel and go through three phases: capture, reassembly, and detection. The main data structure in the capture phase in simple FIFO queue, and the reassembly phase uses a self-balancing tree that contains lists of packets that belong to the same session and uses coarse-grain synchronization. The capture and reassembly phase are each enclosed by transactions. So, the code for each phase is equivalent to the lock version and has short transactions with moderate levels of contention.

Kmeans application implements K-means algorithm that groups objects in and N-dimensional space into K clusters. The algorithm is commonly used to partition data items into related subsets. The application uses transactions to protect the update of the cluster center that occurs during each iteration. The amount of contention among threads depends on the value of K, with larger values resulting in less frequent conflicts as it is less likely that two threads are concurrently operation on the same cluster center. When updating the cluster centers, the size of transaction is proportional to the dimensionality of the space. Thus, the size of the transactions are relatively small.

Labyrinth application uses three-dimensional uniform grid that represents maze. The application uses threads, where each thread grabs a start and end point and tries to connect it with adjacent maze grid points. The calculation of the path is executed in a single transaction. A conflict occurs when two threads pick paths that overlap. Almost all of the application execution time is taken by the patch calculation, and this operation reads and writes an amount of data proportional to the number of total maze grid points, thus, **labyrinth** has very long transactional with very large read and write sets. Almost all the code is executed transactionally, and the amount of contention is very high.

Vacation application implements a travel reservation system powered by an in-memory database. Several client threads interact with the database through transactional requests. The performance is measured as the number of served client requests per second. The paper [Zyulkyarov *et al.* (2010a)] shows that the performance bottleneck in **vacation** is the red-black tree that is used as database storage, and they suggest replacing it with a hash table. We follow the same approach. Moreover, we identified another performance bottleneck: a function for generating random numbers was on the critical path limiting the throughput. We moved the generation of random numbers to the initialization part of the benchmark, where we populate per thread arrays with random numbers. We replaced the original random number function with an array indexing function that reads the random numbers from the arrays. In this way, we improved the throughput of the original benchmark without modifying its semantics.

Ssca2 [Bader & Madduri (2005)] application is a synthetic benchmark that operates on a large, directed, weighted multi-graph. The main loop of **ssca2** traverses all the edges of the graph. The graph can be traversed in any order and the final execution of the benchmark is the same. Thus, transactions in the main loop execute in arbitrary order. We modified the main loop and instead of executing the transactions immediately, we buffer the values of the variables used in the transactions. When the buffer gets full, we execute all the transactions in a tight loop.

Yada application implements Ruppert's algorithm for Delaunay mesh refinement [Ruppert (1995)]. **Yada** uses graph data-structure to store all mesh triangles. In each iteration of the algorithm, one triangle is removed from the work queue, its retriangualtion is performed on the mesh, and any new triangles that result from the retriangualation are added to the work queue. Accesses to the work queue are enclosed by transaction and almost all execution time is spend calculation the retriangualtion.

1.3 Problem Statement

This thesis addresses several issues present in STM systems. These can be categorised under three heads: unintended transaction aborts, transactional meta-data accesses, and transaction starting and committing overheads.

1.3.1 Unintended Transaction Aborts

STM implementations use conflict detection based on the memory location that a transaction accesses. When two transactions update or write and read the same memory location, STM system detects the conflict and aborts one of the two transactions. For example, if two transactions insert items into the same hash-table under different keys witch happen to hash to the same bucket, the insertions update the same memory locations. This creates conflict between transactions because the implementation of the hash table updates the same memory locations even though the higher-level operations they are performing are commutative. Even though the transactions that update the hash table do not have conflict at semantic level (because the final content of the hash table is always and in independent of the order of the transaction commit order), they experience unintended transactional aborts because they update the same memory location.

1.3.2 Transactional Meta-data Accesses

For every transaction in a program, the STM system keeps additional meta-data that is used for storing information about currently executing transactions. This meta-data is accessed and updated during every transactional operation (e.g, transaction read, write, commit). Most of the transactional overheads appear due to meta-data accesses and it is critical to reduce these overheads in order to make STM applications competitive with lock based application.

1.3.3 Transaction starting and committing overheads

For most of the transactions executing in a program, STM system usually initializes transactional meta-data before the execution of a transaction and has to commit the data and meta-data at the end of a transaction. In the case of short transactions, the initialization and commit overheads can dominate the transaction execution time. If transactions are numerous and small, STM overheads accumulate rapidly.

1.4 Previous Techniques for Improving the Performance of STMs

The paper [Chung *et al.* (2008)] uses STM to provide thread-safe dynamic binary translation. They translate all the code transactionally and create transactions at the basic block level. They reduce the overhead of transactional execution by merging consecutive basic blocks into a large transaction.

To address the overhead of STM, researchers have proposed hybrid TM systems that provide some hardware support for conflict detection in STM code [Minh *et al.* (2007a); Saha *et al.* (2006b); Shriraman *et al.* (2007)]. For example, the paper [Saha *et al.* (2006b)] introduces hardware accelerated software transactional memory (HASTM) as ISA extensions and novel hardware mechanisms that improve STM performance by using additional bits per cache line. In addition, minor hardware changes can improve performance of transactional applications significantly.

The paper [Wang *et al.* (2012)] describes a BlueGene/Q machine with best effort HTM. BlueGene/Q maintains speculative states in the L2 cache and uses software register checkpointing with the `setjmp()` function. Even though BlueGene/Q has support for best effort hardware TM, this HTM implementation has issues with small transactions. The authors of the paper admit that the software register checkpointing has significant overhead for small transactions.

Many optimizations have been proposed for compilers and runtime systems to reduce the overheads of STM read and write operations [Afek *et al.* (2011); Dragojevic *et al.* (2009); Wang *et al.* (2007); Wu *et al.* (2009)]. The paper [Wang *et al.*

(2007)] provides compiler optimizations for eliminating unnecessary read/write barriers (read after write barrier, barriers on local variables, etc.) and register checkpointing. The paper [Afek *et al.* (2011)] proposes static analysis and code motion to decrease the number of memory accesses. For eliminating redundant barriers and checkpointing, the paper [Wu *et al.* (2009)] uses compiler optimizations on statistics collected at runtime. The paper [Dragojevic *et al.* (2009)] presents runtime and compiler optimizations to identify transaction-local stack and heap allocation, and an API for annotating thread-local and read-only memory regions.

The paper [Adl-Tabatabai *et al.* (2006a)] uses compiler and runtime optimizations for transactional memory language constructs. They use a JIT compiler to reduce the overheads of STM.

Yoo et al. [Yoo *et al.* (2013)] evaluate the performance of Intel Transaction Synchronization Extensions (TSX) that implement best effort hardware transactional memory. To quantify the transactional overheads, the authors analyze the performance benefits of static and dynamic transaction coalescing[1] and show that transaction coalescing improves performance. However, their implementation of transaction coalescing lacks the profiling mechanism that identifies the optimal transaction granularity at run-time. Their dynamic coalescing combines multiple dynamic instances of the same transactional region.

1.5 Thesis Contributions and Organization

This thesis provides an extensive study of optimization techniques for improving the performance of software transactional memory systems. The main contribution of this thesis are:

- Four new techniques for improving the performance of STM systems: (i) Abstract Nested Transactions (ANT), (ii) TagTM, (iii) Profile-Guided Transaction Coalescing, and (iv) Dynamic Transaction Coalescing (DTC).
- We explain, in detail the hardware and software implementation of our techniques.

[1]Transaction coalescing is called coarsening in Intel's terminology.

- We show that our techniques improve the performance of STM systems.

The rest of the thesis is organized as follows: Chapter 2 explains Abstract Nested Transaction, a runtime technique that reduces the overheads of repeated restarts of transactions; Chapter 3 explains TagTM, a hardware and software technique that reduces the overheads of transactional read, commit, and validate operations; Chapter 4 explains Profile-Guided Transaction Coalescing, a profile-guided compile-time technique that reduces transactional overheads by coalescing transactions; Chapter 5 explains Dynamic Transaction Coalescing, a compile-time and run-time profiling technique that dynamically adjusts the transaction granularity to improve the transactional throughput; Chapter 6 concludes the thesis; and Chapter 7 lists the publications during my PhD.

Chapter 2

Abstract Nested Transactions

The content of this chapter was presented in the paper "Abstract Nested Transactions" at TRANSACT'07 (The Second ACM SIGPLAN Workshop on Transactional Computing).

2.1 Introduction to Abstract nested transactions

TM implementations typically use conflict detection based on the memory locations that a transaction accesses. This can cause transactions to be re-executed because of *benign conflicts*, for example if two transactions insert items into the same hashtable under different keys which happen to hash to the same bucket: the insertions update the same memory locations, even though the higher-level operations they are performing are commutative.

In this chapter we introduce *abstract nested transactions* (ANTs) as a way of improving the scalability of atomic blocks that experience benign conflict. The main idea is that ANTs should contain operations that are likely to be the victims of benign conflicts. The run-time system then performs extra book-keeping so that, if an ANT does experience a conflict, the ANT can be re-executed without needing to re-run the larger transaction that contains it. Unlike closed nested transactions (CNTs) this re-execution can happen *after* the ANT has finished running – in our implementation we only re-execute ANTs at the point that we try to commit a top-level atomic block.

Moving code into or out of an ANT is *semantics preserving*: ANTs affect only the program's performance, not its possible results. This opens the door for future research in automatic ways to place ANTs in programs in order to deal with contention 'hot spots' that they experience.

2.2 Motivation for Abstract nested transactions

Atomic blocks provide a promising simplification to the problem of writing concurrent programs. A code block is marked `atomic` and the compiler and run-time system are responsible for ensuring atomicity during its execution. The programmer no longer needs to worry about manual locking, low-level race conditions or deadlocks.

Atomic blocks are typically implemented over *transactional memory* which provides the abstraction of memory read and write operations which can be grouped together to form a transaction and then committed to the memory as a single atomic step. The book [Larus & Rajwar (2007)] summarizes many of the hardware, software, and hybrid implementation techniques being explored.

In this chapter we highlight a number of ways that the performance of programs can degrade when based on TM. This happens due to *benign conflicts* (Section 2.3) that can occur between transactions. In each of these cases the TM implementation can force one or more transactions to abort because it detects a conflict which does not affect the application logic.

Figure 2.1 (a) shows a running example that we will use. We use a '`g_`' prefix on variable names to indicate that they are shared between threads. Other variables are threadprivate. The example involves two transactions, `Tx-1` and `Tx-2`, which call '`performWork`' on different objects. The function increments a count of the number of times that it has been called and then performs some work on the object that it has been passed. The updates to the shared counter will cause `Tx-1` and `Tx-2` to conflict with one another, leading to the re-execution of a whole `atomic` block even if the bulk of `performWork` is non-conflicting.

We introduce a taxonomy for different variants of this problem in Section 2.3. As we discuss with the taxonomy, there are several ways that programmers can try to avoid problems from benign conflicts. In some cases it is possible to restructure

```
atomic { // Tx-1              atomic { // Tx-2
  performWork(g_o1);            performWork(g_o2);
}                             }

        void performWork(Object o) {
          g_invocation_count ++;
          // Work on 'o'
          work(o);
        }
```

(a) An ordinary implementation of `performWork` introduces conflicts via the global invocation counter.

```
void performWork(Object o) {
  ant {
    g_invocation_count ++;
  }
  // Work on 'o'
  work(o);
}
```

(b) Rewriting `performWork` to use an ANT causes the higher-level update operation to be logged, rather than the low level reads and writes it performs.

Figure 2.1: A benign conflict between Tx-1 and Tx-2.

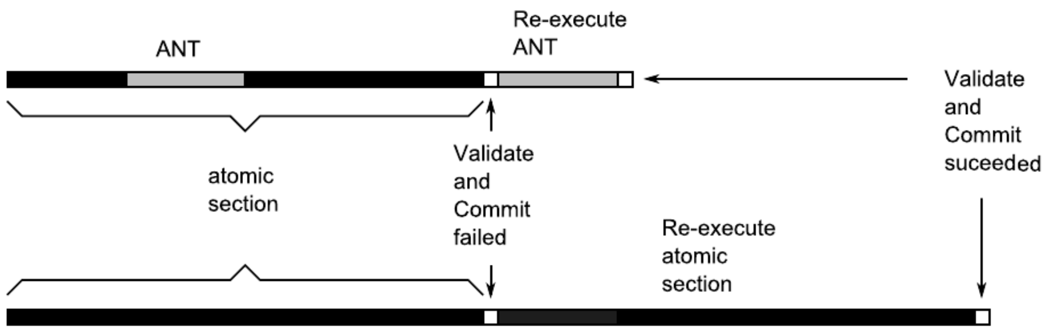

ANT
Re-execute ANT
Validate and Commit suceeded
atomic section
Validate and Commit failed
Re-execute atomic section

Figure 2.2: Execution time of an atomic section with ANT (top) and same atomic section without ANT (bottom).

code, to use *open nested transactions* [Ni *et al.* (2007)], or to use TM-specific optimization interfaces to avoid benign conflicts. All of these approaches have their problems: manually restructuring code can harm the composability benefits of `atomic` blocks, open nested transactions provide a powerful general purpose abstraction but one which relies on programmer care for correct usage, and TM-specific optimization interfaces are hard to use correctly.

In this chapter we introduce a new approach to avoid reexecuting whole `atomic` blocks that contain benign conflict. The idea, which we call *abstract nested transactions* (ANTs), is to identify operations that are likely to be the 'victims' of benign conflicts. This lets the TM implementation keep a separate log of the operations being performed by ANTs and, in the event of a low-level conflict, just re-execute the ANTs rather than re-executing the larger transaction containing them. Figure 2.1 (b) shows our example using an ANT to increment the counter.

Figure 2.2 shows why this can be faster: assuming that the ANT forms a small part of the overall execution time of an `atomic` block, it reduces the amount of work on a conflict.

We discuss the syntax and semantics of ANTs in Section 2.4. A fundamental design principle we took is that they are *semantically transparent*: marking a block of code as an ANT does not affect the possible results of a program. Our motivation in doing this is that it makes ANTs easier to use and, although we show them being used manually in this chapter, in future work they could be placed automatically based on run-time feedback.

This principle guides many aspects of the design and implementation of ANTs: what happens if an ANT conflicts with a transaction that encloses it? What happens if an ANT raises an exception or tries to block using `retry` [Harris *et al.* (2005b)]? What happens if an ANT behaves one way on its first execution and then behaves differently if it is re-executed? We discuss these questions in Section 2.4.

We have prototyped our implementation of ANTs using STM Haskell [Harris *et al.* (2005b)]. In Section 2.5 - 2.6 we describe the details of this implementation and evaluate the performance impact of using ANTs.

In this chapter we make the following contributions:

- An initial taxonomy of *benign conflicts* that can occur between atomic blocks.
- The idea of *abstract nested transactions* as a way of making the performance of atomic blocks more robust to the presence of benign conflict.
- An implementation of ANTs in which they are semantically transparent, being able to be placed around any transactional code and affecting only its performance.
- Specific to our Haskell-based prototype, we describe a new mechanism for comparing possibly-non-terminating computations using lazy evaluation.

Throughout the chapter we assume an implementation over an STM using *lazy conflict detection* [Moore *et al.* (2006a)] that is, detecting conflicts at commit-time in short-running transactions, and periodically in long-running transactions and *lazy versioning* [Moore *et al.* (2006a)] that is, recording tentative updates privately for each transaction, and writing them to the heap upon successful commit.

The reminder of this chapter is organized as follows: in Section 2.3 we define benign conflicts; in Section 2.4 we explain the key ideas behind abstract nested transactions; in Section 2.5 we explain the prototype implementation of ATN; in Section 2.6 we show the experimental results; finally, we conclude in Section 2.7;

```
atomic { // Tx-1              atomic { // Tx-2
  workOn(g_o1);                 workOn(g_o2);
}                             }

           void workOn(Object o) {
             g_temp = o;
             // Work on 'g_temp'
             work(g_temp);
           }
```

Figure 2.3: Shared temporary variables. Transactions Tx-1 and Tx-2 will conflict because they both write to g_temp, even though neither depends on the values written by the other.

2.3 Benign conflicts

In this section we identify a number of different kinds of *benign conflict* that can cause a transaction to abort because of the low-level at which conflicts are typically detected in TM implementations. This is an intentionally imprecise definition; as we illustrate, whether or not a given conflict is benign depends on the context in which the transaction experiencing it occurs. However, distinguishing different kinds of benign conflict helps us identify the cases where ANTs are useful and the cases where they are not.

We focus solely on serializable `atomic` blocks. As the database community has explored, weaker isolation levels can reduce conflicts [Berenson *et al.* (1995)]. However weaker isolation also means that it is no longer possible to reason about `atomic` blocks as executing in isolation from one another; it is unclear whether this would be acceptable as part of a mainstream programming model.

2.3.1 Shared temporary variables

The first kind of benign conflict occurs when global variables are used for transaction local storage. Figure 2.3 shows an example: each transaction starts by writing to `g_temp` and then using it for its own temporary storage. We have seen

```
atomic { // Tx-1              atomic { // Tx-2
  g_obj.x++;                   g_obj.y++;
  // Private work                // Private work
}                             }
```

Figure 2.4: **False sharing.** Transactions Tx-1 and Tx-2 will conflict using the Bartok-STM implementation which detects conflicts on a per-object basis.

this in practice when using the `xlisp` interpreter on Bartok-STM [Harris *et al.* (2006)] and also in the red-black tree implementation that Fraser used in his PhD work [Fraser & Harris (2004)] in which transactions working near the leaves of the tree will write and then read values in a *sentinel node* [Cormen (2001)] whose contents do not need to be retained between operations on the tree.

Fraser provides a mechanism for disabling conflict detection on such data [Fraser & Harris (2004)]; if this is mis-used then transactions may no longer be serializable.

Haskell-STM identifies the special case of *transactionally silent* stores in which a transaction makes a series of updates to a shared field, but the value at the end of the transaction is the same as at the start: the overall access can then be treated as a read rather than a write. This can increase scalability in some cases. However, not all shared temporaries are used in this way.

2.3.2 False sharing

False sharing occurs when the granularity at which TM detects conflicts is coarser than the granularity at which atomic blocks access data.

Figure 2.4 illustrates this with an example of a pair of transactions that conflict when run over the Bartok-STM implementation [Harris *et al.* (2006)]: the two transactions conflict because they both write to fields in the same object. False sharing can also occur in HTMs - for example if conflicts are detected on a per cache-line basis and two transactions update different words on the same cache-line. The paper [Zilles & Rajwar (2007)] analyze the problem of false sharing in TM implementations that use tagless ownership tables, showing that it may happen more frequently than intuition suggests.

```
atomic { // Tx-1                atomic { // Tx-2
  o1 = remove(g_ht, 39);          o2 = remove(g_ht, 49);
  // Work on 'o1'                   // Work on 'o2'
  insert(g_ht, 39, o1);           insert(g_ht, 49, o2);
}                               }
```

Figure 2.5: Using commutative operations with low-level conflicts. Transactions Tx-1 and TX-2 work on objects that they look up from hashtable g_ht under different keys. The hashtable operations may introduce conflicts if the keys hash to the same bucket in the table.

In software, false sharing can be avoided by splitting objects into portions that are likely to be accessed separately - the ability to do this is one motivation for detecting conflicts at an object-granularity because it allows the programmer to control conflicts when deciding which fields to place in the same object.

2.3.3 Tx using commutative operations with low-level conflicts

A further source of benign conflicts occurs when transactions use commutative operations that introduce low-level conflicts. When we say that operations `A` and `B` on a shared object are commutative, we mean that there is no difference, in executing `A`-before-`B` or `B`-before-`A` in terms of the operations' results or the subsequent behavior of the shared object. One example is the shared counter from Figure 1(a): the increment operations are commutative, but the read and writes that they perform are not.

Another example of this kind of benign conflict is lazy initialization: a data item may have its value computed on demand by the first transaction to access it. In many cases the computation can safely be performed more than once , although in other cases this is not true, e.g. in implementations of the singleton design pattern, where a common shared object is being instantiated.

Figure 2.5 shows a more complicated example: two transactions access a shared hashtable (g_ht) and perform operations on different keys. These are likely to conflict in the memory locations that they access if the keys happen to hash to the same bucket in the table.

Ni *et al.* use an example like this to motivate the use of *open-nested transactions* (ONTs) [Ni *et al.* (2007)]. Using ONTs it is possible to prevent `Tx-1` and `Tx-2` from conflicting by (*i*) running the `remove` and `insert` operations in ONTs so that they are performed directly to the hashtable when they are invoked inside transactions, (*ii*) defining compensating operations to roll-back any tentative operations that are made by transactions that subsequently abort, (*iii*) using *abstract-locks* to prevent concurrent transactions performing non-commutative operations on the hashtable - for example insertions under the same key.

Versioned boxes [Cachopo & Rito-Silva (2006)] provide mechanisms for dealing with some kinds of low-level conflict between commutative operations: *delayed computations* that execute at commit time, and *restartable transactions* that perform read-only operations that can be re-executed at commit time to check for benign conflicts. ONTs provide a more general-purpose mechanism to tackle many problems, including this one, but they rely on programmer care in defining the compensating actions and abstract locking discipline in order to ensure that atomic blocks using them remain serializable.

2.3.4 Defining commutative operations with low-level conflicts

A further source of benign conflicts occurs *within* the definition of commutative operations. Figure 2.6 shows an example: two transactions access a sorted linked list of integers, with `Tx-1` searching the list for item 1000 and `Tx-2` inserting item 10. If we assume that the list contains many elements then `Tx-1` will build up a large read-set and conflict with `Tx-2` and any other transactions making updates to the list in the range 1..1000.

ONTs do not provide an obvious solution to this problem: the atomic blocks consist of a single operation on a list, which must be performed atomically whether it is in an ordinary transaction, or in an open one. However, many STM implementations have included 'back doors' by which expert programmers can remove accesses from a transaction's read-set that they believe are unnecessary [Herlihy

```
atomic { // Tx-1              atomic { // Tx-2
  f = listFind(g_l, 1000);      listInsert(g_l, 10);
}                             }

           List listFind(List l, int key) {
             while (l.Next.Key <= key) {
               l = l.Next;
             }
             return l;
           }

           bool listContains(List l, int key) {
             l = listFind(l, key);
             return (l.Key == key);
           }

           void listInsert(List l, int key) {
             l = listFind(l, key);
             if (l.Key != key) {
               l.Next = new List(key, l.Next);
             }
           }
```

Figure 2.6: Defining commutative operations with lowlevel conflicts. Transactions Tx-1 and Tx-2 will conflict in their accesses to the shared list g_l holding sorted integers: Tx-1 will traverse the list up to the node holding 1000, and Tx-2 will conflict with these reads when it inserts a node holding 10.

```
while (true) {                  while (true) {
  atomic { // Tx-1                atomic { // Tx-2
    t = getAny(g_in);               t = getAny(g_in);
    if (t == null) break;           if (t == null) break;
    // Work on t                    // Work on t
    put(g_out, t);                  put(g_out, t);
  }                               }
}                               }
```

Figure 2.7: Making arbitrary choices deterministically. Transactions Tx-1 and Tx-2 both take work items from an input input pool (g_in), work on them, and place the results in an output pool (g_out). A deterministic implementation of getAny will lead them both to pick the same item.

et al. (2003)]. In this case `listFind` could be rewritten to retain only its accesses to nodes in the vicinity of the key: earlier nodes would be removed from the read-set and concurrent updates to these nodes would not be treated as conflicts.

Using these operations correctly requires great care from the programmer. For example, using them here leads to similar search and insert functions to the non-blocking linked list algorithms by Harris [Harris (2001)] and Michael [Michael (2002)]. Furthermore, adding an additional operation to a data structure can make the implementation of existing operations incorrect. For example, if we added `listDeleteFrom` implemented by cutting off the tail of a list at a specified element, then it would no longer be correct to remove elements from the read-set during a call to `listContains`.

2.3.5 Making arbitrary choices deterministically

A final example of benign conflict is caused by *making arbitrary choices deterministically*. Figure 2.7 shows an example. Two threads repeatedly take items from a pool of input items `g_in`, work on them, and place them into an output pool `g_out`. All of the items must be processed, but it does not matter what order this happens in.

If we assume that the input pool is implemented by a shared queue then the two threads' transactions will conflict because they will deterministically select the first item from the queue even though, in this context, any item is acceptable.

ONTs can be used in this case: each thread executes `getAny` in an ONT and uses a compensating action to replace the item. There is one subtlety to beware of in this example - a transaction observing the queue to be empty can only be allowed to commit once there is no possibility of a concurrent call to `getAny` being compensated.

2.3.6 Discussion

In this section we have introduced a number of ways in which programs can experience benign conflicts. There is a wide range of existing techniques addressing parts of this problem space:

- Converting shared temporaries into transactionally-silent stores reduces conflicts using some STM implementations (Section 2.3.1) and restructuring how data is partitioned between objects can reduce false sharing (Section 2.3.2).

- Open nested transactions can be used to avoid serializing commutative operations (Section 2.3.3) and avoid making arbitrary choices deterministically (Section 2.3.5).

- Manual optimization interfaces can be used to trim unnecessary reads from transactions read-sets (Section 2.3.4).

Common to all of these is the use of manual techniques that introduce a risk of changing the behavior of the `atomic` blocks as well as their performance: if they are mis-used then ONTs and read-set reduction interfaces can lead to nonserializable executions of `atomic` blocks.

Our goal is to explore how far we can go with techniques without that risk. The 'abstract nested transactions' in this chapter are the first step in that direction. In particular, we aim to tackle the problems of false conflicts (Section 2.3.2) and atomic blocks using commutative operations with low-level conflicts (Section 2.3.3).

Why do we not tackle the other problems? Essentially because we believe they are best tackled elsewhere. First, it is likely that shared temporary variables can be identified automatically by modifications to the TM implementation. Second, we believe that scalable implementations of data structures involving arbitrary choices can be built over `atomic` blocks and ANTs using randomization techniques similar to those in Scherer's exchanger [Scherer III (2006)] – in effect, making the operations *non-deterministic* where possible. Third, we believe that some cases of read-set reduction are possible by compile-time analyses. Whether or not these techniques are effective is the subject of future work.

2.4 Abstract nested transactions

The key idea with ANTs is to identify operations like false sharing (Section 2.3.2) and commutative operations (Section 2.3.3), which are likely to be the victims of benign conflicts when executed over TM. For example, the hashtable operations in Figure 2.5 could be executed inside ANTs, as could the accesses to `g_obj.x` and `g_obj.y` in Figure 2.6.

We chose the name ANTs because the programmer can think of them as being handled at a different level of abstraction from the ordinary reads and writes that a transaction performs. For example, an `atomic` block that inserts data into a hashtable within an ANT will only be forced to re-execute if a concurrent transaction inserts a conflicting item into the table, rather than (with a typical hashtable implementation) if a concurrent transaction inserts a value that happens to hash to the same bucket.

We detect benign conflicts without programmer annotations about which operations conflict with one another. Instead, we perform extra book-keeping at run-time which lets us (*i*) identify benign conflicts involving ANTs, (*ii*) recover from benign conflicts by just re-executing the ANTs, rather than re-executing the *atomic* block that contains them.

The main difficulty, of course, is ensuring that it is correct to just re-execute the ANTs. We discuss the mechanisms we use to do this in Section 2.5 after first discussing the syntax (Section 2.4.1) and semantics (Section 2.4.2) of ANTs, and programmer guidelines for how to use them to improve performance (Section 2.4.3).

2.4.1 Syntax

The exact way that ANTs are exposed to programmers will depend on the language. In the example in Figure 2.1 (b) we suggested using `ant` blocks. We will use this concise form for examples in the remainder of the chapter, both as stand-alone ANT blocks (`ant{X}`) and as ANT blocks that return a result (`Y = ant{X}`).

In practice, in a language like C# or Java, it would be more natural to express ANTs using an attribute on individual method signatures, or on a class in which case all methods on the class would execute in ANTs. That would be consistent with the expectation that all operations on a given shared object would be performed through ANTs. Of course, many other possibilities can be imagined, such as creating an ANT-wrapper around an existing object, or designating an object as ANT-wrapped at the point that it is instantiated.

2.4.2 Semantics

ANTs are semantically transparent. In our pseudo-code, running `ant{X}` is always equivalent to `X`, no matter what operations are performed in `X` and what context the ANT occurs in. The same is true for ANTs returning a result.

This is an important decision: it means that the addition or removal of ANTs is based purely on performance considerations, making it easier to use feedback-directed tools to identify contention. We did consider whether we could use a different implementation of *closed nested transactions* (CNTs) instead of introducing new notation for ANTs. However, most languages that expose CNTs via nested `atomic` blocks choose to allow exceptions raised inside a CNT to roll-back the nested transaction. This means it is not possible to add or remove CNTs without considering semantic changes to the program.

Another important consequence of our design decision is that an `atomic` block containing ANTs can always be executed in *fallback-mode* in which the ANTs are executed without any special run-time support. We use this idea to simplify our prototype implementation.

2.4.3 Performance

Our implementation of ANTs is based on re-executing ANTs that experience conflicts without needing to re-execute the whole `atomic` block that contains them.

For instance, using the hashtable example from Figure 2.5, if the two keys hash to the same bucket then the second transaction to commit can experience conflicts from the first. However, if the `remove` and `insert` operations are implemented using ANTs then the second transaction will abort its initial execution of `remove` and `insert` and re-execute just these operations rather than needing to reexecute the entire `atomic` block. If the re-execution succeeds without conflict, if the result returned by `remove` is the same upon re-execution, and if this result is the only way that the ANTs interacted with the outer block, then the atomic block and the re-executed ANTs can be committed. Otherwise, the outer transaction is reeccecuted with flat nesting.

The programmer should use the following rules in order to achieve good performance:

- A given piece of data should be consistently accessed inside ANTs, or consistently accessed outside ANTs. Consider the following example:

  ```
  atomic {
    ant { g_temp = remove(g_ht, 39); }
    // Work on 'g_temp'
    work(g_temp);
  }
  ```

  In this example, the ANT interacts with the rest of the `atomic` block through `g_temp`. We detect that ANT is not commutative with the outer transaction, and we restart the outer transaction and execute it in flatten mode (by flattening ANT in outer transaction).

- ANTs should constitute a small portion of the execution time of the atomic block that contains them. Otherwise, there is no practical gain by re-executing just the ANTs.
- The programmer should mark the commutative sections of the code with ANT transactions.

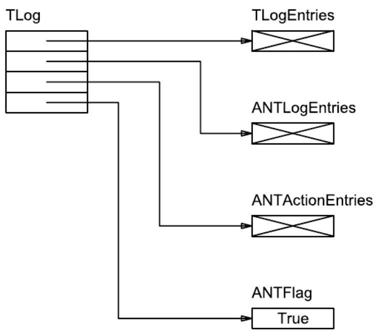
2.5 Prototype implementation

We have implemented support for ANTs in the *run time system* (RTS) of the Glasgow Haskell Compiler (GHC). Aside from the subtlety discussed in Section 2.5.3 about detecting equality between Haskell values, the implementation should be applicable to other languages using similar STM algorithms. Although STM-Haskell does not expose nested `atomic` blocks to the programmer, our implementation of ANTs *does* support closed nested transactions which are used internally by the GHC RTS in its implementation of exception handling and the `orElse` and `retry` constructs for composable blocking [Harris *et al.* (2005b)].

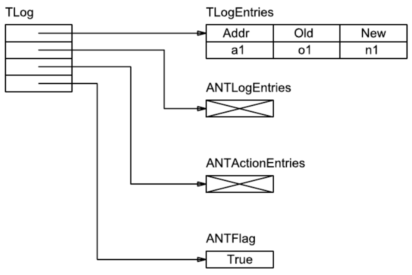
GHC's existing support for STM uses lazy conflict detection and lazy versioning using transaction logs that keep track of shared memory accesses. Each transaction's log is a list of log entries containing the following fields: the *memory address* being accessed, the *old value* that the transaction expects to be stored there and the *new value* that it wants to write there. Every read or write to a shared memory location is performed first by scanning through the transaction log and, if the location is not found in the log, the memory access is performed and a new log entry is created in the log.

2.5.1 Changes when executing an atomic block

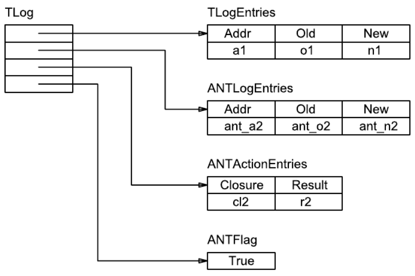
While executing an `atomic` block we differentiate between memory accesses made from within ANTs and those made from the enclosing transaction. To achieve this, the transaction log keeps entries in two separate lists: one list records the accesses in the ANTs, and the other list records all the normal transactional accesses.



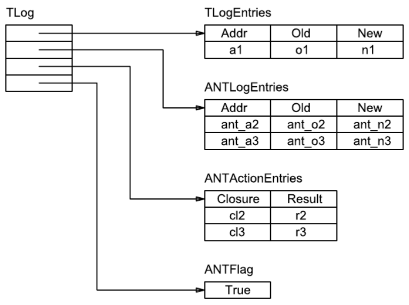
(a) A transaction starts with an empty transaction log.



(b) The transaction writes address a1.



(c) The first ANT finishes, having made updates to address ant_a2 and returned result r2.



(d) The second ANT finishes.

Figure 2.8: Transaction execution with ANTs enabled.

The structure of the transaction log (TLog) can be seen on the Figure 2.8 (a). The TLog has four fields. TLogEntries holds the normal transactional accesses. ANTLogEntries holds the accesses made within ANTs. ANTActionEntries records the high-level operations being performed by ANTs. Each is represented by a pair of pointers. The first points to the block of code (closure) that executes the ANT. The second points to the result that was returned by the closure when the ANT was first executed. ANTFlag is used to implement fallbackmode: if the flag is set to True, then ANTs are enabled and logging is done using the ANTLogEntries list. If the flag is set to False, then ANTs are disabled and all logging is done to the TLogEntries list.

We will show, on a small example, how these logs are used. The following example is written in Java-like pseudocode:

```
1. atomic {
2.    a1 = <LARGE COMPUTATION>;
3.    r2 = ant { <SMALL COMPUTATION>; }
4.    <LARGE COMPUTATION>;
5.    r3 = ant { <SMALL COMPUTATION>; }
6.    <LARGE COMPUTATION>;
7. }
```

In Figure 2.8, we can see the structure of the TLog at different stages during the execution of the atomic block. The atomic block starts a transaction by creating an empty TLog (after executing line 1). TLogEntries, ANTLogEntries and ANTActionEntries are empty in the beginning (Figure 2.8 (a)). Line 2 modifies variable `a1` with a tentative change to the TLogEntries (Figure 2.8 (b)). In line 3, the ANT uses the ANTLogEntries for its tentative update: the ANT modifies variable `a2` and the change is logged in the ANTLogEntries. After the execution of the ANT, the pointer to the ANT's code and its result are saved in the ANTActionEntries (Figure 2.8 (c)). Line 4 reverts to using TLogEntries for logging. In the example case, no access to the other transaction variables occured. The ANT in the line 5 changes the variable `a3` and uses the same the ANTLogEntries slot that was used by the ANT in line 3. The return value and the closure are once again saved in ANTActionEntries.

2.5.2 Changes when committing an atomic block

Ordinarily, at the end of an `atomic` block, the GHC RTS implementation of STM validates the transaction log and commits the updates to memory. The following algorithm explains how we modify the commit phase of a transaction:

```
start:
  case validate(TLog):
    OK : commit TLog
    ANTLogEntries and TLogEntries intersect:
      set ANTFlag = False
      restart transaction
    ANTLogEntries invalid, TLogEntries valid:
      re-execute ANTActionEntries
      goto start
    TLogEntries invalid:
      restart transaction
```

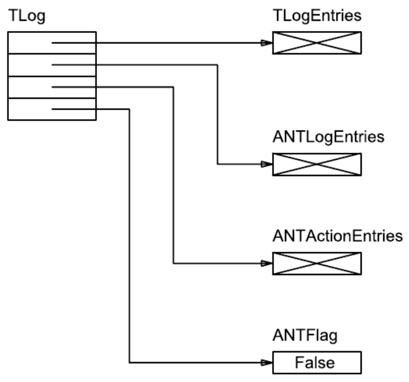
This algorithm starts by validating the entire TLog, comprising the entries in TLogEntries and ANTLogEntries. There are four cases to consider:

1. *TLogEntries and ANTLogEntries are all valid.* This means that there have been no conflicts at all: not with the ANTs or with the remainder of the `atomic` block. In this case we commit all the log entries to memory.

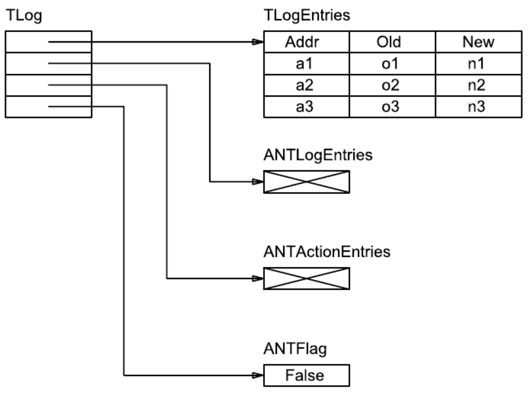
2. *TLogEntries and ANTLogEntries intersect.* This occurs when a program uses the same transactional variable in the ANT and outside of it. The whole atomic block has to be re-executed with the ANTFlag set to False, disabling ANT usage. There is no scalability gained from the ANTs, but the semantics of the atomic section are preserved.

   Figure 2.9 shows what happens in our example. The transaction is restarted (Figure 2.9 (a)) and builds up a single log during its re-execution (Figure 2.9 (b)).

3. *ANTLogEntries are invalid and TLogEntries are valid.* This shows that there has been a conflicting memory access on one or more of the ANTs.



(a) Transaction re-executes



Addr	Old	New
a1	o1	n1
a2	o2	n2
a3	o3	n3

(b) Re-executed transaction finishes

Figure 2.9: Transaction re-execution with ANTs disabled (ANTFlag set to False).

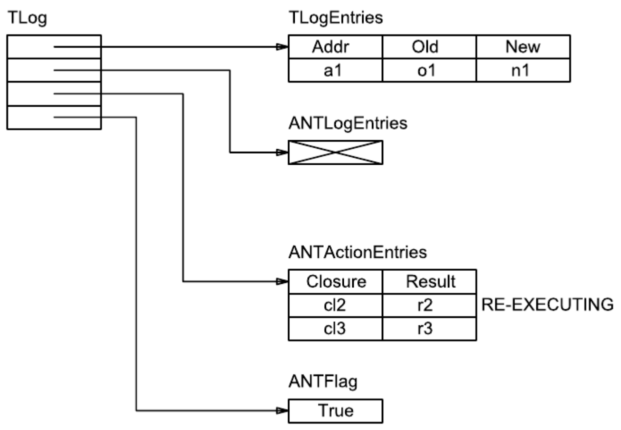


Figure 2.10: Transaction about to re-execute ANTs.

This is the case when we can re-execute all the closures that are in the ANTActionEntries. After re-execution of every closure, the new return values from the closures are compared with the return values of the previous execution. If all the values are the *same* (implementation of equality is explained in Section 2.5.3), we know that any computation in the atomic block that depends on the those values will be unaffected by the re-execution of the closures. If any of the return values has changed then the atomic block has to be re-executed.

Figure 2.10 shows what happens in our example. The ANTLogEntries structure is emptied and each ANT from the ANTActionEntries is re-executed in turn. After successful re-execution of these closures the TLog will once again look as shown on Figure 2.8 (d).

4. *TLogEntries are invalid.* In this case there has been a conflict with the main transaction: we must re-execute the whole `atomic` block.

2.5.3 Implementing equality in RTS

There is a subtle problem in how we implement the equality test between different executions of an ANT. There are two factors to consider. First, for safety, we must err on the side of caution: two results can be claimed distinct when in fact they are equivalent. Second, our design principle that ANTs are semantically transparent means that the implementation of the equality test must not change the semantics of the `atomic` block. This means that we cannot generally use programmer-supplied equality tests unless we wish to trust these to be correct: in C# or Java terminology we would use '`==`' rather than '`.Equals`'.

However, working in Haskell raises another problem: the language uses lazy evaluation and so the result from an ANT may be returned as an un-evaluated closure rather than as a result which can be compared. We cannot simply evaluate the closure in case it is a non-terminating computation that is not needed by the program.

In practice we have not seen closures occurring in this way and so our prototype conservatively uses (*i*) pointer equality between objects with identity (e.g. mutable variables whose addresses can be compared), (*ii*) a shallow comparison function between objects without identity (e.g. boxed integer values). However, in a full implementation we could perform equality tests between a first result `R1` and a second result `R2` as follows:

- If `R1` and `R2` are both objects with identity then use pointer equality.
- If `R1` and `R2` are both objects without identity then recursively compare their constructor tags and fields.
- If `R1` has been evaluated but `R2` has not, then evaluate `R2` and repeat the comparison. This deals with the case where the `atomic` block has forced `R1` to be evaluated and may therefore depend on its result. If `R2` does not complete evaluation promptly then abandon it and reexecute the atomic block in fallback-mode.
- If R1 has not been evaluated *then* `R1` *and* `R2` *can be treated as equal.* The key insight is that if `R1` was not evaluated then the `atomic` block cannot depend on the (still-unknown) value it may yield.

  However, there is one further caveat in this case: the atomic block may itself return `R1` or store it into shared memory when it commits. In this case we must replace `R1` with `R2` so that, if it is ever evaluated, the commit-time result R2 is obtained. This can be done in the GHC RTS by atomically overwriting R1 with an indirection to R2. The paper [Harris *et al.* (2005a)] provides an introduction to the management of closures, indirections, and so on in the RTS.

- In other cases treat R1 and R2 as distinct.

2.6 Results

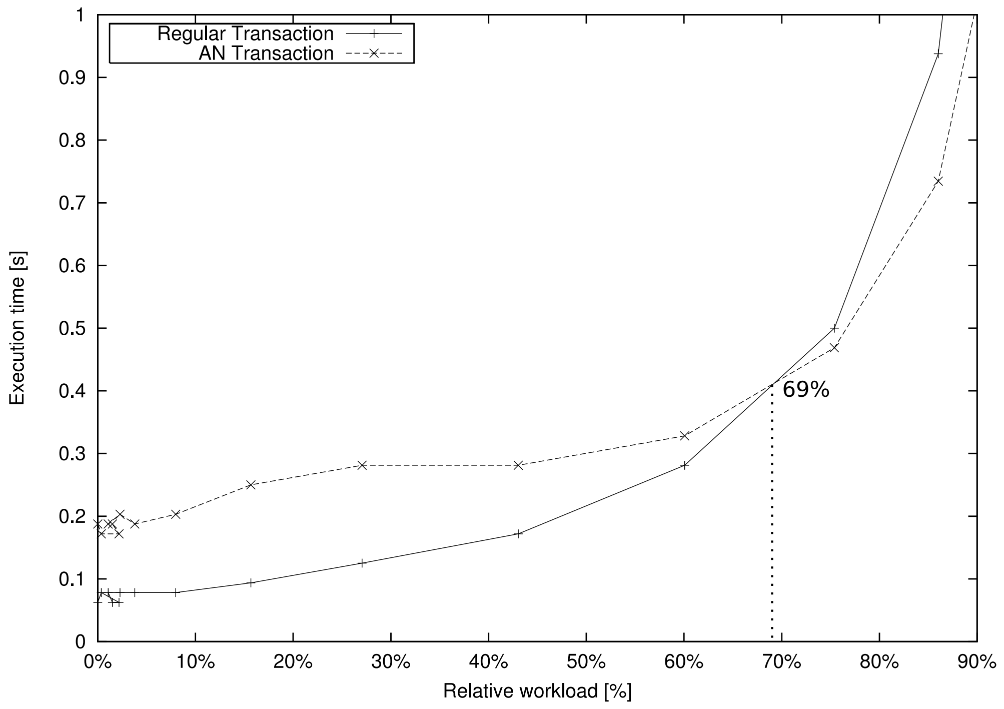
To explore the performance of ANTs we used a test program with the following structure:

```
atomic {
  v = ant { <SMALL COMPUTATION>; } // A1
  <LARGE COMPUTATION>; // L1
  ant { <SMALL COMPUTATION>; } // A2
}
```

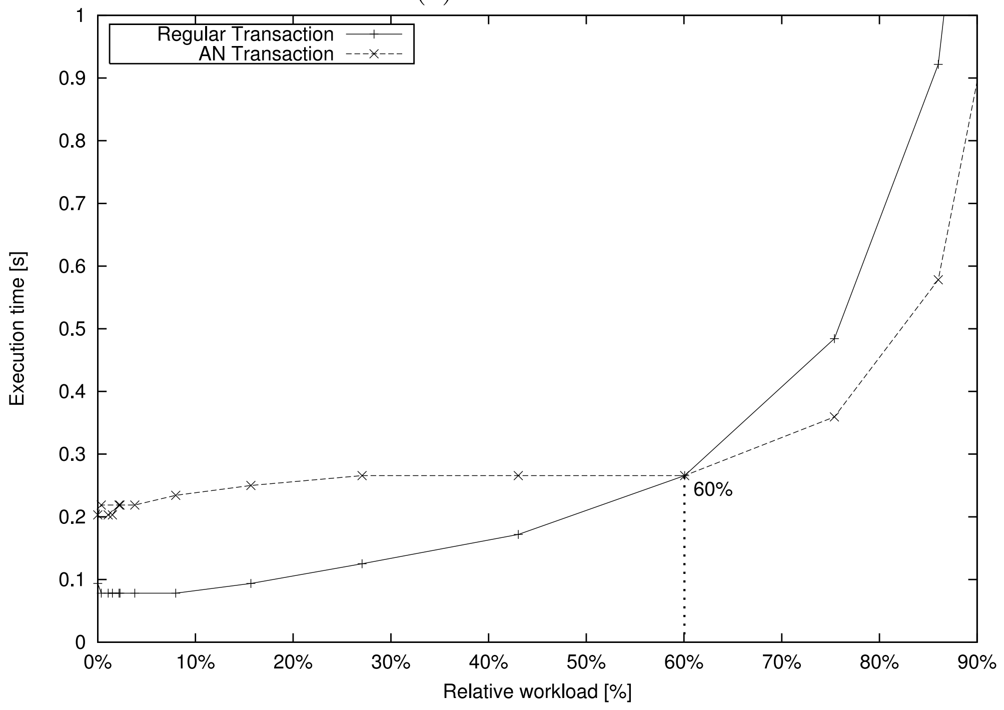
Our test lets us vary the amount of time spent inside the ANTs by varying the amount of work performed in the large computation L1. As we discussed in Section 2.4.3, ANTs should be used for small parts of the transaction that are likely to conflict with other transactions and so we need to quantify what this means.

In our test program the ANT A1 removes an item from a shared linked list of key-value pairs and A2 returns an item to the head of the list. Each thread works on disjoint keys: concurrent invocations will always conflict in their reads and writes to the list, but the operations themselves are commutative. For the <LARGE COMPUTATION>, we uses a simple function performing a private loop of fixed duration. We vary the number of threads operating on the list and the ratio of the large computation's execution time to the ANTs'.

Our test machine ran Windows Server 2003, with 2 quadcore Intel Xeon CPUs (in total 8 cores) and 4GB of RAM. All the tests were compiled with optimizations and ran with GHC's heap configured to 512MB of heap so that garbage collection did not play any role in the execution times.



(a) 4 threads



(b) 8 threads

Figure 2.11: The execution time of the program plotted while varying the relative workload using a list of 32 elements. A low relative workload means that most of the test program is spent inside ANTs. A high relative workload means that most of the test program is spent outside ANTs.

Figure 2.11 shows the execution time of the test program with 4-thread and 8-thread runs as the size of the large computation is varied. The graph compares the performance of the test using ANTs ('AN Transaction') against the performance with ANTs disabled ('Regular Transaction'). The x axis shows the 'relative workload' which is the fraction of execution time spent *outside* ANTs.

As we discussed in Section 2.4.3 we would expect this to be high in the intended uses of ANTs. The implementation using ANTs out-performs the existing implementation when nested transactions account for 69% or more of the execution time. The reason for this is that the re-execution time of regular transaction is larger than re-execution time of the transaction with ANTs; the regular transaction has to re-execute the whole `atomic` section, and on the other hand, the transaction with ANTs has to re-execute just ANTs. Of course, by making the size of the ANTs increasingly small, the performance difference could be made arbitrarily good. However, the important result is to understand the kind of range below which ANTs are ineffective.

Conversely, when most time is spent in non-transactional code, we can see that the use of ANTs slow down the program (for small `atomic` sections, the slowdown can be around 2x). This is because most of the execution time of the transaction is spent in `stmCommitTransaction()` and our prototype effectively introduces two passes over the log entries; one to distinguish the four cases in Section 2.5.2 and another to actually commit the changes to memory.

In Figure 2.12 we compare the performance of our test program using ANTs with the same test program using ONTs in the case where around 8% of time is spent inside the ANTs. Both ANTs and ONTs improve scalability compared with executing all the operations in a single transaction. One would expect ONTs to out-perform ANTs in this workload: no compensating actions are run and so the work performed by ONTs is strictly less than ANTs.

2.7 Summary

This chapter has introduced the idea of *abstract nested transactions* (ANTs) for identifying sections of an `atomic` block that are likely to be the victims of benign conflicts. By re-executing ANTs we can avoid re-executing the whole

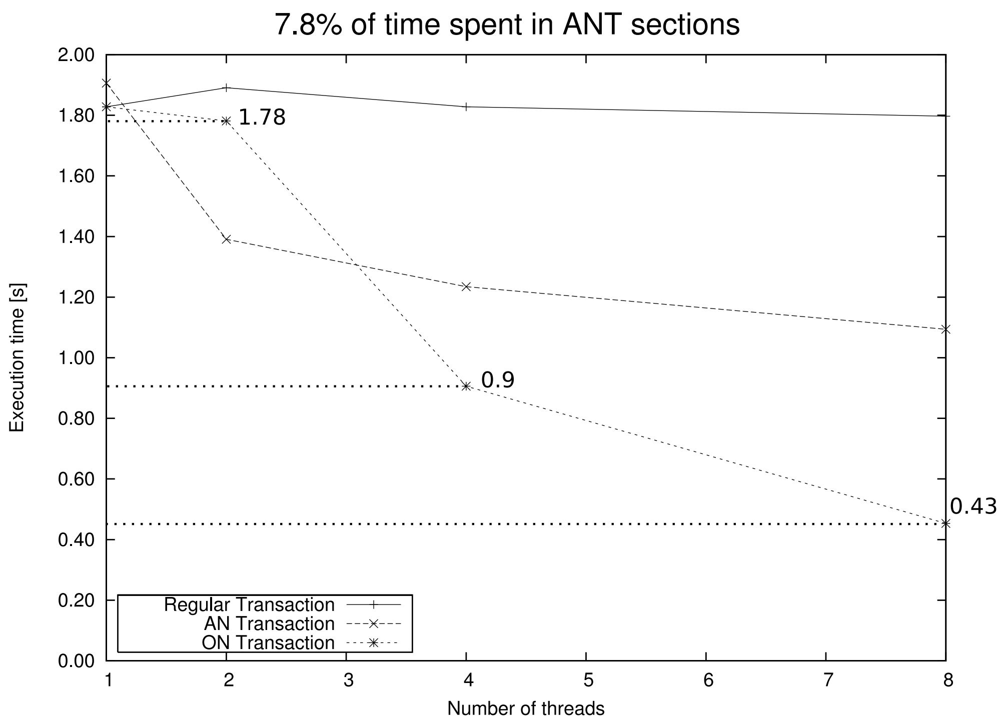


Figure 2.12: Execution times of regular transactions, ANTs and ONTs.

atomic block that contains them. Unlike other techniques for improving the scalability of `atomic` blocks ANTs are semantically transparent and can be used as a performance tuning technique without risk of changing the semantics or serializability of the code in which they are used.

This is not to say that they provide a replacement for other abstractions such as low-level unsafe optimization interfaces or open nested transactions. However, the unique features of ANTs open the possibility for completely automatic feedback-directed optimization of transactional programs to try to identify contention hot-spots.

We started working on ANT believing that it could be widely used in STM systems but we were not able to find a lot of use cases of ANT in real world TM applications. Simple explanation why ANT does not work in practise is that ANT tries to minimize transaction re-execution time after the conflict happened. In other words, ANT helps when the TM system experiences a lot of aborts. So, if TM system experiences a lot of aborts then we can assume that system/program performs badly. In this case, the best way to improve the performance of the program is is to rewrite it in in order to minimize the possible aborts. So we believe that ATN is an evolutionary dead end.

Chapter 3

TagTM

The content of this chapter was presented in the paper “TagTM - accelerating STMs with hardware tags for fast meta-data access” at DATE 2012 (Design, Automation and Test in Europe).

3.1 Introduction to TagTM

In this chapter we introduce TagTM, a Software Transactional Memory (STM) system augmented with a new hardware mechanism called GTags. In this chapter we introduce GTags as a new hardware cache coherent tags that are used for fast meta-data access. TagTM uses GTags to reduce the cost associated with accesses to the transactional data and corresponding metadata. For the evaluation of TagTM, we use the STAMP TM benchmark suite. In the average case TagTM provides a speedup of 7-15% (across all STAMP applications), and in the best case shows up to 52% speedup of committed transaction execution time (for SSCA2 application).

The main contributions of this chapter are:

- We propose a hardware extension (GTags) that stores transactional meta-data.
- We show how GTags allow (*i*) accessing both the data and its metadata with a single memory operation instead of two and (*ii*) fetching both the

New ISA instructions	Description
ldt r1 $\leftarrow$ T[r2]	Load tag
stt T[r1] $\leftarrow$ r2	Store tag
cast T[r1] $\leftarrow$ r2 if T[r1] == r3	Compare tag and store tag
ldtv r2 $\leftarrow$ T[r1], r3 $\leftarrow$ M[r1]	Load tag and value
sttv T[r1] $\leftarrow$ r2, M[r1] $\leftarrow$ r3	Store tag and value
castv T[r1] $\leftarrow$ r2, M[r1] $\leftarrow$ r3 if T[r1] == r4	Compare tag and store tag and value

Table 3.1: GTags manipulation instructions

data and the metadata together thus eliminating the cache misses due to poor spatial locality.

- We show how GTags improve the performance of STM systems.

The rest of this chapter is organized as follows: We describe our extensions which we call *Global Tags (GTags)* in Section 3.2. GTags extend the computers' memory hierarchy with coherent metadata tags and a set of instructions to operate them. In Section 3.3 we introduce TagTM, an extension of STM library with GTags. We use GTags for accelerating the maintenance of the transactional metadata. In Section 3.4 we present an evaluation of TagTM, in Section 3.5 we discuss the related work, and in Section 3.6 we conclude.

3.2 Global Tags (GTags)

In this section we introduce global tags (GTags), a mechanism for annotating transactional data at cache line granularity across the different levels of the memory hierarchy. In our design GTags extend every cache line with 64-bit tags. These tags are associated with the data in the cache lines and kept always coherent. When a cache line is invalidated, its tag is also invalidated and vice versa.

GTags are accessed and modified with the special tag manipulation instructions which are shown in Table 3.1. Furthermore, all instructions in the table are executed atomically. All the instructions that operate on the addresses that fall in the same cache line share the same tag.

Extending memory hierarchy with GTags requires small architectural changes which affect the CPU caches and the memory controller. In our particular case, we store GTags at the data part of the cache lines. Such a design decision allows reading and writing the GTags with simple load/store like instructions and does not require any changes to the otherwise complicated cache lookup logic.

In the main memory, we store the tags separately from the data in special GTag-pages. The GTag-pages are allocated by operating system. This allocation enables easy mapping of the physical address to the corresponding tag. The memory controller does a simple shift operation and adds an offset to calculate the physical address of the corresponding tag. In our simulations, we model a memory controller which can automatically fetch and store both the data and the tag from their corresponding pages (Figure 3.1). With such functionality, the tags are preserved when the cache lines and the tags are evicted from the cache.

3.3 TagTM

In this section we introduce TagTM, a modified version of TinySTM. The goal of this section is to demonstrate how to use GTags to improve the performance of an STM library. In Section 3.3.1 we explain how TinySTM works and in Section 3.3.2 we describe the bottlenecks of the implementation of TinySTM and later we show how to extend transactional operations with GTags.

3.3.1 TinySTM

Felber et al. proposed TinySTM [Felber *et al.* (2007)], a lightweight and efficient word-based STM system. TinySTM implements timestamp-based versioning algorithm. It utilizes a shared array of locks (SAL) to manage concurrent accesses to memory. Each lock covers a portion of the address space. The least significant bit of the lock indicates if the lock is owned, and the remaining bits of the lock store the version number that corresponds to the commit timestamp of the transaction that last updated the memory location covered by the lock.[1]

[1]For the full implementation, please refer to the original paper.

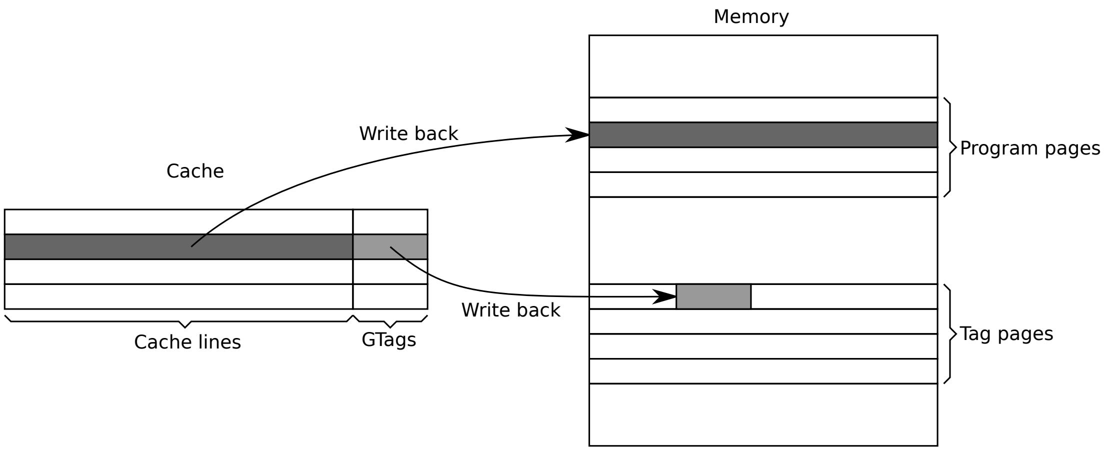


Figure 3.1: Cache line eviction

3.3.2 Bottlenecks in TinySTM

Figure 3.2 shows the breakdown of transactional overheads for STAMP applications using TinySTM executing with one thread. The slowdown is divided in 4 parts: `tx_read`, `tx_validate`, `tx_commit`, and other(`tx_write` and `tx_start`). It is important to note that all these overheads do not exist in non-transactional execution. `tx_read` is a dominant overhead for Genome, K-means, Vacation, and Yada. The total transactional overhead can be up to 69.82% (in the case of Vacation-high) compared to the non-transactional execution time. `tx_commit` is a dominant overhead for Intruder and SSCA2. In SSCA2, `tx_commit` and `tx_validate` adds up to 200.4% and 98.59% transactional overhead compared to the non-transactional execution time.

3.3.3 Using GTags in TinySTM

In Section 3.3.2 we show that the performance-critical operations in TinySTM are `tx_read`, `tx_commit`, and `tx_validate`. All these operations can benefit from the use of GTags. GTags and timestamp-based versioning STMs are a natural fit because the SAL can be stored in the tags, next to the actual data. Because of the tight coupling of the tags and the data in the cache lines, the lock access will force the inclusion of the memory location to the cache. This acts as "free"

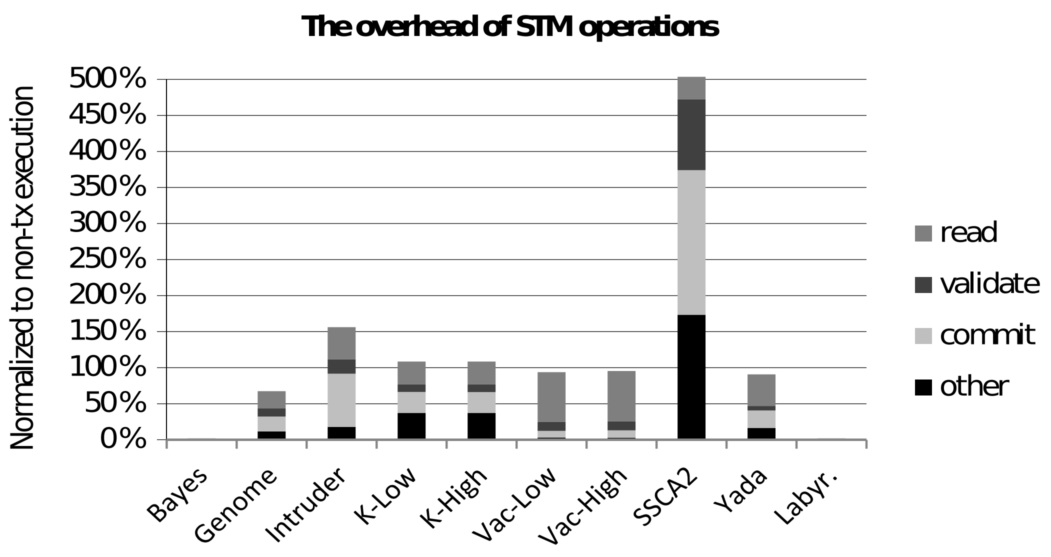
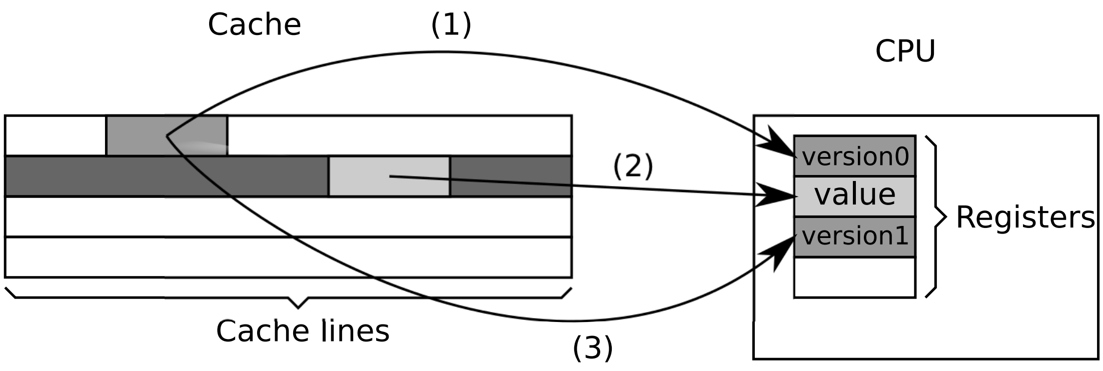


Figure 3.2: STM overheads running one CPU.

prefetching of the corresponding memory location that will be used in the transaction. This improves the performance of `tx_read` operation that is on the critical path of the transactional execution (`tx_read` always reads the lock from the SAL to get the version number). With GTags, the combined write-back of the tag and the new memory value saves one write to the memory per write-set entry. As we demonstrate later, this can provide big improvements of the `tx_commit` operation in large transactions. `tx_validate` also benefits from GTags by having better cache locality. In the following sections we explain in detail how to extend TinySTM with GTags.

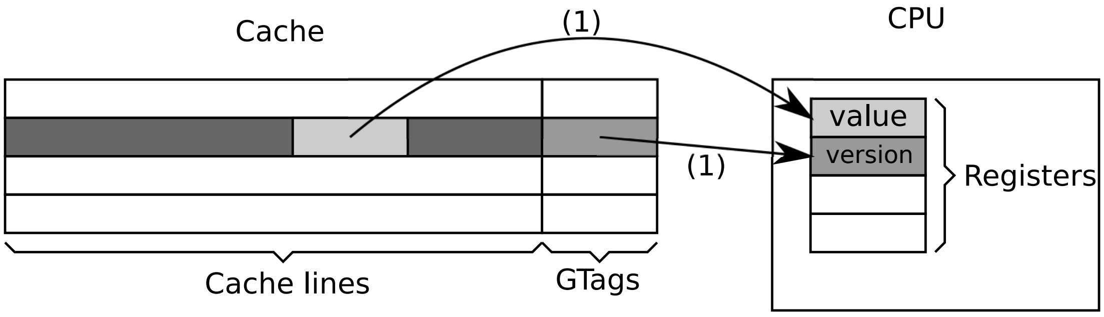
3.3.4 Improving the tx_read operation

In typical lazy versioning TM systems, an unmodified `tx_read` operation consists of three phases: (*i*) the query of the write-set phase, (*ii*) the address and version read phase, and (*iii*) the read-set update phase. In Figure 3.3, we can see the pseudo code for the `tx_read` operation while the address and version read phase of `tx_read` is represented explicitly in the code, and graphically. The compiler cannot remove redundant memory read operations, because the exact ordering of



```
     tx_read(addr) {
       <write_set_query>
     read_value_start:
(1)    version0 = load_lock(&addr);
(2)    value = *addr;
(3)    version1 = load_lock(&addr);
       if (version0 != version1) { goto read_value_start;}
       <read_set_update>
       return value; }
```

Figure 3.3: Tx_read in lazy versioning STM.



```
     tx_read(addr) {
       <write_set_query>
(1)    load_tag_and_value(addr, &version, &value);
       <read_set_update>
       return value; }
```

Figure 3.4: Tx_read in TagTM.

the read instructions is necessary for the correct execution of `tx_read`.

To improve the performance of the `tx_read` operation, we use GTags' *load tag and value* instruction (Table 3.1) to combine the read of the memory address and the read of the corresponding lock. This improves the performance of `tx_read` by reducing the number of cache accesses in the address and version read phase, which reduces the number of cycles spent waiting for the memory subsytem. The use of GTags in `tx_read` is presented in Figure 3.4 where the single *load tag and value* instruction is executed to load the address value and version, instead of 3 separate instructions. This simplifies the `tx_read` operation and improves its performance. Because `tx_read` operation is on the critical path in STMs [Zyulkyarov *et al.* (2010b)], this has positive impact to the application's performance. In Section 3.4, we show the reduction in the number of executed cycles for the second phase of `tx_read`.

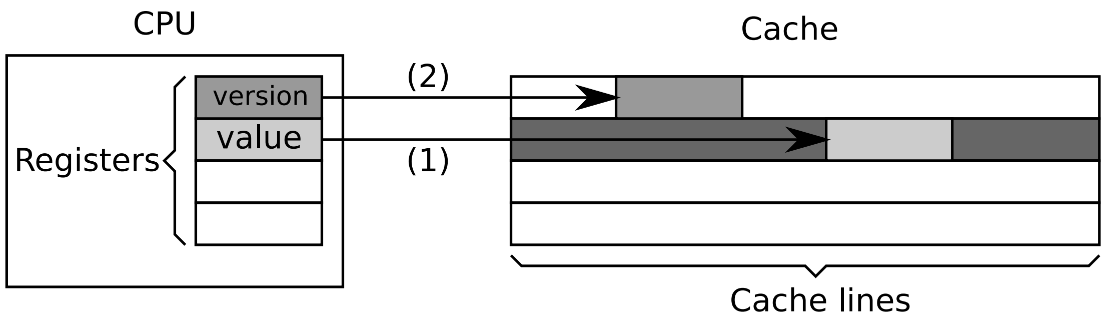
3.3.5 Improving the tx_commit operation

In typical lazy versioning TM systems, unmodified `tx_commit` consists of three phases: (*i*) the lock acquisition phase, (*ii*) the validation phase, and (*iii*) the write-back[1] phase (Figure 3.5). The write-back phase of `tx_commit` is represented explicitly in the code, and graphically.

To improve the performance of `tx_commit` operation, we use GTags' *store tag and value* instruction to combine the write to the memory address and the write to the corresponding lock. This improves the performance of the write-back phase by reducing the number of executed instructions, and the number of updated cache lines, and by releasing the lock earlier and thus publishing the results sooner.

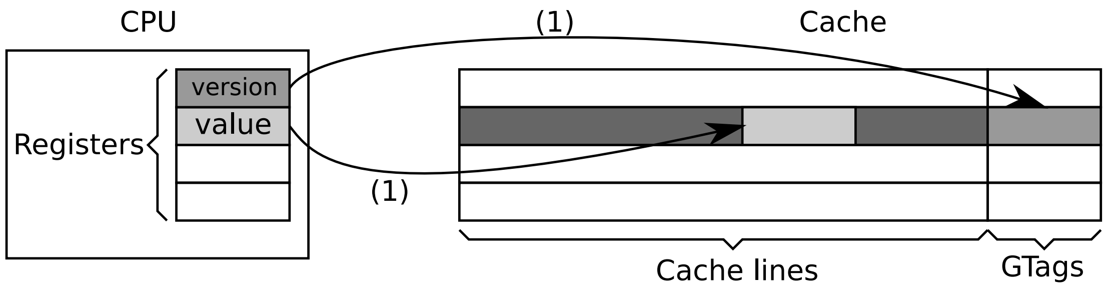
The use of GTags in `tx_commit` is presented in Figure 3.6. The code for the lock acquisition phase and for the validation phase is the same in both versions of the code. The difference exists in the write-back phase. In this phase, the original `tx_commit` operation without GTags has to execute two memory writes per write-set entry, one for the memory update and other for the lock release

[1]The write-back phase includes the release of the acquired locks from the lock acquisition phase.



```
   tx_commit() {
     try_acquire_locks();
     tx_validate();
     for (ws_entry in write_set) {
(1)    *ws_entry.addr = ws_entry.value;  // update memory
(2)    *ws_entry.lock_addr = new_version;  // lock release }}
```

Figure 3.5: Tx_commit in lazy versioning STM.



```
   tx_commit() {
     try_acquire_tags();
     tx_validate();
     for (ws_entry in write_set) {
       // update memory and release lock
(1)    store_tag_and_value(ws_entry.addr, new_version, ws_entry.value); }}
```

Figure 3.6: Tx_commit in TagTM.

operation[1]. `tx_commit` with GTags can execute just one *store tag and value* instruction that will update the memory reference and will release the lock. Because the `tx_commit` operation is on the critical path in STMs, this has positive impact to the application's performance. In Section 3.4, we show the reduction in the number of cycles for the write-back phase.

3.3.6 Modifying remaining transactional operations

The implementation of `tx_validate`, `tx_write`, and `tx_abort` operations is the same in TinySTM and in TagTM, with one small difference. In original TinySTM, these operations access the locks from the SAL, and in TagTM, these operations access the locks stored in the cache line tags. GTags improve the performance of `tx_validate` operation indirectly, by having better cache locality than original TinySTM. In Section 3.4, we show the reduction in a number of executed cycles the for the validation phase.

3.4 Evaluation

For the evaluation we use the M5 full system simulator [Binkert *et al.* (2006)]. The bus-based coherency protocol is replaced with directory-based MESI cache coherence protocol. We use in order DECAlpha CPU cores extended with the new instructions for tag manipulation. We extend the cache lines to store tags and extend the cache coherence protocol to make data and tags cache coherent. The configuration parameters used for the simulation are shown on Table 3.2.

Minh et al. created STAMP [Cao Minh *et al.* (2008)], a state-of-the-art benchmark suite for evaluating TM systems. We use STAMP to compare the unmodified TinySTM against TagTM. We use the recommended input parameters for the application in STAMP for simulation run [Cao Minh *et al.* (2008)].

[1]The release also updates the version number.

Feature	Description
Processors	1 to 32 DECAlpha cores, in-order, single-issue
L1 Cache	64KB, private, 4-way assoc., 64B line, 2-cycle access
Coherence protocol	MESI protocol
L2 cache	8MB, shared, 32-way assoc., 64B line, 16-cycle access
Memory	300-cycle off chip access
GTags	Every cache line is extended with 64-bit tag.

Table 3.2: The simulation parameters.

3.4.1 Transactional operations performance improvements

Figures 3.7, 3.8, and 3.9 show the time spent in the memory hierarchy for the address and version read, the write-back, and the validation phases in STAMP benchmarks respectively. The X axis of the graph shows the benchmarks with TinySTM and TagTM, interleaved. The Y axis depicts the time that is normalized to TinySTM. The cycles spent for memory accesses are broken down in three parts for L1, L2, and main memory accesses.

Figure 3.7 suggests that GTags reduce the number of memory accesses in the "address and version read" phase of `tx_read` operation because of the improved spatial locality of data and tags. Vacation and Bayes benchmarks show the biggest time reduction (up to 41.93% for Vacation-low) because GTags successfully eliminate the accesses to main memory. The other applications show a modest improvement of execution time, which is attributed to the reduction in the number of accesses to the L2 cache.

Figure 3.8 suggests that GTags reduce the number of memory accesses in the write-back phase of `tx_commit` operation because of the improved spacial locality of the data and tags. Almost all the benchmarks show reduction in accesses to main memory and to L2 caches. This presents a large performance improvement of 58.96% (geometric mean of all the applications) for the write-back phase. For GTags, there is slight increase in the time spent for the accesses of L1 cache because in the simulation we add one cycle latency to the L1 hit latency when GTags access data and meta-data with same instruction.

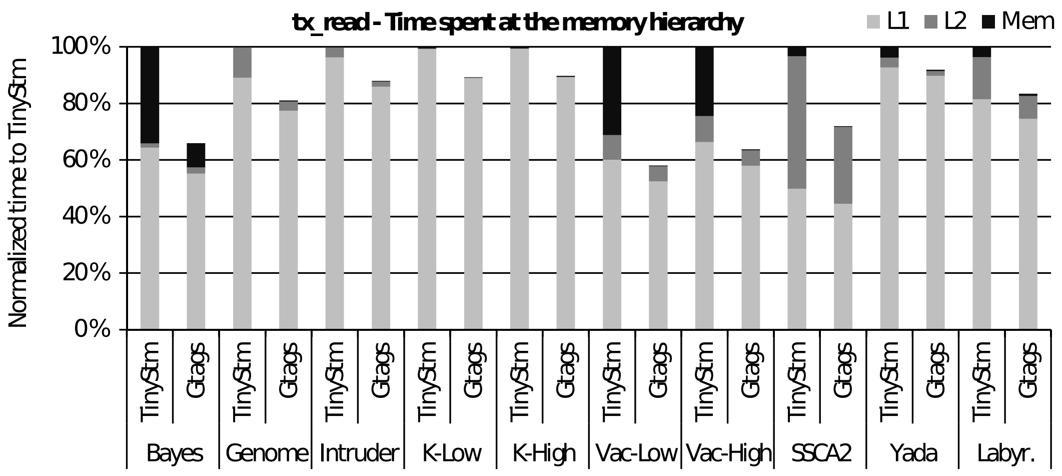


Figure 3.7: `tx_read` - Time spent in the memory hierarchy. Time is normalized to TinySTM.

Figure 3.9 suggests that GTags reduce the number of memory accesses in the "validation" phase of `tx_commit` operation because the number of accesses to main memory is greatly reduced. The validation routine has to traverse all the locks that are stored in the transaction's read-set during short time interval which will populate the caches with the locks from the SAL. This will kick out some of the transactional data from the caches to the main memory. TagTM does not exhibit this problem because the locks stored in the tags do not compete with transactional data for the caches, therefore GTags effectively increase the associativity of the caches for transactional applications. The performance benefit of GTags for the validation is 26.76% for the applications with small and medium read-sets (geometric mean for Kmeans-high, Kmeans-low, Vadation-high, and Vacation-low). The benefit is bigger for the applications with medium and large read-sets and is 85.55% (geometric mean for the rest of the STAMP applications).

3.4.2 Transaction execution performance improvements

Figure 3.10 shows the speedup of TagTM over TinySTM. The X axis shows STAMP applications running from 1 up to 32 threads. The Y axis shows the speedup gain provided by GTags. The right most columns on the graph repre-

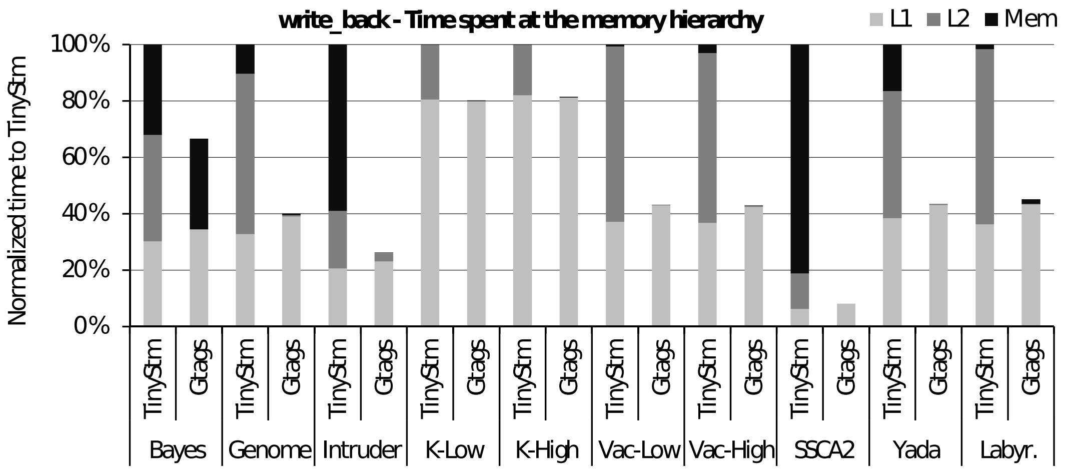


Figure 3.8: `write_back` - Time spent in the memory hierarchy. Time is normalized to TinySTM.

sent geometric mean of the speedup of the applications. Three applications from STAMP (Kmenans, Vacation, and SSCA2) show similar behaviour when running on different number of CPUs. All of them have more or less constant performance improvement while the number of CPUs change (except Kmeans on 32 CPUs). SSCA2 shows the best speedup (of 52%). This improvement comes from the fact that SSCA2 has the biggest transactional overhead of 503.52% (Figure 3.2). Kmeans (low and high) and Vacation (low and high) have similar transactional overheads and exhibit the similar speedup with GTags of about 13.28% (geometric mean). Intruder has a performance gain of about 17.2% while running from 1 to 8 threads, and has a performance loss for 16 and 32 threads. The rest of transactional applications (Genome, Bayes, Yada, and Labyrinth) have small transactional overhead, and show that the benefit of GTags can be positive (for Bayes and Yada) or negative (for Labyrinth). The performance benefit happens when the (transactional) data and the lock are accessed in transaction. In the applications with small transaction overhead, the performance loss happens when the non-transactional data pays the price of fetching tags that will not be used. This is the one of the reasons why Labyrinth experiences performance degradation. Overall, we have an average speedup of 13.05% in committed transaction

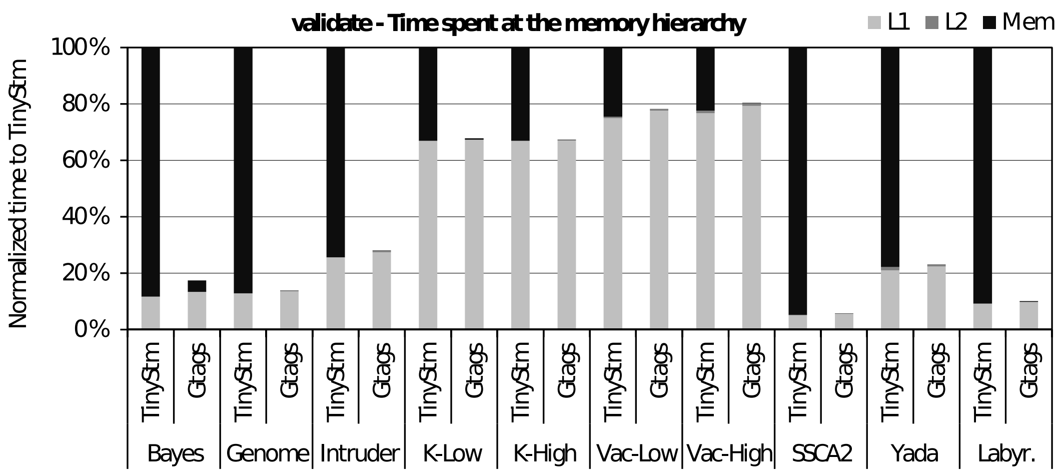


Figure 3.9: `validate` - Time spent in the memory hierarchy. Time is normalized to TinySTM.

Cache type	**Regular L1**	**L1 extended with GTags**
Size	64KB	64KB + 8KB for GTags
Cache line size	64 bytes	64 bytes + 8 bytes for GTags
Other	4-way, 1 bank, 45nm	4-way, 1 bank, 45nm
Total area (mm2)	0.841	0.892 (6.05% increase)

Table 3.3: Cache parameters calculated with CACTI 5.3.

execution time while running on 1-32 cores.

3.4.3 GTags - L1 cache overhead

We use CACTI 5.3 [Shivakumar & Jouppi (2001)] to evaluate the increase in L1 cache area caused by adding GTags to the data part of the cache lines (see Table 3.3). GTags increase the L1 cache area by 6.05%. This is a small overhead compared to the total cache size, because GTags utilize the already existing lookup logic.

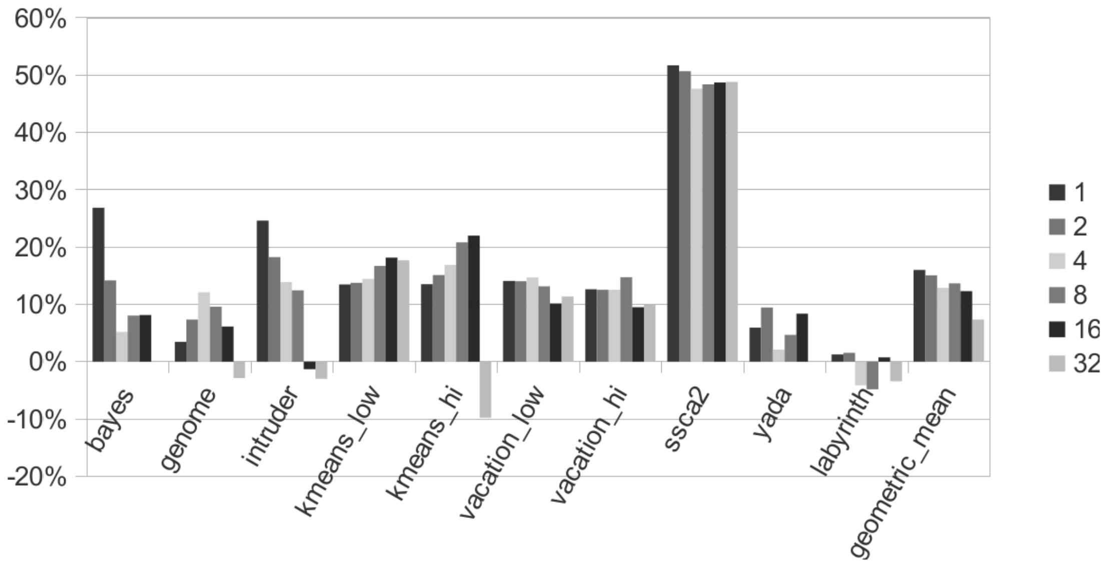


Figure 3.10: Percentage speedup of TagTM over TinySTM

3.5 Related Work

Saha et al. proposed HaSTM [Saha *et al.* (2006c)] for improving the performance of STM systems by using additional bits per cache line. Every 16 bytes of an L1 cache line is protected with 1 bit. HaSTM uses these bits to eliminate redundant logging and to eliminate validation overhead altogether for transactions whose transaction records fit in the cache. In the case of GTags, the tags speed-up the logging by reducing the time necessary to accesses to global version table.

Ming el al. proposed SigTM [Minh *et al.* (2007b)] that uses hardware signatures to track the read-set and write-set for pending transactions and to perform conflict detection between concurrent threads. All other transactional functionality, including data versioning, is implemented in software. However, the biggest performance benefit of SigTM is the elimination of read-set logging. GTagTM do not have the problem of false aborts because the version information stored in tags is exact.

Hammond et al. introduced Transactional Memory Coherence and Consistency (TCC) [Hammond *et al.* (2004b)] to execute transactions in hardware. Their system changes the coherence hardware and require that all the code executes simple transactions. On the other hand, GTags requires no changes to existing coherence protocols and can be used to speedup existing software TM systems.

Harris et al. proposed Dynamic Filtering (DF) [Harris *et al.* (2010)], a multipurpose architecture support for language runtime systems, to speedup software TM systems and to reduce the overheads in language base security systems. DF reduces the metadata accesses in eager STM systems by reducing unnecessary transactional logging. GTags reduce the metadata accesses by reducing the overheads associated with accesses to cache-line locks/timestamps.

Adl-Tabatabai et al. designed compiler and runtime support for software TM [Adl-Tabatabai *et al.* (2006b)] which is able to reduce the overheads of STM. Their implementation should benefit from the uses of GTags because, atomic GTags' instructions can enable some further optimizations that would reduce the STM overheads even more.

Baugh et al. [Baugh *et al.* (2008)] used fine-grained protection mechanism to isolate transactional data in an implementation of TM with strong atomicity, and to separate hardware-managed and software-managed transactional data in hybrid systems. This approach could be used to extend use of GTags for hardware HTM systems.

Several other proposals add hardware for fast meta-data acesses and application monitoring. Zeldovich et al. [Zeldovich *et al.* (2008)] implementad Loki, a tagged memory architecture, to enforce appilcation security policies in hardware. Venkataramani et al. [Venkataramani *et al.* (2007)] propose hardware support for memory access monitoring tasks. Chen et al. [Chen *et al.* (2008)] use hardware extensions to accelerate metadata accesses.

3.6 Summary

In this chapter we introduced TagTM, a software TM system augmented with new hardware mechanism that we call GTags. GTags are new hardware cache coherent tags used for fast meta-data access. TagTM use GTags to reduce the cost associated with accesses to the transactional data and corresponding metadata. For the evaluation of TagTM, we used STAMP benchmark suite. In the average case TagTM provide the speedup of 7-15% (across all STAMP applications), and in the best case shows up to 52% speedup of committed transaction execution time (for SSCA2).

Chapter 4

Profile-Guided Transaction Coalescing

The content of this chapter was presented in the paper "Profile-guided transaction coalescing – lowering transactional overheads by merging transactions" at TACO 2014 (ACM Transactions on Architecture and Code Optimization).

4.1 Introduction to Profile-Guided Transaction Coalescing

Previous studies in Software Transactional Memory mostly focused on reducing the overhead of transactional read and write operations. In this chapter we introduce *transaction coalescing*, a profile-guided compiler optimization technique that attempts to reduce the overheads of starting and committing a transaction by merging two or more small transactions. We develop a profiling tool and a transaction coalescing heuristic to identify candidate transactions suitable for coalescing. We implement a compiler extension to automatically merge the candidate transactions at compile-time. We evaluate the effectiveness of our technique using the hash table micro-benchmark and the STAMP benchmark suite. Transaction coalescing improves the performance of the hash table significantly and the performance of Vacation and SSCA2 benchmarks by 19.4% and 36.4% respectively, when running with 12 threads.

4.2 Motivation for Profile-Guided Transaction Coalescing

Most of the current TM systems are implemented in software as a library or as a compiler extension. In order to support concurrent execution of transactions, a software TM (STM) system manages transactional metadata during transactional execution. This creates overheads related to transaction initialization, versioning, read/write instrumentation, etc. At the beginning of each transaction, the STM system initializes transactional metadata (read and write sets) and creates a checkpoint of the current thread state (the register file and local variables). If the transaction aborts, due to a conflict with another transaction, the checkpoint is used to restore the state from the beginning of the transaction. Otherwise, it commits changed shared data and associated metadata. If transactions in an application are numerous and small, STM overheads accumulate rapidly. The overheads degrade performance since most of the execution time is spent on creating and committing transactions - in extreme cases even more than 70% [Chung *et al.* (2008)].

Until now researchers have been trying to decrease STM overheads by applying various compiler and runtime optimizations on read and write barriers [Afek *et al.* (2011); Dragojevic *et al.* (2009); Wang *et al.* (2007); Wu *et al.* (2009)], or by accelerating some STM functionalities in hardware [Minh *et al.* (2007a); Saha *et al.* (2006b); Shriraman *et al.* (2007); Stipic *et al.* (2012)]. In our work, we follow a different approach and focus on reducing the overheads of starting and committing transactions.

This chapter makes the following contributions:

- We introduce *transaction coalescing* (TC), a profile-guided compiler technique that lowers the overheads of starting and committing transactions. TC merges two or more consecutive transactions into a large transaction that has less transactional overhead compared to the original unmerged transactions (we use ‘merging’ and ‘coalescing’ interchangeably). To the best of our knowledge, this is the first profile-guided compiler optimization technique for STM systems.

- We develop a profiling tool for collecting and analysing characteristics of a transactional application.
- We design a transaction coalescing heuristic that identifies candidate transactions suitable for merging.
- We implement TC as a profile-guided compiler optimization that uses profile information and the transaction coalescing heuristic to merge the candidate transactions.

We evaluate our proposal using the STAMP [Minh *et al.* (2008)], red-black tree, and hash table benchmarks where we show profiling information for all the benchmarks. Using the profile-guided optimization we identify the benchmarks that benefit from TC, and for these benchmarks we show full statistics and performance improvement gained by applying TC. TC improves the performance of hash table significantly and the performance of Vacation and SSCA2 benchmarks by 19.4% and 36.4% respectively, when running with 12 threads.

The reminder of this chapter is organized as follows: in Section 4.3 we present transactional overheads as motivation of our work; in Section 4.4 we introduce transaction coalescing, a novel profile-guided compiler technique; in Section 4.5 we present our profiling tool, transaction coalescing heuristic, and explain how a compiler uses the heuristic for transaction coalescing; in Section 5.5 we show experimental results; finally, we conclude in Section 4.7.

4.3 Transactional overheads

A programmer writes a TM application by identifying sections of code for atomic execution isolated from other threads, and puts the code in `tm_atomic` blocks, i.e. transactions. The compiler and a runtime system ensure atomicity and isolation of transactions. At the beginning of a transaction, the compiler inserts a `TX_START()` function call. `TX_START()` initializes the transactional metadata and creates the snapshot of the register file and local variables. The compiler instruments all read and write operations that access shared memory locations. The functions `TX_READ()` and `TX_WRITE()` keep track of all speculative reads

```
while (!stop) {
 tm_atomic {
  for (i = 0; i < NUM_ACCOUNTS; i++) {
    accounts[i].balance *= interest_rates[i];
  }
}
```

(a) transactional source code

```
while (!stop) {
 TX_START();
  for (i = 0; i < NUM_ACCOUNTS; i++) {
   tmp1 = TX_READ(interest_rates[i]);
   tmp2 = TX_READ(accounts[i].balance);
   tmp3 = tmp1 * tmp2;
   TX_WRITE(accounts[i].balance, tmp3);
  }
 TX_COMMIT();
}
```

(b) compiler-instrumented code

Figure 4.1: Calculating account balance.

and writes. At the end of the transaction, the compiler inserts a `TX_COMMIT()` function call to make speculative writes globally visible.

In order to determine the overheads of `TX_START()` and `TX_COMMIT()` functions in TinySTM 1.0.3 [Felber *et al.* (2007)], we implement a very simple *interest rate calculation* benchmark[1] (Figure 4.1.a) where each thread calculates account balance for several accounts within a transaction denoted with `tm_atomic`. Figure 4.1.b shows the same code after transactional instrumentation. To measure the throughput, we use the environment described in Section 5.5, and we vary the `NUM_ACCOUNTS` parameter from 1 to 256 and the number of threads from 1 to 12 (Figure 4.2). We observe the following:

- As the `NUM_ACCOUNTS` parameter increases, the throughput increases as well. This happens because the total time spent in `TX_START()` and `TX_COMMIT()` decreases compared to the time spent inside the transaction.

- Increasing the transaction size is beneficial up to some limit. In our example, the throughput is very simmilar for `NUM_ACCOUNTS = 64` and `NUM_ACCOUNTS = 256`. The further increase of the `NUM_ACCOUNTS` parameter gives negligible performance improvements because the time spent in `TX_START()` and `TX_COMMIT()` is small w.r.t. the transaction size.

- The biggest jump in the throughput is for the small values of `NUM_ACCOUNTS` parameter (e.g., `NUM_ACCOUNTS = 16`). This shows that reducing the `TX_START()`

[1]The benchmark resembles the main loop in transaction processing applications.

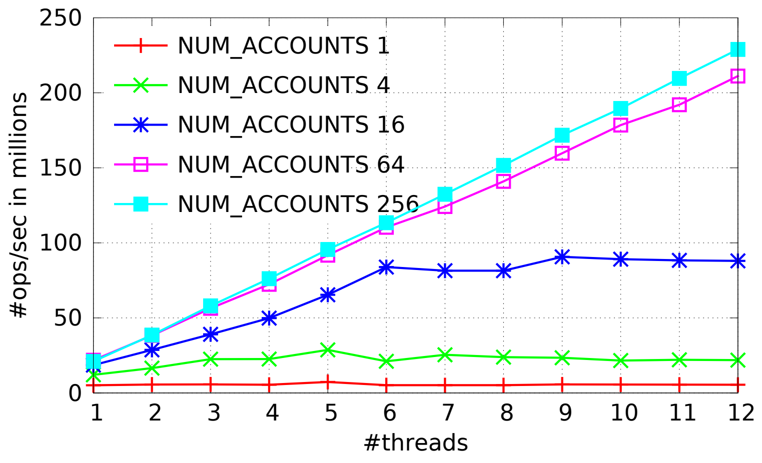


Figure 4.2: Throughput for the code from Figure 4.1. 'NUM_ACCOUNTS' represents the number of balance calculations in a transaction. '#ops/sec' represent the number of balance calculations per second.

and `TX_COMMIT()` overheads has a big performance improvement for small transactions.

- For small transactions (`NUM_ACCOUNTS <= 16`), the throughput is limited when transactions execute on different physical sockets[1] (even though the threads operate on independent data). The throughput is limited because STM serializes transactions at commit time by updating the shared global "commit counter", and small transactions finish before the "commit counter" gets in the exclusive state in CPU's private L1 cache. For larger transactions (`NUM_ACCOUNTS > 16`), the transactions execution time is larger than time required to get the "commit counter" in exclusive state (the transaction execution is overlapped with internal STM synchronization.)

From this simple example, we conclude that increasing the size of transactions up to some limit reduces `TX_START()` and `TX_COMMIT()` overheads. The new insight is that we have to pay the price of transaction initialization just once

[1]The throughput is limited to 6 cores because we use two socket machine where each socket has 6 cores on chip

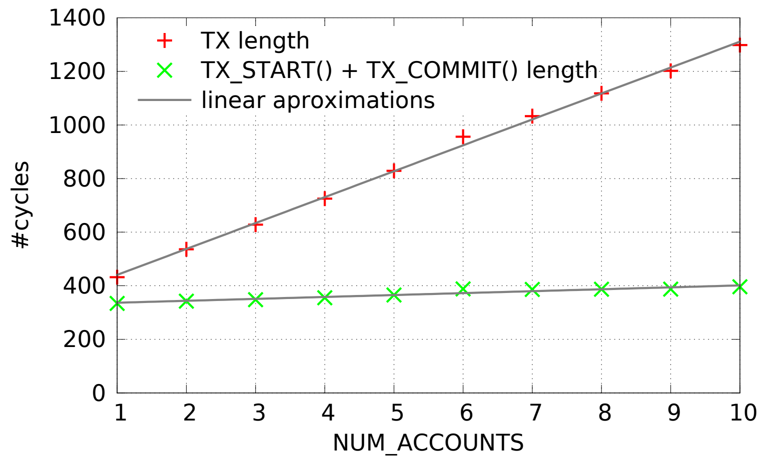


Figure 4.3: Execution time in cycles for the code from Figure 4.1 when running with 1 thread.

if we join two or more transactions. So, we remove some of the cost of the transaction initialization.

In order to quantify the transactional overheads of `TX_START()` and `TX_COMMIT()`, we execute the interest rate calculation benchmark with 1 running thread. Figure 4.3 shows that `TX_START()` and `TX_COMMIT()` consume the largest part of a transaction when `NUM_ACCOUNTS = 1` (334 out of 432 cycles, which is 77.3%[1]). Increasing the `NUM_ACCOUNTS` decreases the percentage of the transactional overheads compared to the whole transaction execution. Therefore, we conclude that increasing the number of read and write operations (in a transaction) amortizes the overheads of `TX_START()` and `TX_COMMIT`.

4.4 Transaction coalescing

In this chapter we introduce *transaction coalescing* (TC), a compiler technique that merges small transactions with large transactional overhead into a large transaction with lower transactional overhead. TC merges two or more consecutive transactions or transactions separated by non-transactional code.

[1]The transactional overhead depends on the specifics of the STM implementation.

In Section 4.3 we showed that for small transactions increasing the number of operations per transaction reduces the transactional overhead of `TX_START()` and `TX_COMMIT()`. In this section, we provide a few use cases for applying TC and reducing the transactional overhead: (i) transactions are consecutive, (ii) a compiler applies traditional compiler optimizations (loop unrolling and function inlining), and (iii) transactions are separated by non-transactional code.

Figure 4.4.a shows an example of incrementing two counters in two transactions. The compiler transforms the `tm_atomic` blocks and instruments the code inside. TC merges two consecutive transactions by removing `TX_COMMIT()` from the first transaction and `TX_START()` from the second transaction. This reduces the overall overhead associated with the transactional execution.

Figure 4.4.b illustrates an example where the compiler uses TC after loop unrolling. The `for` loop atomically increments each element of the array, and each iteration of the loop pays the overhead for `TX_START()` and `TX_COMMIT()`. After the loop unrolling, two consecutive increments can be coalesced in one large transaction with lower transactional overhead.

Figure 4.4.c demonstrates how TC can be combined with function inlining. The figure shows two transactions in two separate functions (`increment()` and `decrement()`). After inlining these functions, two transactions become consecutive, and TC can be applied. TC optimization pass depends on unrolling and inlining similarly as e.g. modulo scheduling (aka. software pipelining) depends on loop unrolling (modulo scheduling requires loop unrolling pass)

Up to now, we have considered the ideal cases when transactions are consecutive. However, a more likely case is that a piece of code separates transactions. For example, Figure 4.4.d shows typical operations with a shared list: the function `work_with_list()` atomically pops an element from the list, processes the element outside of `tm_atomic` blocks, and atomically pushes the processed element to the list. The compiler inlines the functions `get_elem()` and `put_elem()` and now TC creates one transaction with `pop()`, `push()`, and a transactional version of `process()`. The instrumentation of the `process()` function introduces an additional overhead. Therefore, to find transactions for merging, we need to analyze both the code within transactions (because of the benefit of TC) and the code between transactions (because of the disadvantages of TC).

transactional source code

```
tm_atomic {
  counter1++;
}


tm_atomic {
  counter2++;
}
```

after compiler transformations

```
TX_START();
tmp1 = TX_READ(counter1);
tmp1++;
TX_WRITE(counter1, tmp1);
TX_COMMIT();

TX_START();
tmp2 = TX_READ(counter2);
tmp2++;
TX_WRITE(counter2, tmp2);
TX_COMMIT();
```

after applying TC

```
TX_START();
tmp1 = TX_READ(counter1);
tmp1++;
TX_WRITE(counter1, tmp1);
// no TX_COMMIT();

// no TX_START();
tmp2 = TX_READ(counter2);
tmp2++;
TX_WRITE(counter2, tmp2);
TX_COMMIT();
```

(a) The basic example of TC

```
for(i=0; i<1000; i++){
  tm_atomic {
    a[i]++;
  }
}
```

```
for(i=0; i<1000; i+=2){
  tm_atomic {
    a[i]++;
  }
  tm_atomic {
    a[i+1]++;
  }
}
```

```
for(i=0; i<1000; i+=2){
  tm_atomic {
    a[i]++;
    a[i+1]++;
  }
}
```

(b) Loop unrolling and TC

```
void update_counters(){
  increment();
  decrement();
}
inline void increment(){
  tm_atomic {
    counter1++;
  }
}
inline void decrement(){
  tm_atomic {
    counter2--;
  }
}
```

```
void update_counters(){
  tm_atomic {
    counter1++;
  }
  tm_atomic {
    counter2--;
  }
}
```

```
void update_counters(){
  tm_atomic {
    counter1++;
    counter2--;
  }
}
```

(c) Function inlining and TC

```
void work_with_list(){
  int el = get_elem();
  process(el);
  put_elem(el);
}

int get_elem(){
  int el;
  tm_atomic {
    el = list.pop();
  }
  return el;
}
void put_elem(int el){
  tm_atomic {
    list.push(el);
  }
}
```

```
void work_with_list(){
  int el;
  tm_atomic {
    el = list.pop();
  }
  process(el);
  tm_atomic {
    list.push(el);
  }
}
```

```
void work_with_list(){
  int el;
  tm_atomic {
    el = list.pop();
    process(el);
    list.push(el);
  }
}
```

(d) Code between transactions included in a merged transaction

Figure 4.4: Compiler transformations: instrumentation and TC (a), optimizations and TC (b),(c),(d). In all examples the compiler inserts `TX_START()`, `TX_COMMIT()`, `TX_READ()`, `TX_WRITE()` calls like in (a); however we do not show them in (b), (c), (d) for clarity of code.

4.5 Applying TC

In this section, we introduce our profiling tool, transaction coalescing heuristic, and compiler pass. The tool generates profile information, the heuristic uses the profile information to find candidate transactions, and the compiler pass merges the candidate transactions.

4.5.1 Profiling tool

We developed a profiling tool for finding transactions that benefit from TC. The tool executes an unmodified program, and gathers statistics from a single-threaded execution. The tool outputs the following information:

- **Transaction execution information.** For every transaction, the tool collects: (i) the number of cycles executed in the transaction, (ii) the percentage of the total execution time spent in the transaction, and (iii) the sizes of read and write sets. The tool uses the transaction coalescing heuristic to identify small and frequently executed transactions that are candidates for TC.

- **Transaction transition information.** For every pair of transactions the tool calculates a *transaction transition matrix* and a *transaction distance matrix*. The transaction transition matrix counts the number of transitions[1] between transactions, and the transaction distance matrix calculates the average number of cycles executed between transactions. The tool uses the transaction coalescing heuristic to find transactions that are close to each other during the execution.

To use the profiling tool, we compile and run STAMP benchmarks (using the input parameters from Table 4.1) and two micro-benchmarks (hash table and red-black tree). We show the transaction execution information in Table 4.2 and the transaction transition information in Table 4.3 and Table 4.4. Although some

[1]Each transaction has a unique static transaction ID. The transition is a pair of unique static transactions IDs where the first ID represents the ID of the committed transaction and the second ID represents the ID of the next committed transaction executed by the same thread.

	benchmark input arguments
Bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1
Genome	-g16384 -s64 -n16777216
Intruder	-a10 -l128 -n262144 -s1
Kmeans	-m40 -n40 -t0.00001 -i rnd-n65536-d32-c16
Labyrinth	-i rnd-x128-y128-z5-n128.txt
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3
Vacation	-n4 -q60 -u90 -r1048576 -t4194304
Yada	-a15 -i ttimeu1000000.2

Table 4.1: STAMP benchmark input parameters.

benchmarks have more than 3 transactions, we present only the transactions that occupy most of the benchmark execution time, because the reset of the transactions execute less than 5% of the execution time and are not candidates for coalescing.

4.5.2 Transaction Coalescing Heuristic

The transaction coalescing heuristic uses profile generated statistics to identify candidate transactions suitable for merging. The heuristic looks for transactions that are *small*, *frequent* and *close* to each other, by using the following rules:

- A *small transaction* is a transaction that executes in less than 3,000 cycles on average[1] and has less than 16 and 4 elements in read and write sets[2], respectively.
- A *frequently executed transaction* is a transaction that consumes at least 10% of the execution time of the program.
- *Close transactions* are a pair of transactions that have the transition probability higher than 80% and have less than 1,000 cycles between each other on average[3].

[1]We choose 3,000 cycles as the upper limit because in this case TX_START() and TX_COMMIT() create overhead of more than 10% of transaction execution time (Section 4.3 shows that the overhead is about 300 cycles).

[2] These sizes of read and write sets work well in practice.

[3] The upper limit of 1,000 cycles between transactions is determined empirically.

	tx1	tx2	tx3	tx1	tx2	tx3
benchmark	**Bayes**			**Labyrinth**		
length (cycles)	908,161,356	536,427,523	406,716,287	110,363,661	262,308	2,735
execution time	46.31%	27.36%	20.74%	99.74%	0.24%	<0.1%
read set size	11509169.82	6465018.77	5057245.93	2751731.61	7.96	2.00
write set size	2344162.23	1309395.11	1025313.10	335355.95	0.99	4.00
benchmark	**Genome**			**Rb tree**		
length (cycles)	20,789	14,640	960	980	2,016	2,153
execution time	65.42%	33.88%	<0.1%	90.11%	1.6%	0.68%
read set size	186.01	137.10	3.83	29.82	31.75	38.86
write set size	12.07	4.00	3.78	0.00	7.00	9.00
benchmark	**Hash table**			**SSCA2**		
length (cycles)	333	664	921	1,511	14,836,229	6,386
execution time	78.53%	1.12%	0.86%	59.31%	<0.1%	<0.1%
read set size	2.50	3.00	4.00	5.00	1.00	1.00
write set size	0.00	1.66	4.00	2.00	1.00	1.00
benchmark	**Intruder**			**Vacation**		
length (cycles)	14,933	662	579	1,742	6,595	1,897
execution time	81.23%	3.6%	3.15%	71.01%	15.0%	4.27%
read set size	163.84	6.00	4.13	9.21	84.90	9.00
write set size	13.85	1.00	1.13	1.00	6.99	1.00
benchmark	**Kmeans**			**Yada**		
length (cycles)	6,199	465	937	379,700	1,633	1,224
execution time	47.46%	1.19%	<0.1%	98.66%	0.58%	0.32%
read set size	130.00	1.00	1.00	7264.59	14.98	11.70
write set size	33.00	1.00	1.00	1468.79	2.20	1.54

Table 4.2: Transaction execution information. The *length* is the average number of cycles spent in a transaction. The *execution time* of a transaction is the average percentage of total execution time spent in the benchmark. The *read/write set size* is the average size of the read/write sets of a transaction. Gray cells show small transactions, dark gray show frequent transactions, light gray cells show transactions with small read and write sets, and circled cells show candidate transactions for TC.

Ht.	tx1	tx2	tx3
tx1	98.76%	0.85%	0.38%
tx2	92.97%	-	7.03%
tx3	100%	-	-

Vac.	tx1	tx2	tx3
tx1	89.98%	5.02%	4.99%
tx2	89.98%	5.18%	4.83%
tx3	90.53%	4.81%	4.66%

SSCA2	tx1	tx2	tx3
tx1	99.99%	-	<0.01%
tx2	-	-	-
tx3	<0.01%	-	99.99%

Table 4.3: Transaction transition matrix: the probability of transitions between transactions. For candidate transactions 'tx1' from Table 4.2, the dominant transition is to the transactions 'tx1' (circled cells).

Ht.	tx1	tx2	tx3
tx1	81	97	90
tx2	110	-	112
tx3	109	-	-

Vac.	tx1	tx2	tx3
tx1	219	222	230
tx2	214	220	231
tx3	178	177	189

SSCA2	tx1	tx2	tx3
tx1	886	-	1.9G
tx2	-	-	-
tx3	138K	-	74

Table 4.4: Transaction distance matrix: the average number of cycles between transactions. Circled cells confirm candidate transactions suitable for TC.

The information from Table 4.2 is crucial to find small and frequently executed transactions that are candidates for TC. The heuristic identifies that hash table, Vacation, and SSCA2 have candidate transactions suitable for TC (circled cells in the table). For example, the transaction 'tx1' of Vacation executes in 1,742 cycles and consumes 71.01% of the total execution time. However, transactions not suitable for TC are either (i) large transactions (e.g., 'tx1' in Bayes), or (ii) infrequently executed transactions (e.g., 'tx2' in Intruder).

Table 4.3 shows the *transaction transition matrix* for the three benchmarks (hash table, Vacation, and SSCA2) where TC can be applied. For all these benchmarks, the dominant transition is from 'tx1' to 'tx1' with more than 80% transition probability. Thus, 'tx1' becomes a stronger candidate for TC and the heuristic has to ensure that multiple invocations of 'tx1' are close to each other. The heuristic uses the *transaction distance matrix* (Table 4.4) to get the distance in cycles between the invocations of 'tx1'. Finally, the heuristic confirms that in all the cases, inter-transaction distance is fewer than 1,000 cycles (circled cells), thus the transactions are *close*.

Based on the heuristic, the compiler applies TC on 'tx1' of hash table, Vacation, and SSCA2. This implies that repetitive invocations of 'tx1' should be coalesced. Therefore, the compiler unrolls the loop where 'tx1' is located and

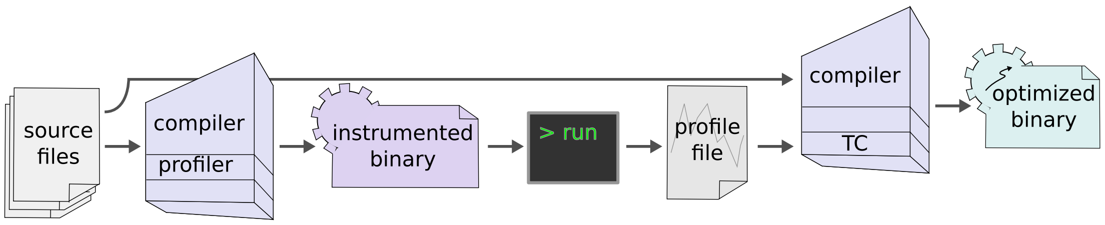


Figure 4.5: The workflow of transaction coalescing. A user compiles source files with the profiler and runs the application to get a profile file. The user then compiles the source files again when TC coalesces transactions according to the profile file.

applies TC on multiple instances of 'tx1'.

4.5.3 Compiler Pass

We implement TC as a profile-guided optimization pass using Berkeley's C Intermediate Language (CIL) tool [Necula *et al.* (2002)]. CIL is a compiler frontend that transforms C language constructs to CIL's abstract syntax tree (AST) and uses GCC [http://gcc.gnu.org] as a backend compiler. We implement the transaction coalescing algorithm in CIL where the algorithm merges transactions identified by the transaction coalescing heuristic.

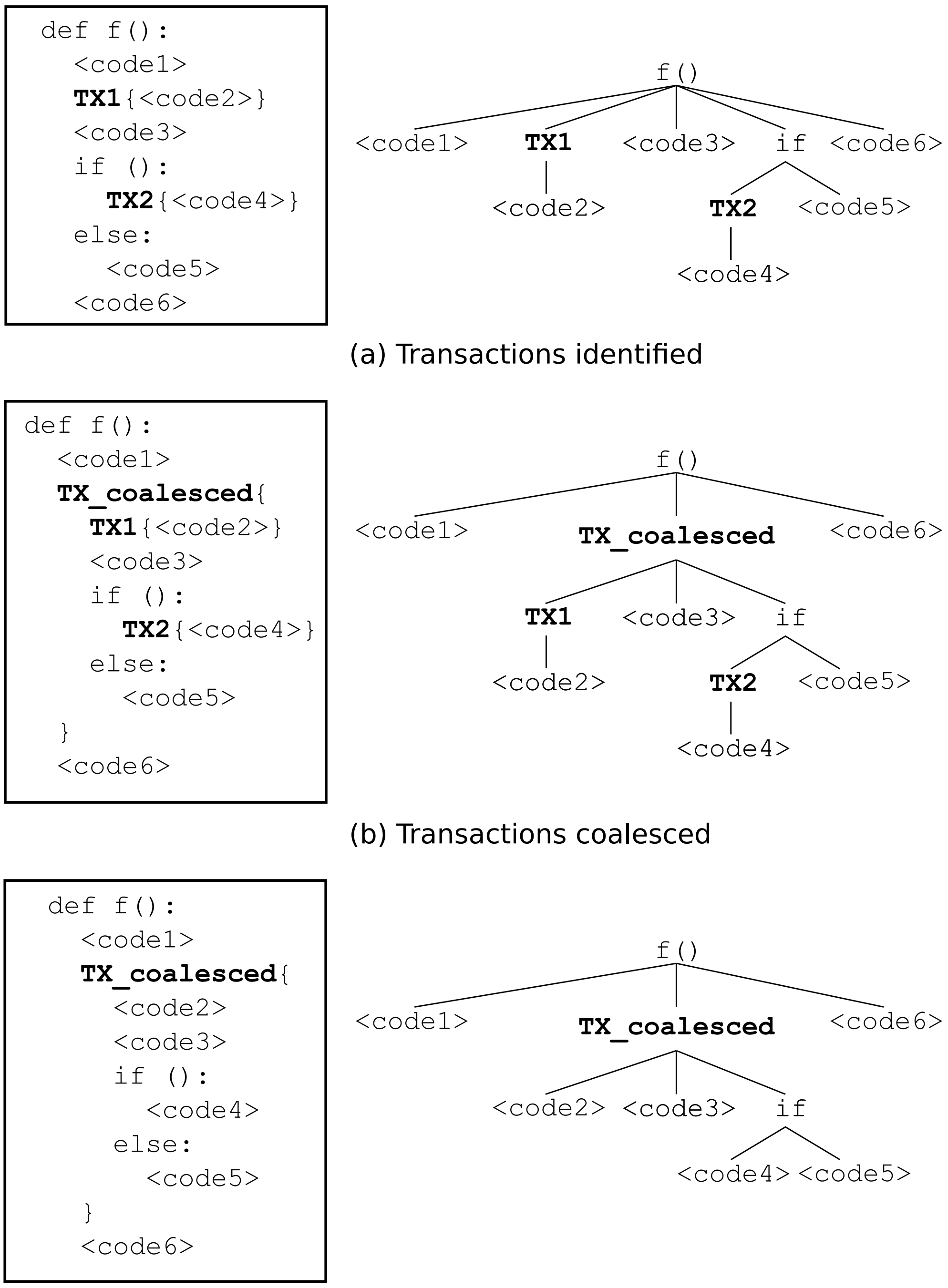
In Figure 4.5 we show the workflow of our implementation. The user compiles an application with profiling enabled and runs the application[1] to generate the profile information. The second time the user compiles the application, the TC compiler applies the transaction coalescing heuristic on the profile information to identify candidate transactions. TC is dependent on loop unrolling and function inlining, which gives the compiler opportunity to coalesce the candidate transactions using the transaction coalescing algorithm. In our implementation, the compiling and running steps are totally automatized.

Transaction coalescing algorithm merges two transactions at the AST level (Figure 4.6) into one large transaction[2].

The algorithm executes in three steps:

[1] Because transaction coalescing is a profile-guided optimization technique, the user has to provide a representative input set for the sample run.

[2] The algorithm can be used recursively to merge more than two transactions.



```
def f():
  <code1>
  TX1{<code2>}
  <code3>
  if ():
    TX2{<code4>}
  else:
    <code5>
  <code6>
```

(a) Transactions identified

```
def f():
  <code1>
  TX_coalesced{
    TX1{<code2>}
    <code3>
    if ():
      TX2{<code4>}
    else:
      <code5>
  }
  <code6>
```

(b) Transactions coalesced

```
def f():
  <code1>
  TX_coalesced{
    <code2>
    <code3>
    if ():
      <code4>
    else:
      <code5>
  }
  <code6>
```

(c) Transactions flattened

Figure 4.6: Transaction coalescing algorithm for merging transactions at the abstract syntax tree (AST) level. The AST on the right is equivalent to the source code on the left.

1. The algorithm identifies transactions (`TX1` and `TX2`) selected for merging (Figure 4.6.a).

2. The algorithm finds the first common parent node (`f()`) of the transactions in the AST, identifies parent's children nodes (`TX1` and `if`) that contain selected transactions, and creates a new transaction node (`TX_coalesced`) including these nodes and all their sibling nodes in between (`TX1`, `<code3>`, and `if`) (Figure 4.6.b).

3. The algorithm removes the original transaction nodes (`TX1` and `TX2`) from the AST (Figure 4.6.c).

4.5.4 Transaction Coalescing - Correctness

The transaction coalescing algorithm preserves atomicity and isolation of transactions. In other words, in the case of race free programs, the coalescing preserves the correctness of a program and it is safe to merge two adjacent transactions[1] or to include non-transactional code in a transaction. When non-transactional code becomes part of a larger transaction, the transaction coalescing algorithm uses the compiler to instrument the accesses to shared data. We would like to emphasise that the compiler will not instrument the accesses to thread-local variables. To illustrate this, we use the following example:

```
__thread int thread_local;
int global;

void f() {
  __transaction_atomic {  /* coalesced tranaction */
    __transaction_atomic{ ++global;} /* tx1 */
    ++thread_local;
    __transaction_atomic{ ++global;} /* tx2 */
  }
}
```

[1] The TC algorithm is analogous to the lock coalescing algorithm [Diniz & Rinard (1997)].

The example shows the function `f()` that has one coalesced transaction. The coalesced transaction includes two smaller transactions (`tx1` and `tx2`) and non-transactional code (`++thread_local`). The following code shows the disassembly of the function `f()`:

```
gcc 4.7 -O2 -fgnu-tm:

f():
    movq %rbx, -16(%rsp)
    movq %rbp, -8(%rsp)
    movl $41, %edi
    subq $24, %rsp
    xorl %eax, %eax
    call _ITM_beginTransaction
    movq %fs:0, %rbx                 /****************************/
    movl $global, %edi               /* 'global' is instrumented */
    call _ITM_RfWU4                  /* with _ITM functions.     */
    movl %eax, %ebp                  /* 'thread_local' is not    */
    addq $thread_local@tpoff, %rbx   /* instrumented and is      */
    movq %rbx, %rdi                  /* accessed directly        */
    call _ITM_RfWU4                  /* with add instruction.    */
    leal 1(%rax), %esi               /****************************/
    movq %rbx, %rdi
    call _ITM_WaWU4
    leal 2(%rbp), %esi
    movl $global, %edi
    call _ITM_WaWU4
    call _ITM_commitTransaction
    movq 8(%rsp), %rbx
    movq 16(%rsp), %rbp
    addq $24, %rsp
    ret
```

The TC algorithm relies on the ability of the compiler to instrument only accesses to shared variables (`global`) but not accesses to thread-local variables (`thread_local`, line 11). If a transaction aborts, the transaction restores variables as they were at the beginning of the transaction, which preserves the semantics of the original program.

GCC provides proper instrumentation of thread-local variables in the following way: Before the start of a transaction the current values of the thread-local variables that are accessed inside of a transaction are pushed on the stack. During the transaction execution, the accesses to thread-local variables are not instrumented. Just in the case of the abort, the values of thread-local variables are restored from the stack.

4.5.5 Executing non-undoable code

It is possible that the transaction coalescing algorithm creates a coalesced transaction that includes non-undoable operations (e.g. `asm`, `volatile`, `printf()`). In that case, the compiler detects the non-undoable code and executes the transaction guaranteeing that it will commit without any abort. A more elaborate explanation follows.

Sometimes, the memory accesses (in the non transactional code) are aliased to MMIO space. The common programming practice is to access MMIO addresses with `ACCESS_ONCE()` macro defined as:

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))}
```

GCC treats all accesses to volatile variables, all system calls, and all non-transactified function calls as non-undoable code. At compile time, GCC has the access to all transactional code in the program. If any code path in the transaction tries to execute non-undoable code, the compiler inserts the code that acquires a shared global lock. The first transaction that acquires the shared global lock validates itself. The validation can succeed or fail:

1. If the validation succeeds, the transaction commits its speculative changes and continues executing while holding the shared global lock. When the transaction ends it releases the shared global lock.

2. If the validation fails, the transaction rollbacks and restarts without releasing the global shared lock. Now, the transaction runs from the beginning to the end while holding the shared global lock. When the transaction ends it releases the shared global lock.

In both cases the execution of non-undoable code is protected with the shared global lock, which ensures correct execution of non-undoable transactions (no transaction can abort the transaction holding the shared global lock). Other transaction that tries to execute non-undoable code will be blocked on the acquisition of the shared global lock.

If the program is not race free, including non-transactional code in a transaction can introduce deviation of the program execution in the presence of rollbacks[1]. The compiler has to ensure that non-transactional code included in the transactions can not be accessed by other threads concurrently. In our implementation, TC is an unsafe optimization and a programmer should enable TC explicitly only if he/she knows that the program is race free.

In Section 5.5 we evaluate the performance improvements of TC on the benchmarks selected for transaction coalescing.

4.6 Evaluation

We perform experiments on a Sun Fire x4140 system equipped with two Six-Core AMD Opteron 2427 (12 cores in total), 32GiB RAM, running Linux 2.6.32-5. We compile the applications with GCC 4.7 with Transactional Memory support and link with TinySTM[Felber *et al.* (2008)] 1.0.3. We implement the TC compiler pass in CIL 1.5 and use GCC as a backend. In this section we explain the implementation of the applications we selected in Section 4.5, i.e. hash table, Vacation, and SSCA2, and we evaluate their performance.

[1]STAMP benchmarks suite is race free because all the shared variables are accessed only inside of transactions and TC can be applied safely.

4.6.1 Benchmarks

Hash table executes repetitively the transactional `lookup()`, `add()`, or `remove()` functions. Each function operates on shared data stored in the hash table. We control the number of updates (`add()` and `remove()`) relative to the total number of operations (`lookup()`, `add()`, and `remove()`) with the *update rate* parameter. We include a function `work()` between the invocations of `remove()` and `add()` to show how non-transactional code instrumentation influences TC. `work()` executes a simple mathematical function ($f(x) = 13 * x \, mod \, y$) *work loop count* (wlc) times.

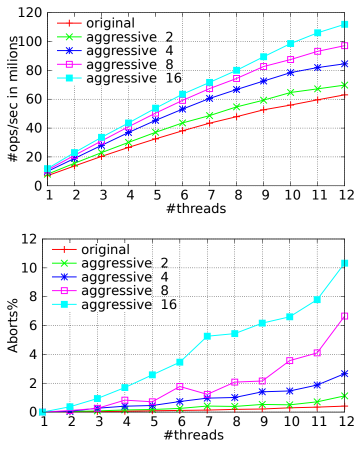
Applying TC on the benchmarks: In Section 4.5.2, the compiler identified 'tx1' of hash table, Vacation, and SSCA2 as candidate transactions for TC. To merge 'tx1' with itself, TC unrolls the loop where 'tx1' is located and merges two (or more) instances of 'tx1' into one transaction[1].

The number of coalesced transactions in the TC compiler pass is controlled by the `aggressive` flag, i.e. when `aggressive = n` the compiler pass merges `n` transactions. If it is not possible to merge `n` transactions, the compiler finds a loop that contains transactions and unrolls it `n` times. After loop unrolling, the compiler merges transactions. If the compiler does not find the loop that contains transactions suitable for merging, the compiler bails out. In our implementation we use `aggressive = 2` as a default parameter but in the benchmarks we vary the `aggressive` factor between 2 and 16. In 'aggressive 2/4/8/16' the loop is unrolled more times to merge 2, 4, 8 and 16 instances of 'tx1' respectively into one transaction.

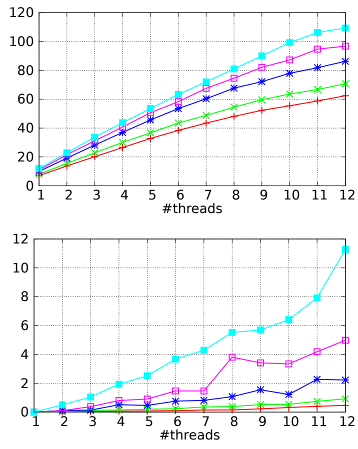
4.6.2 Results

All of the benchmarks (hash table, SSCA2, Vacation) initialize their input set using the `random()` function for which we provide the initial seed value. Hash table uses `random()` to generate the sequence of hash table update/lookup operations (insert 10, delete 3, delete 9, lookup 17, insert 11,...), SSCA2 uses `random()` to generate graph structure, and Vacation uses `random()` to populate

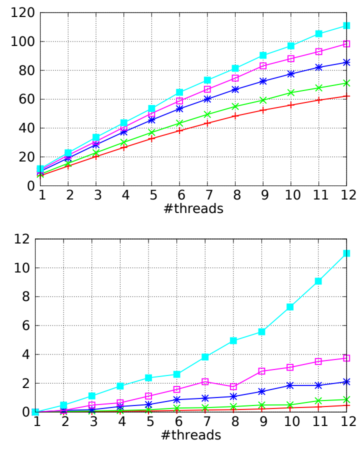
[1]TC would fail in case of Vacation and SSCA2 if the compiler does not unroll loop and does not inline functions with transactions.



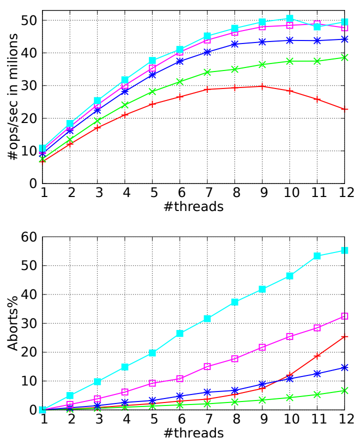
(a) ur = 1%; wlc = 0



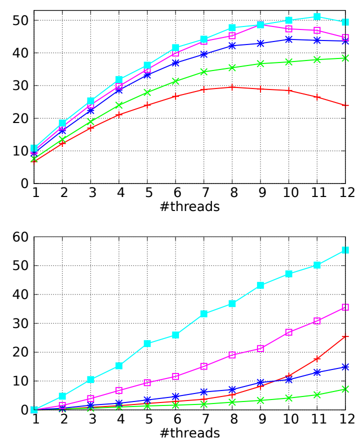
(b) ur = 1%; wlc = 100



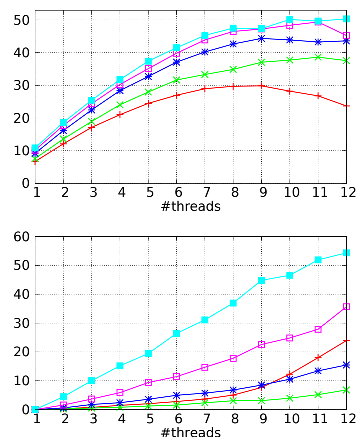
(c) ur = 1%; wlc = 1,000



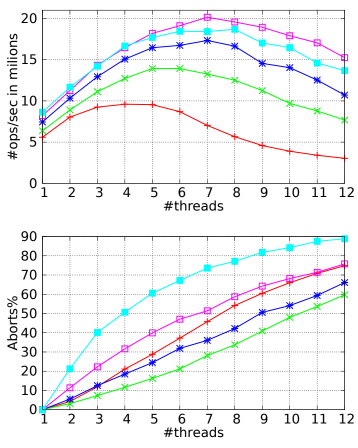
(d) ur = 5%; wlc = 0



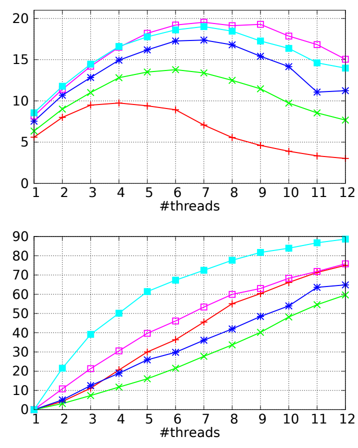
(e) ur = 5%; wlc = 100



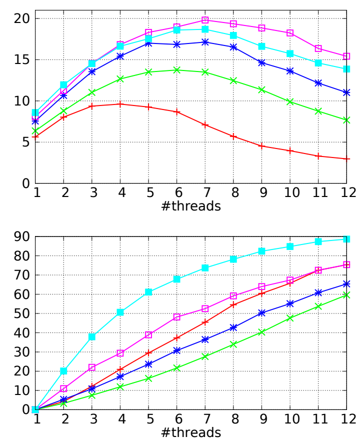
(f) ur = 5%; wlc = 1,000



(g) ur = 20%; wlc = 0



(h) ur = 20%; wlc = 100



(i) ur = 20%; wlc = 1,000

Figure 4.7: Hash table with 1%, 5% and 20% *update rate* (ur), and 0, 100, and 1.000 as *work loop count* (wlc).

the in-memory database and to generate the sequence of accesses to the database (make_car_reservation 3, delete_customer 7, delete_flight_reservation 15,...). For every run of the benchmarks we initialize the seed to a different value. By changing the seed values, the benchmarks generate one input set for profiling and a different input set for measurement phase (which is necessary for obtaining correct results).

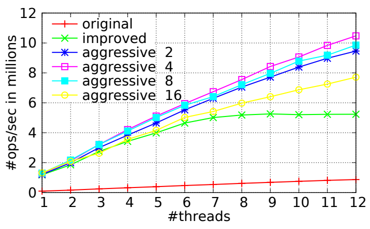
Figure 4.7 shows the performance of the hash table for various update rates (1%, 5%, and 20%), and work loop counts (0, 100, and 1,000). We measure the performance as the number of million operations (`lookup()`, `add()`, or `remove()`) per second. TC improves the performance for all the variations of the input parameters. When transaction coalescing is too aggressive, the performance starts suffering, e.g. 'aggressive 16' performs worse than 'aggressive 8' in Figure 4.7.g. However, 'aggressive 2' (the default aggressive factor) performs from 10.7% (Figure 4.7.a) to 158.5%[1] (Figure 4.7.h) better than 'original', when running with 12 threads. When the update rate is 20% (Figures 4.7.g.h.i) and the aggressive factor is 16, the number of aborts goes up to 90%. For this particular benchmark, the high abort rate has small impact to scalability, but this can not be stated in general[2].

Figure 4.8 shows 'original', 'improved', and 'aggressive' versions of Vacation benchmark. 'improved' is based on 'original' where red-black tree is substituted with hash-table and where random-number generation is moved to the initialization part of the benchmark (explained in Section 4.6.1). These changes improved the performance of 'original' significantly. We apply TC ('aggressive n') on the improved baseline ('improved') to show the benefit of using TC.

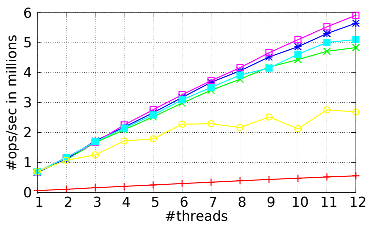
Figure 4.8 shows graphs for different input values of *queries per transaction* (qpt). The size of transactions scale linearly with qpt. When qpt is low (Figure 4.8.a), TC always improves the performance. In Figure 4.8.b we show the performance for the default input value of qpt for the benchmark. 'aggressive 2'

[1]The improvement of 158.5% is mostly due to bad scalability of the hash table with 20% update rate.

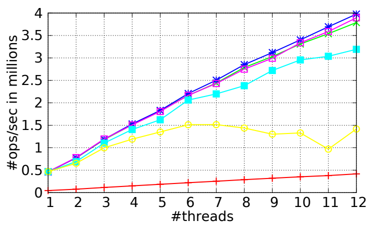
[2] The high abort rate is caused by transactions that repeatedly abort immediately at the beginning of a transaction. Effectively, the aborting transactions spins on the conflicting address.



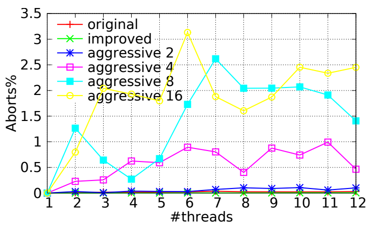
(a) qpt = 1



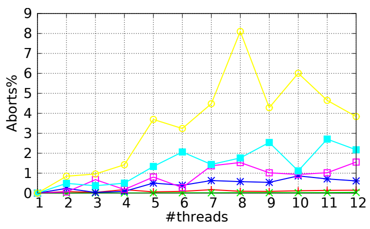
(b) qpt = 2



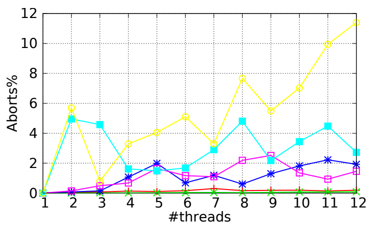
(c) qpt = 3



(d) qpt = 1



(e) qpt = 2



(f) qpt = 3

Figure 4.8: Vacation: Throughput and abort rate for different numbers of *queries per transaction* (qpt).

(the default aggressive factor) performs 19.4% better than ‘improved’, when running with 12 threads. The best case, ‘aggressive 4’, performs 21.2% better than ‘improved’, when running with 12 threads. However, when qpt is 3 ‘aggressive 8-16’ perform worse than ‘improved’ (Figure 4.8.c) due to the high abort rate (Figure 4.8.f). The increased number of aborts negates the benefits of TC.

Figure 4.9 shows ‘original’, ‘improved’, and ‘aggressive’ versions of SSCA2 benchmark. ‘improved’ is based on ‘original’ where we modify the main loop and instead of executing the transactions immediately, we buffer the values of the variables used in the transactions (explained in Section 4.6.1). These changes improved the performance of ‘original’ slightly. We apply TC (‘aggressive n’) on the improved baseline (‘improved’) to show the benefit of using TC. The figure shows the performance of the parallel section of SSCA2 where ‘aggressive 2-16’ have better scalability than ‘improved’. ‘aggressive 2’ (the default aggressive factor) and ‘aggressive 16’ (the best case) perform better than ‘improved’ by 36.4% and 47.4%, respectively, when running with 12 threads.

The default aggressive parameter (‘aggressive 2’) improves the performance of

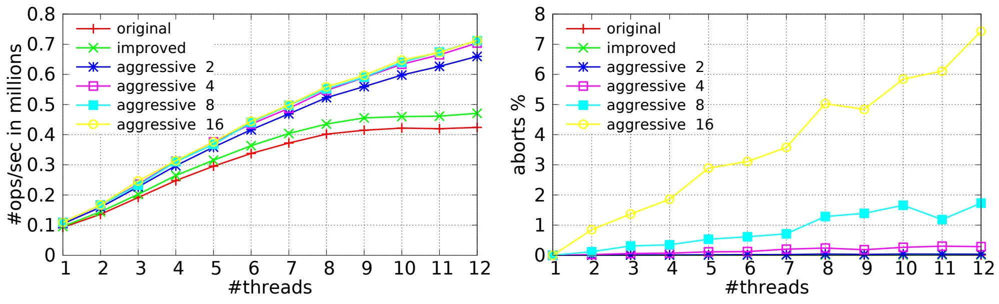


Figure 4.9: SSCA2: Throughput (left) and abort rate (right)

hash table, Vacation, and SSCA2 with negligible increase of the abort rate. The higher values of the aggressive parameter increase the abort rate considerably (Figure 4.7, Figure 4.8, Figure 4.9) and can decrease performance ('aggressive 16' for Vacation, Figure 4.8).

4.7 Summary

In this chapter we showed that transaction coalescing can reduce transactional overheads. We introduced a profiling tool and a transaction coalescing heuristic for collecting and analysing transactional information. We developed a profile-guided compiler pass that identifies and coalesces transactions, where the overheads of transactional start and commit are less than in the original transactions. We evaluated transaction coalescing using the STAMP applications and micro-benchmarks. For the default value of the aggressive factor, transaction coalescing improves the performance by 10.7-158.5% in hash table, by 19.4% in Vacation and by 36.4% in SSCA2, when running with 12 threads. We showed that even better performance improvement can be achieved with larger aggressive factors. The improvement can go up to 21.2% in Vacation (for the aggressive factor equals 4) and up to 47.4% in SSCA2 (for the aggressive factor equals 16), when running with 12 threads.

Chapter 5

Dynamic Transaction Coalescing

The content of this chapter was presented in the paper "Dynamic Transaction Coalescing" at CF'14 (ACM International Conference on Computing Frontiers 2014).

5.1 Introduction to Dynamic Transactional Memory

In Chapter 4, we identified that STMs have high overheads related to starting and committing transactions that may degrade the application performance. To amortize these overheads, we proposed transaction coalescing techniques that coalesce two or more small transactions into one large transaction. However, TC either coalesce transactions statically at compile time, or lack on-line profiling mechanisms that allow coalescing transactions dynamically. Thus, such approaches lead to sub-optimal execution or they may even degrade the performance.

In this chapter, we introduce *Dynamic Transaction Coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput. DTC reduces the overheads of starting and committing a transaction. At compile-time, DTC generates several code paths with a different number of coalesced transactions. At runtime, DTC implements low overhead online profiling and dynamically selects the corresponding code path that maximizes throughput.

Compared to coalescing transactions statically, DTC provides two main improvements. First, DTC implements online profiling which removes the dependency on a pre-compilation profiling step. Second, DTC dynamically selects the best transaction granularity to maximize the transaction throughput taking into consideration the abort rate. We evaluate DTC using common TM benchmarks and micro-benchmarks. Our findings show that: (i) DTC performs like static transaction coalescing in the common case, (ii) DTC does not suffer from performance degradation, and (iii) DTC outperforms static transaction coalescing when an application has phased behavior[1].

5.2 Motivation for Dynamic Transaction Coalescing

Dynamic Transaction Coalescing (DTC) is a compile-time and run-time technique that improves the application performance. At compile-time, DTC generates several code paths with a different number of coalesced transactions. At runtime, DTC implements low overhead online profiling and dynamically selects the corresponding code path that maximizes throughput. In this way, DTC improves the transaction throughput of the loops executing transactions. To evaluate the effectiveness of DTC, we used various benchmarks (CLOMP-TM, SSCA2, Vacation) that are widely used in TM research, and micro-benchmarks (hash-table and red-black tree). Our findings show that DTC improves the performance of SSCA2, Vacation, CLOMP-TM, and hash-table by 44.4%, 45.8%, 66.9%, and 62.9% respectively when running with 12 threads. Also, we show that the DTC's online profiling mechanism has low overhead (11% in the worst case).

The main contributions of this chapter are:

- We show that statically coalescing transactions performs sub-optimally –

[1]We say that an application has phased behaviour when the runtime parameters of the application change during the execution of the application (e.g., number of threads change, transaction throughput change, transitioning from the initialization of the application to the main loop of the application).

even degrading the performance – when the running conditions of the program change (e.g. thread count, abort rate).

- We introduce *Dynamic Transaction Coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput of the loops executing transactions.

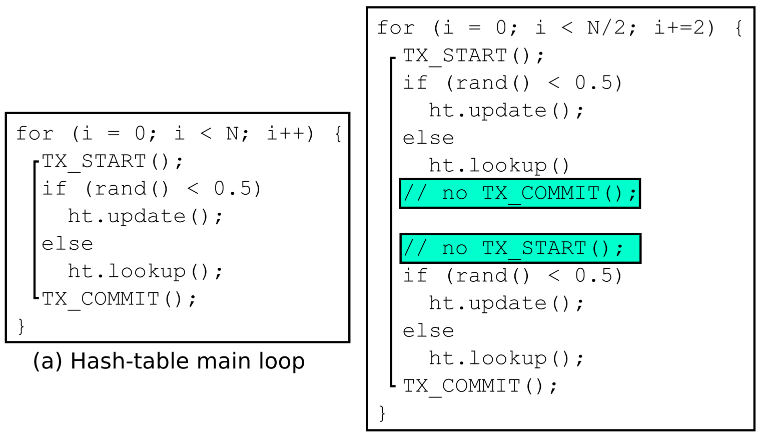
- We show that DTC dynamically selects the best transaction granularity (to maximize transactional throughput) and adapts it accordingly to the program behavior.

The remainder of this chapter is organized as follows: in Section 5.3 we motivate our work; in Section 5.4 we explain the design and implementation of the *dynamic transaction coalescing* technique; in Section 5.5 and Section 5.6 we show experimental results; finally, we conclude in Section 5.7.

5.3 Background

Prior works in TM has shown that starting and committing transactions can incur high overheads [Chung *et al.* (2008); Stipic *et al.* (2014); Wang *et al.* (2012); Yoo *et al.* (2013)]. These overheads may account for even more than 70% of the total execution time in extreme cases [Chung *et al.* (2008)]. In response, researchers proposed mechanisms to coalesce transactions in order to minimize the associated overheads of starting and committing transactions. In this section we demonstrate the short-comings of Stat-TC (from Chapter 4) and we motivate our work on DTC.

Figure 5.1 shows the hash-table benchmark with statically coalesced transactions. The main loop of the benchmark has one transaction that repetitively executes either the `lookup()` or the `update()` function based on a probability. Assuming that the profiling step indicated as optimal a TC factor of 2, Stat-TC unrolls the loop once and removes the unnecessary extra calls to `TX_START()` and `TX_COMMIT()` functions, generating a single coalesced transaction. Hence, the resulting code has lower transactional overhead than the two original transactions.



```
for (i = 0; i < N; i++) {
  TX_START();
  if (rand() < 0.5)
    ht.update();
  else
    ht.lookup();
  TX_COMMIT();
}
```

(a) Hash-table main loop

```
for (i = 0; i < N/2; i+=2) {
  TX_START();
  if (rand() < 0.5)
    ht.update();
  else
    ht.lookup()
  // no TX_COMMIT();

  // no TX_START();
  if (rand() < 0.5)
    ht.update();
  else
    ht.lookup();
  TX_COMMIT();
}
```

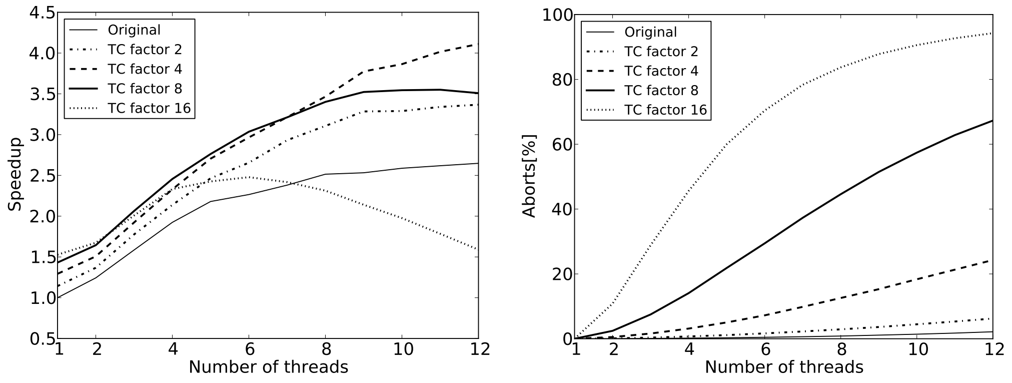
(b) Hash-table with unrolled main loop and coalesced transaction

Figure 5.1: **Hash-table main loop:** (a) For-loop with one transaction in the loop body; (b) For-loop unrolled with the unroll factor 2. The body of the unrolled loop contains the coalesced transaction, which contain the two initial transactions.

Stat-TC generates profiling information based on *single-threaded* execution and uses this information to coalesce transactions at compile time. Such a static approach ignores the behavioral changes of the application as, for example, the number of threads changes. Thus, coalescing transactions statically may result in sub-optimal execution and even performance degradation.

To better understand the limitations of coalescing transactions statically, we use the hash-table benchmark (Figure 5.1.a) executing the `update()` function with a 50% probability. We first run the 'original' benchmark where the main loop executes only one transaction. Then, we instruct the compiler to unroll the loop and to coalesce transactions for various TC factors generating different executables with TC factors equal to 2, 4, 8 and 16.

Figure 5.2a shows the speedup for the hash-table benchmark. We observe that the best TC factor changes with the number of threads. More specifically, when the number of threads is small (1 or 2) the best TC factor is 16, when the number of threads is modest (3-6) the best TC factor is 8, and when the number of threads is large (more than 7) the most suitable TC factor is 4. Moreover, the results show that Stat-TC can increase the performance when the TC factor is



(a) Speedup (b) Abort rate (%)

Figure 5.2: **Hash-table speedup and aborts** with update rate 50%. Hash-table has 1024 entries with 2048 possible key values.

small (2 or 4) but may also degrade performance when the TC factor is large (16) due to high abort rates as we explain next.

Transaction Coalescing increases the length of the transaction, which in turn may increase the abort rate and decrease overall performance. Figure 5.2b shows the abort rate for the hash-table benchmark. For TC factor 2 and 4, the abort rate stays below 25% even when the number of threads increases up to 12. For TC factors 8 and 16 the number of aborts increase rapidly (by more than 50% when the thread number is higher than 8). Based on these results, we conclude that higher TC factors are likely to increase the abort rate. It is also difficult to predict the speedup from the abort rate and vice versa.

Summarizing, this simple example shows that there cannot be a single TC factor performing best for all thread counts. Thus, the approach of Stat-TC to statically generate code with fixed TC factors performs sub-optimally when the number of threads changes. Also, aggressively coalescing transactions based on profiling the single-threaded application may degrade the performance due to an increased abort rate. Since Stat-TC lacks a dynamic feedback mechanism, it cannot adapt to changing abort rates caused by a large number of running threads. In the following section we introduce DTC, a technique that dynamically adjusts the transaction coalesce factor in the presence of changing conditions of the application.

5.4 Dynamic Transaction Coalescing

In this chapter we introduce *Dynamic transaction coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput of the loops executing transactions. At compile-time, DTC unrolls the loops containing the transaction and applies the transaction coalescing algorithm on the transactions contained in the loop body. Instead of generating only one unrolled loop instance, DTC generates several loop instances where each instance is unrolled with a different unroll factor. Consequently, each loop instance contains one coalesced transaction with a different 'TC factor'. At run-time, DTC profiles the application and dynamically selects which loop instance to execute. This way, DTC dynamically selects and executes coalesced transactions with the most appropriate TC factor.

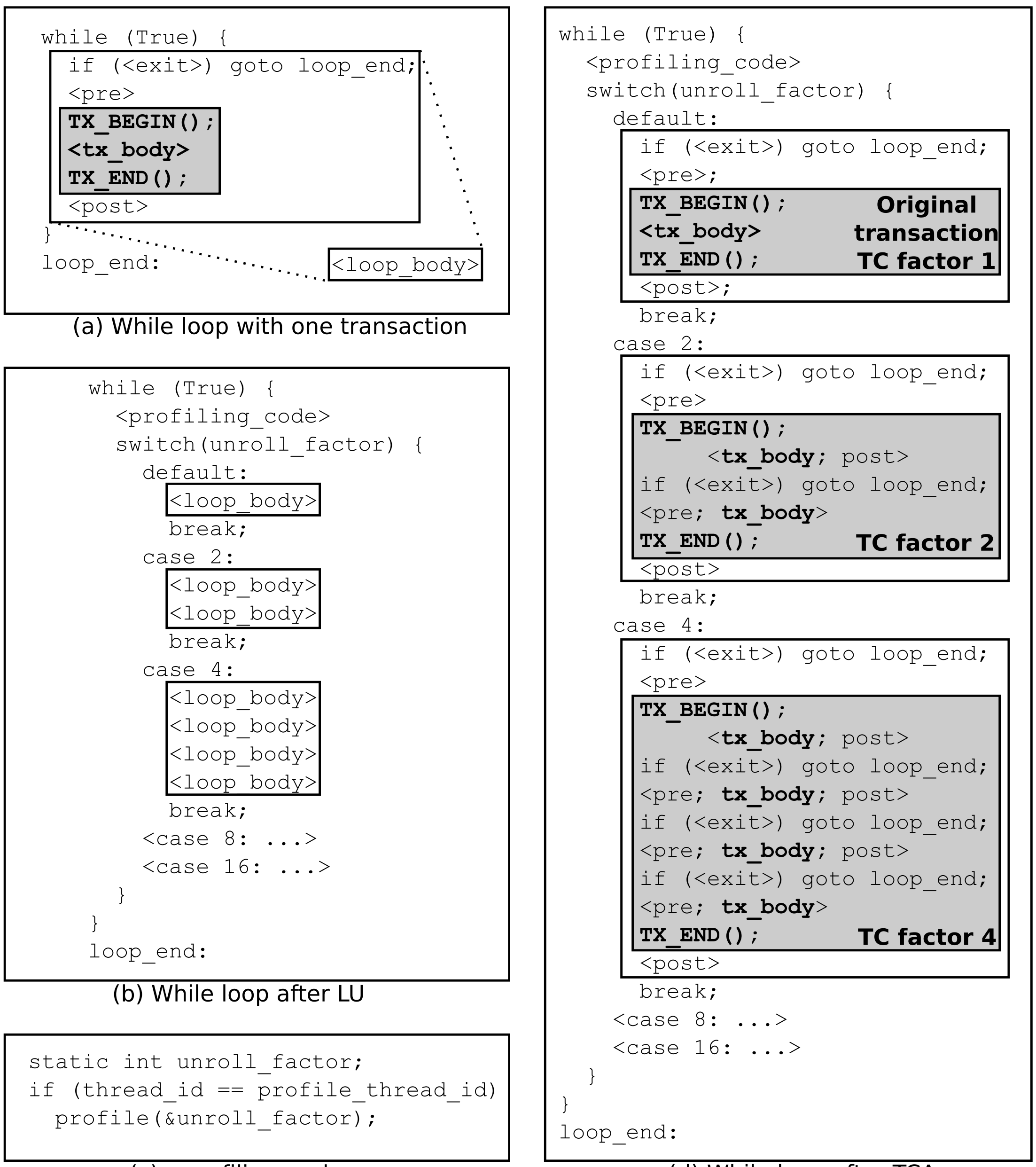
Our implementation of DTC has three main components: (i) loop replacement and unrolling, (ii) transaction coalescing algorithm, and (iii) run-time transaction profiling. These components are related in the following way:

- **Loop replacement and Unrolling (LU):** At compile-time, LU replaces the loops that contain transactions with multiple loop instances where each loop instance is unrolled with a different unroll factor.

- **Transaction Coalescing Algorithm (TCA):** At compile-time, TCA coalesces transactions in each loop instance generated by the LU step. As a result, each loop instance contains one coalesced transaction that has its transaction coalesce factor equal to the loop unroll factor.

- **Run-time Transaction Profiling (RTP):** At compile-time, RTP inserts additional code that profiles transactions. At run-time, the profiling code measures transactional throughput and dynamically selects the TC factor in order to maximize the throughput.

In the remaining of this section, we explain in more detail the LU, TCA, and RTP components of DTC and how they interact with each other.



```
while (True) {
  if (<exit>) goto loop_end;
  <pre>
  TX_BEGIN();
  <tx_body>
  TX_END();
  <post>
}
loop_end:                <loop_body>
```

(a) While loop with one transaction

```
while (True) {
  <profiling_code>
  switch(unroll_factor) {
    default:
      <loop_body>
      break;
    case 2:
      <loop_body>
      <loop_body>
      break;
    case 4:
      <loop_body>
      <loop_body>
      <loop_body>
      <loop_body>
      break;
    <case 8: ...>
    <case 16: ...>
  }
}
loop_end:
```

(b) While loop after LU

```
static int unroll_factor;
if (thread_id == profile_thread_id)
  profile(&unroll_factor);
```

(c) <profiling_code>

```
while (True) {
  <profiling_code>
  switch(unroll_factor) {
    default:
      if (<exit>) goto loop_end;
      <pre>;
      TX_BEGIN();
      <tx_body>
      TX_END();
      <post>;
      break;
    case 2:
      if (<exit>) goto loop_end;
      <pre>
      TX_BEGIN();
           <tx_body; post>
      if (<exit>) goto loop_end;
      <pre; tx_body>
      TX_END();
      <post>
      break;
    case 4:
      if (<exit>) goto loop_end;
      <pre>
      TX_BEGIN();
           <tx_body; post>
      if (<exit>) goto loop_end;
      <pre; tx_body; post>
      if (<exit>) goto loop_end;
      <pre; tx_body; post>
      if (<exit>) goto loop_end;
      <pre; tx_body>
      TX_END();
      <post>
      break;
    <case 8: ...>
    <case 16: ...>
  }
}
loop_end:
```

(d) While loop after TCA

Figure 5.3: **While loop with transaction - Creating coalesced transactions.** Example of a while loop (a) with one transaction in the loop body. The compiler transforms the loop applying *loop replacement and unrolling* giving an expanded loop (b). The expanded loop contains several code paths (default, case 2, 4, 8, 16) containing while loop bodies unrolled with a different loop unroll factor (white rectangles). Next, the compiler transforms the expanded loop, applying the *transaction coalescing algorithm* on the code in each 'case block', giving several coalesced transactions (gray rectangles (d)). In the end, the compiler inserts profiling_code (c) at top of the while loop.

5.4.1 Loop Replacement and Unrolling (LU)

At compile-time, LU replaces the loops containing transactions with multiple loop instances where each loop instance is unrolled with a different unroll factor. To better understand LU's code transformations, we use a simple while loop[1] example (Figure 5.3.a). The loop body contains one transaction surrounded with `<pre>` and `<post>` code that is not a part of the transaction. LU transforms the while loop into a switch statement (Figure 5.3.b) which contains several unrolled instances of the loop. Each `case 2, 4, 8, 16` has the loop body unrolled with unroll factors 2, 4, 8, 16, respectively, while the `default` case statement contains the original (non-unrolled) version of the loop body.

The resulting code has two main characteristics: (i) the case statements contain unrolled loop bodies that are suitable for transaction coalescing and (ii) the code can dynamically select which case statement to execute by changing the value of the `unroll_factor` variable.

5.4.2 Transaction Coalescing Algorithm (TCA)

The compiler takes the transformed while loop from the LU step (Figure 5.3.b) and applies the Transaction Coalescing Algorithm (TCA) [Stipic *et al.* (2014)] on each case block (the code between `case` statement and the following `break` statement). TCA coalesces transactions into one larger transaction in the following way. TCA takes the Abstract Syntax Tree (AST) of each case block, finds the transactions nodes, finds the common parent node of all transactions, encloses the parent node in a transaction, and removes original transaction nodes from the AST. By design, each case statement consists of an unrolled loop body that contains a single coalesced transaction, where each loop unroll factor is equal to the TC factor of the corresponding coalesced transaction. The resulting code (Figure 5.3.d) contains several coalesced transactions where each transaction has a different TC factor. If the loop body contains two or more transactions, TCA coalesces all the transactions by finding the common parent node of all transactions.

[1]Every for-loop, while-loop, and do-while-loop can be transformed into an equivalent while-loop.

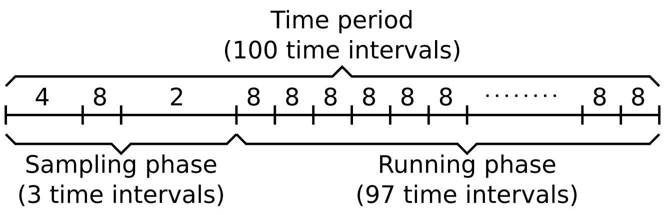


Figure 5.4: **Time period** consists of time intervals where every time interval lasts for 256 transaction commits. The sampling phase has 3 time intervals and the running phase has 97 time intervals. This example shows a sampling phase that uses unroll factors of 4, 8, and 2. In this example, the time interval with unroll factor 8 was measured to be the fastest, therefore, the unroll factor 8 is used for all the time intervals in the running phase.

It is important to stress that TCA correctly coalesces smaller transactions into one larger transaction[1]. Despite the existence of the `goto` statements, the compiler handles the coalesced transaction correctly because it inserts additional code that commits the transaction on every code path that escapes out of the coalesced transaction.

5.4.3 Run-time Transaction Profiling (RTP)

At the beginning of the transformed loop, RTP inserts the `<profiling_code>` (Figure 5.3.c) that calls the `profile()` function. This function measures the transactional throughput of the loop for various TC factors, and selects which coalesced transaction to execute by updating the `unroll_factor` variable. Since all threads read the `unroll_factor`, all of them execute the same code path with the same TC factor. In order to reduce the profiling overhead, we only select one thread to be the profiling thread (`thread_id == profile_thread_id`).

To explain better the implementation of the `profile()` function, we introduce the following terms:

- **Time interval** is the time it takes to commit 256 initial transactions. The `unroll_factor` variable remains constant during the time interval.

[1]For the implementation and the correctness of TCA, please refer to [Stipic *et al.* (2014)].

- **Time period** consists of 100 consecutive time intervals (Figure 5.4).
- **Profiling phase** consists of the first 3 time intervals of a time period.
- **Running phase** consists of the remaining 97 time intervals of a time period. All 97 time intervals in the running phase use the same `unroll_factor` value.

The `profile()` function periodically estimates the transactional throughput by measuring the duration of the time intervals in the sampling phase. For each time interval of the profiling phase, `profile()` sets a different unroll factor (current unroll factor, next larger unroll factor, next smaller unroll factor), and selects the best unroll factor for the next running phase. Since the number of committed transactions is fixed, the throughput is inversely proportional to the interval length. So, the best unroll factor has the shortest time interval.

Our profiling approach introduces two sources of overhead: (i) instrumentation overhead and (ii) phase check overhead. The instrumentation overhead exists because the transformed code is larger than the original code but this overhead is negligible (less than 0.1%). The phase check overhead exists because after every time period (100 time intervals) the `profile()` function changes the unroll factor to check if there is a better one available. So, even if the benchmark executes with the best unroll factor, the profiling executes for two time intervals with "non optimal" unroll factors during the sampling phase. Changing the unroll factor in the sampling phase creates interference in the program execution since it affects the execution of all the threads running the program. To minimize the interference, we empirically selected profile parameters values (256 commits, 100 time intervals, and 3 profiling intervals) in order to provide a good balance between performance and interference.

5.4.4 Discussion

In this chapter we propose DTC as the combination of compile-time and run-time techniques where we generate alternative code paths at compile-time and we select the code paths to execute at run-time. We follow this approach because we target the C/C++ programming environments that do not support run-time

Benchmark	Input arguments
Hash-table	fixed update rates 1%, 20%, 50%
Red-black tree	fixed update rates 1%, 20%, 50%
Hash-table + phases	4 phases with update rates 1%, 20%, 50%, 100%
Red-black tree + phases	4 phases with update rates 1%, 20%, 50%, 100%
Vacation qpt=1,2,3	-n1,2,3 -q90 -u98 -r1048576 -t4194304
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3
CLOMP-TM; No conflicts	-1 1 x1 d6144 128 Stride1 3 1 0 6 1000
CLOMP-TM; Rare conflicts	-1 1 x4 d6144 128 Adjacent 3 1 0 6 1000
CLOMP-TM; High conflicts	-1 1 64 100 128 firstParts 3 1 0 6 1000

Table 5.1: **Benchmark input parameters.**

code generation. Thus we need to generate the different code paths at compile-time. However, implementing DTC for other programming environments that support run-time code generation, e.g. JVM and .Net, may allow the use of just-in-time (JIT) compilation for generating the alternative code paths. This is a potential direction for future work.

5.5 Evaluation Methodology

We perform the experiments on a Sun Fire x4140 system equipped with two Six-Core AMD Opteron 2427 (12 cores in total), with 32GiB RAM, and running Linux 2.6.32-5. We compile the applications with GCC 4.7 which includes Transactional Memory support and link them with TinySTM [Felber *et al.* (2008)] 1.0.3. We manually implement the DTC compiler pass and use GCC as a backend.

5.5.1 Benchmarks

We evaluate the effectiveness of DTC using 2 micro-benchmarks (hash-table and red-black tree) and 3 well-known benchmarks that are used in TM research (Vacation [Minh *et al.* (2008)], SSCA2 [Bader & Madduri (2005)], and CLOMP-TM [Schindewolf *et al.* (2012)]). We select these benchmarks because they cover different application domains: (i) hash-table and red-black tree are used ubiquitously in programs, (ii) Vacation mimics a travel reservation application powered by an in-memory database, (iii) SSCA2 mimics applications operating over large directed, weighted, multi-graphs (e.g. social network graphs and page-rank), and (iv) CLOMP-TM mimics large scale multi-physics applications used in high performance computing. Finally, we use the input parameters specified on each of the benchmark documentation (Table 5.1).

All of the benchmarks have a similar program structure which consist of an initialization section, an execution section, and a finalization section. The initialization sets up the data structures necessary for program execution, the execution section contains the parallel region that runs the transactions in benchmarks' main loop, and finalization section checks the results consistency. All transactions run in the benchmarks' main loop. Next we briefly describe the benchmarks used in this chapter[1].

Hash-table and red-black tree execute repetitively the transactional `lookup()`, `add()`, or `remove()` functions. Each function operates on shared data stored in the hash-table or the red-black tree. We control the number of updates (`add()` and `remove()`) relative to the total number of operations (`lookup()`, `add()`, and `remove()`) with the *update rate* parameter.

The **CLOMP-TM** [Schindewolf *et al.* (2012)] benchmark generates memory accesses that emulate the synchronization characteristics of HPC applications. An unstructured mesh is divided into partitions, where each partition is subdivided into zones. Threads concurrently modify these zones to update the mesh. Specifically, each zone is pre-wired to deposit a value to a set of other zones, called scatter zones, which involves (i) reading the coordinate of a scatter zone, (ii) doing some computation, and (iii) depositing the new value back to the scatter zone. Since threads may be updating the same zone, value deposits need to be synchronized. Conflict probability can be adjusted by controlling how the zones are wired, and by changing the number of scatters per zone; the amount of work done in a critical section can also be adjusted. For our evaluation we use the "Large TM" part of CLOMP-TM , that uses transactions to synchronize critical sections.

[1]For the description of Vacation and SSCA2 please refer to Section 1.2.

5.5.2 Metrics

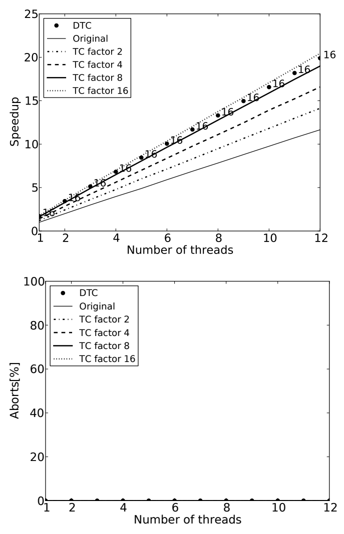
Our results show the performance of applying DTC on the benchmarks' main loop. For all benchmarks we present two plots, one for the speedup and one for the abort rate. We normalize the speedup results to the benchmark running with one thread. In the speedup plots, we also show the dominant TC factor that DTC chooses for that benchmark run (a number next to the small circle in the figures). In the abort rate plots, the abort rate of transactions is plotted in terms of percentage of total executed transactions. Finally, we compare DTC with Stat-TC, and we show the results for different TC factors (2, 4, 8, and 16). We do not plot results for TC factors larger than 16 because they do not provide any performance improvements.

5.6 Results

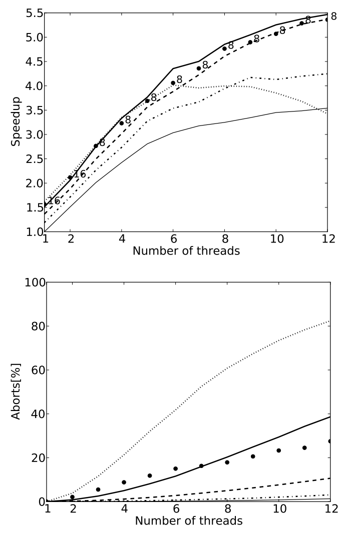
In this section we show how DTC affects the performance of each benchmark, we compare DTC with the performance of Stat-TC, and we also analyze the performance of both mechanisms in the presence of phased execution in the benchmarks.

5.6.1 Hash-table

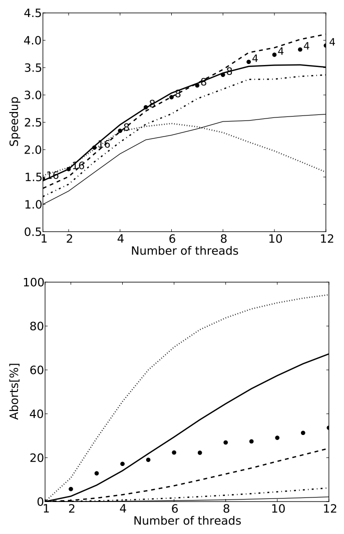
The hash-table benchmark with 1% update rate (Figure 5.5.a) contains short transactions and exhibits a negligible abort rate. Such characteristics make the benchmark well suited for transaction coalescing techniques. The results show that DTC improves the application performance by 62.9% over the original version when running with 12 threads. Compared to Stat-TC, DTC performs slightly worse (7.7%) although DTC chooses correctly the dominant TC factor. This performance difference is due to: (i) the profiling overheads of DTC, and (ii) the suitability of the benchmark to static transaction coalescing since it exhibits well-expected behavior without any phases or abort rate changes. Still, we observe that DTC improves significantly the application performance and performs close to Stat-TC even for the case that dynamically coalescing transactions is not necessary.



(a) Hash-table; ur = 1%



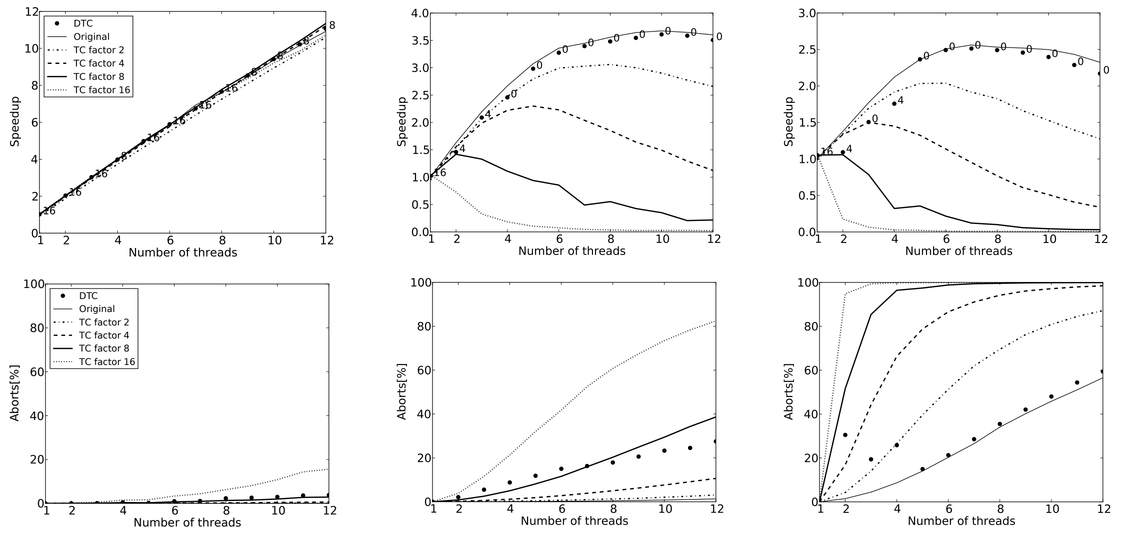
(b) Hash-table; ur = 20%



(c) Hash-table; ur = 50%

Figure 5.5: **Hash-table:** Throughput and abort rate for different numbers of *update rate.*

Figures 5.5.b and 5.5.c show the results for the hash-table with update rates of 20% and 50%. We see that DTC consistently improves the application performance for all thread counts and always follows the best case of Stat-TC. We make the following important observations. First, in Stat-TC there is no single best TC factor that performs best for all thread counts. For example, Figure 5.5.c shows that the best TC factor is 16 for up to 3 threads, 8 for up to 8 threads, and 4 for up to 12 threads. Second, aggressively increasing the TC factor may degrade the application performance significantly. For the case of 50% update rate and with thread count larger than 8, Stat-TC with TC factor 16 performs worse than the original benchmark because of the high abort rate that reaches more than 80%. Unlike Stat-TC, DTC dynamically identifies the best TC factor taking the abort rate into consideration. Due to profiling overheads, DTC does not attain the best performance but is just 5.2% slower than the best TC factor for all thread counts.

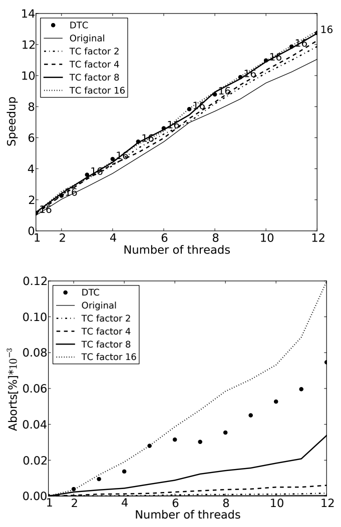


(a) Red-black tree; ur = 1% (b) Red-black tree; ur = 20% (c) Red-black tree; ur = 50%

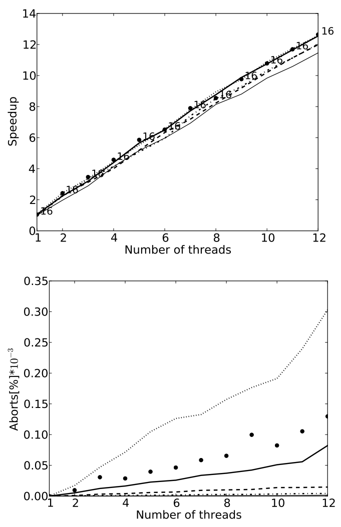
Figure 5.6: **Red-black tree:** Throughput and abort rate for different numbers of *update rate.*

5.6.2 Red-black tree

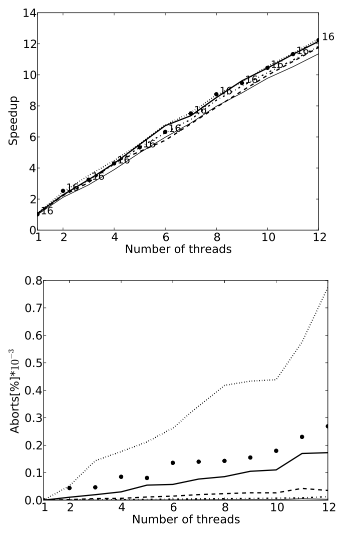
Figures 5.6.a, 5.6.b and 5.6.c show the results for the red-black tree benchmark with update rates 1%, 20% and 50% respectively. For update rates other than 1%, we observe that the benchmark is unsuitable for static transaction coalescing since the performance only degrades. For example, running the benchmark under Stat-TC with more than 4 threads and TC factors of 4 and more degrades performance significantly. This drastic performance degradation is due to the extremely high abort rate that reaches up to 99.9%. On the other hand, DTC does not degrade transaction throughput and follows the performance of the original version, having only 6.6% performance loss due to the profiling overhead (12 threads, update rate 50%). Hence, DTC does not suffer from Stat-TC's performance degradation when coalescing is not useful.



(a) Vacation; qpt = 1



(b) Vacation; qpt = 2



(c) Vacation; qpt = 3

Figure 5.7: **Vacation:** Throughput and abort rate for different numbers of *queries per transaction* (qpt).

5.6.3 Vacation & SSCA2

Vacation (Figures 5.7.a[1], 5.7.b and 5.7.c) and SSCA2 (Figure 5.8) exhibit low abort rate (less than 1%) and demonstrate that coalescing techniques can improve performance. For Vacation with 12 threads and qpt=1, DTC and Stat-TC perform similarly well and improve the performance of the original benchmark by 15.3% and 16.4%. For SSCA2 with 12 threads, DTC and Stat-TC improve the performance of the original benchmark by 44.4% and 45.8%. Again, we conclude that DTC significantly improves performance of the applications, and closely follows the performance of Stat-TC in case there is no need for dynamically changing the TC factor, and meanwhile the profiling overhead remains low.

[1] Figure 5.7.a shows maximum abort rate of 0.12% which is higher than the abort rate of 3% in Figure 4.8. This difference exist because in the Chapter 4. we use automated CIL tool to transforms the code containing transactions and resulting transactions become overinstrumented. In the Chapter 5. we manually transform the code containing transctions and we are able to create new transactions with smaller overhead compared to the CIL tool. The overinstumented transaction case higher abort rate in the case of Vacation.

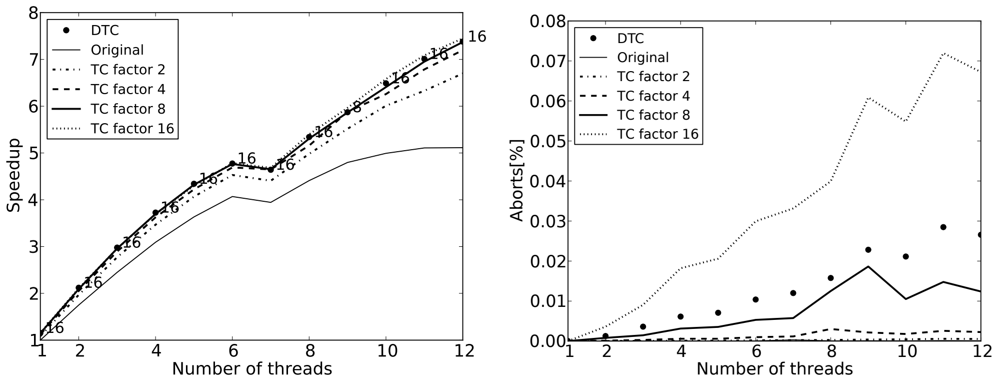


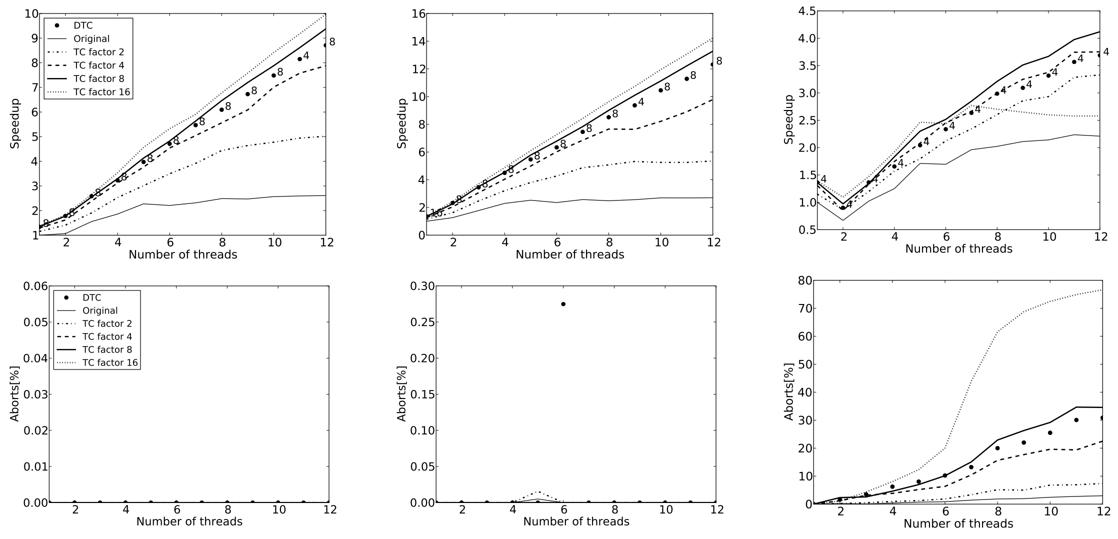
Figure 5.8: **SSCA2.**

5.6.4 CLOMP-TM

Figure 5.9 shows the performance of the CLOMP-TM benchmark. TC factor 8 and 16 perform the best when the conflicts are rare (Figure 5.9.a and Figure5.9). DTC selects TC factors 4 and 8 for the execution but the best performing TC factors are 8 and 16. DTC is not able to 'find' the best TC factor due to profiling interference. Still, DTC does not suffer from TC's performance degradation (Figure 5.9.c with TC factor 16) and DTC performs just 11% worse than the best TC factor, while still being substantially better than the original version.

5.6.5 Phased execution

Up to this point, we analyzed the performance of DTC and we compared it to that of Stat-TC. However, we performed our evaluation on various benchmarks that do not have phased execution. In other words, the running conditions (thread count) or the program behavior (e.g. abort rate) did not change dynamically during the execution. In order to compare the performance of DTC and Stat-TC in changing conditions, we modify the hash-table and red-black tree benchmarks and introduce phased execution. We introduced four phases in the benchmark execution where each phase has the same duration (in seconds) and executes with different update rate parameters (ur = 1%, 20%, 50%, 100%).



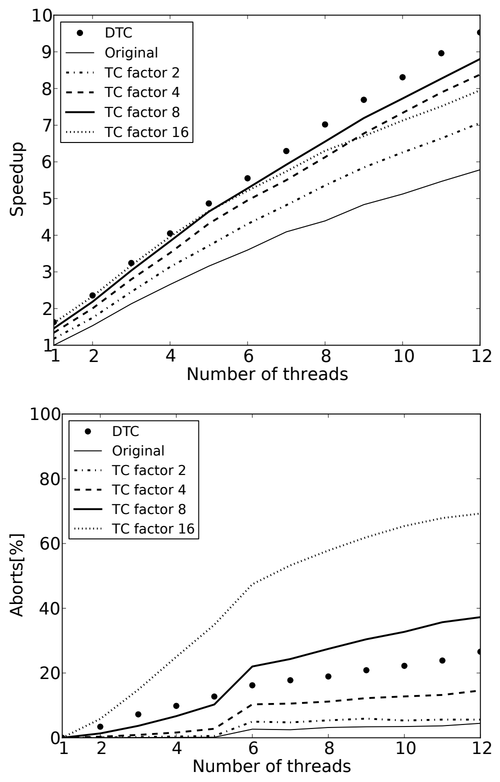
(a) CLOMP-TM; No conflicts (b) CLOMP-TM; Rare conflicts (c) CLOMP-TM; High conflicts

Figure 5.9: **CLOMP-TM**

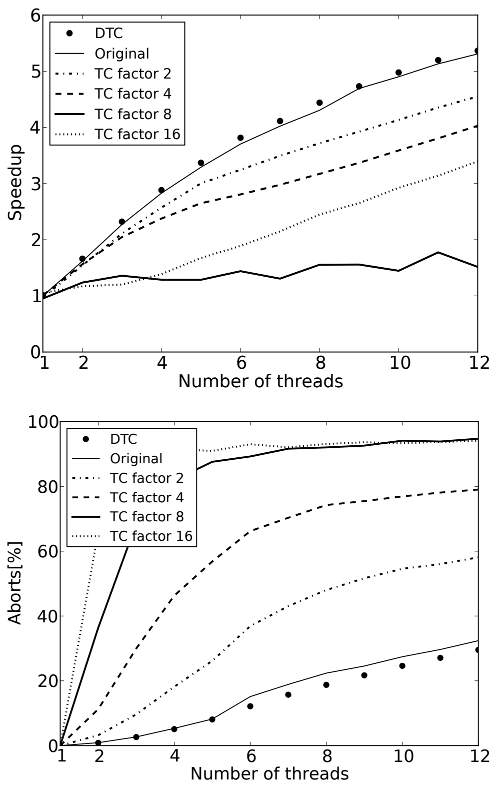
Figure 5.10.a shows the speedup[1] and the abort rate for the hash-table benchmark with phases. We observe that DTC improves the performance by 64.7% for the hash-table benchmark, outperforming Stat-TC by 8.3%. The reason is that there is no single TC factor that performs best for all execution phases. Thus, Stat-TC executes various phases with sub-optimal TC factor. In contrast, DTC dynamically adapts to the program execution and selects the best TC factor improving application performance.

Figure 5.10.b shows the results for the red-black tree benchmark with phased execution. We observe that DTC performs equally or slightly better than the original version. DTC's profiling mechanism does not identify any potential benefits through coalescing, and executes most of the time without any coalesced transactions (TC factor 0).

[1]We do not plot the dominant TC factor for DTC because it changes during the program execution.



(a) Hash-table with phased execution



(b) Red-black tree with phased execution

Figure 5.10: **Hash-table and red-black tree** with phased execution. Each phase last for a quarter of the total execution time and each phase has update rate (ur) of 1%, 20%, 50%, and 100%.

5.6.6 Overview

We make the following observations to summarize this section:

- DTC is able to considerably increase the application performance over the original version (e.g. hash-table, Vacation, and SSCA2).
- DTC follows closely the performance of Stat-TC with the best TC factor for those applications that do not exhibit high abort rates. The performance gap is due to the profiling mechanism, but still remains lower than 11% in the worst case.

- DTC outperforms both the original version and Stat-TC with the best TC factor when programs exhibit phased execution (e.g., hash-table and red-black tree with phases).

- DTC does not introduce performance degradation (except for the profiling overhead) when transactional coalescing is not beneficial, while Stat-TC may harm the performance (e.g., red-black tree).

5.7 Summary

In this chapter we introduced *dynamic transaction coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput of the loops executing transactions. We explained how DTC transforms the loops and how DTC generates coalesced transactions of different sizes. Also, we explained the implementation of DTC's online profiling and showed how profiling helps to find the best transaction granularity that increases the throughput. We evaluated DTC using 3 benchmarks(SSCA2, Vacation, and CLOMP-TM) and 2 micro-benchmarks (hash-table and red-black tree).

We show that DTC improves the performance of SSCA2, Vacation, CLOMP-TM, and hash-table by 44.4%, 45.8%, 66.9%, and 62.9% respectively (running with 12 threads and having a high conflict rate). We also show that DTC performs close to the statically selected best transaction coalesce factor, and that DTC's online profiling has small performance overhead. The overhead is 6.6% in hash-table and red-black tree; 11% in CLOMP-TM; and less than 1% in SSCA2 and Vacation. Finally, we show that DTC performs better than Stat-TC when phases are present improving the performance of hash-table and red-black by 8.2% and 1.1% with respect to Stat-TC with the best TC factor.

Chapter 6

Conclusions

6.1 Thesis Contributions

In this thesis we presented four new techniques for improving the performance of software transactional memory systems: (i) Abstract Nested Transactions (ANT), (ii) TagTM, (iii) profile-guided transaction coalescing, and (iv) dynamic transaction coalescing.

In Chapter 2, we introduced the idea of *abstract nested transactions* (ANTs) for identifying sections of an `atomic` block that are likely to be the victims of benign conflicts. By re-executing ANTs we can avoid re-executing the whole atomic block that contains them. Unlike other techniques for improving the scalability of `atomic` blocks ANTs are semantically transparent and can be used as a performance tuning technique without risk of changing the semantics or serializability of the code in which they are used.

In Chapter 3, we introduced TagTM, a software TM system augmented with new hardware mechanism that we call GTags. GTags are new hardware cache coherent tags used for fast meta-data access. TagTM use GTags to reduce the cost associated with accesses to the transactional data and corresponding metadata. For the evaluation of TagTM, we used STAMP benchmark suite. In the average case TagTM provide the speedup of 7-15% (across all STAMP applications), and in the best case shows up to 52% speedup of committed transaction execution time (for SSCA2).

In Chapter 4, we showed that transaction coalescing can reduce transactional overheads. We introduced a profiling tool and a transaction coalescing heuristic for collecting and analysing transactional information. We developed a profile-guided compiler pass that identifies and coalesces transactions, where the overheads of transactional start and commit are less than in the original transactions. We evaluated transaction coalescing using the STAMP applications and micro-benchmarks. For the default value of the aggressive factor, transaction coalescing improves the performance by 10.7-158.5% in hash table, by 19.4% in Vacation and by 36.4% in SSCA2, when running with 12 threads. We showed that even better performance improvement can be achieved with larger aggressive factors. The improvement can go up to 21.2% in Vacation (for the aggressive factor equals 4) and up to 47.4% in SSCA2 (for the aggressive factor equals 16), when running with 12 threads.

In Chapter 5, we introduced *dynamic transaction coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput of the loops executing transactions. We explained how DTC transforms the loops and how DTC generates coalesced transactions of different sizes. Also, we explained the implementation of DTC's online profiling and showed how profiling helps to find the best transaction granularity that increases the throughput. We evaluated DTC using 3 benchmarks(SSCA2, Vacation, and CLOMP-TM) and 2 micro-benchmarks (hash-table and red-black tree). We show that DTC improves the performance of SSCA2, Vacation, CLOMP-TM, and hash-table by 44.4%, 45.8%, 66.9%, and 62.9% respectively (running with 12 threads and having a high conflict rate). We also show that DTC performs close to the statically selected best transaction coalesce factor, and that DTC's online profiling has small performance overhead. The overhead is 6.6% in hash-table and red-black tree; 11% in CLOMP-TM; and less than 1% in SSCA2 and Vacation. Finally, we show that DTC performs better than static TC when phases are present improving the performance of hash-table and red-black by 8.2% and 1.1% with respect to static TC with the best TC factor.

6.2 Future Work

We showed that TagTM and transaction coalescing improve the performance of TM systems. TagTM showed that by putting the metadata in the cache line can speedup the performance of STM systems. We believe that it would be possible to extend the hardware prefetching to prefetch the metadata during transaction execution. We believe that TC and DTC could be used even in the hardware TM systems to lower the overheads of hardware snapshotting. Unfortunately, Intel did not release the implementation details of of their HTM implementation, so we are not able to predict the potential impact of transaction coalescing techniques in HTM systems.

We believe that TM will find its uses in system libraries and in the OS implementations. So it would be worth investigating real life uses of hardware TM implementations in Linux kernel to speedup the synchronisation of shared data-structures used in the OS. And finally, We believe that TM is here to stay.

Chapter 7

Publications on the topic

7.1 Publications from the thesis:

- Tim Harris and Srđan Stipić. “Abstract nested transactions”. In The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2007), Portland, OR (United States), Aug 2007.

- Srđan Stipić, Saša Tomić, Ferad Zyulkyarov, Adrian Cristal, Osman Unsal and Mateo Valero. “TagTM - Accelerating STMs with hardware tags for fast meta-data access”. In Design, Automation and Test in Europe, Dresden (Germany), Mar 2012.

- Srđan Stipić, Vesna Smiljković, Osman Unsal, Adrian Cristal and Mateo Valero. “Profile-Guided Transaction Coalescing—Lowering Transactional Overheads by Merging Transactions”. In ACM Transactions on Architecture and Code Optimization 2014, Vienna (Austria), Jan 2014.

- Srđan Stipić, Vasileios Karakostas, Vesna Smiljković, Vladimir Gajinov, Osman Unsal, Adrian Cristal and Mateo Valero. “Dynamic Transaction Coalescing”. ACM International Conference on Computing Frontiers, Cagliari (Italy), May 2014.

7.2 Related publication not included in the thesis:

- Milos Milovanovic, Osman Unsal, Adrian Cristal, Srđan Stipić, Ferad Zyulkyarov and Mateo Valero. “Compile time support for using transactional memory in C/C++ applications”. In The 11th Annual Workshop on the Interaction between Compilers and Computer Architecture (INTERACT-11), pp. 16-23, Phoenix, AR (United States), Feb 2007.

- Nehir Sonmez, Cristian Perfumo, Srđan Stipić, Osman Unsal, Adrian Cristal and Mateo Valero. “UnreadTVar: Extending Haskell Software Transactional Memory for Performance”. In The 8th Symposium on Trends in Functional Programming (TFP 2007), pp. 1-11, New York (United States), Apr 2007.

- Nehir Sonmez, Cristian Perfumo, Srđan Stipić, Adrian Cristal, Osman Unsal and Mateo Valero. “Increasing the Performance of Haskell Software Transactional Memory”. In II Congreso Español de Informática (CEDI 2007), Zaragoza (Spain), Sep 2007.

- Milos Milovanovic, Osman Unsal, Adrian Cristal, Srđan Stipić, Ferad Zyulkyarov and Mateo Valero. “Extending C/C++ Language with Atomic Constructs”. In II Congreso Español de Informática (CEDI 2007), Zaragoza (Spain), Sep 2007.

- Cristian Perfumo, Nehir Sonmez, Srđan Stipić, Osman Unsal, Adrian Cristal, Tim Harris and Mateo Valero. “The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment”. In 5th Conference on Computing Frontiers, pp. 67-78, Ischia (Italy), May 2008. [PDF]

- Nehir Sonmez, Cristian Perfumo, Srđan Stipić, Osman Unsal and Mateo Valero. “Profiling Transactional Memory Applications on an Atomic Block Basis”. In Advanced Computer Architecture and Compilation for Embedded Systems. ACACES 2008, pp. 75-79, L’Aquila (Italy), Jul 2008.

- Gokcen Kestor, Srđan Stipić, Osman Unsal, Adrian Cristal and Mateo Valero. "RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications". In 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2009), Raleigh, NC (United States), Feb 2009. [PDF]

- Nehir Sonmez, Cristian Perfumo, Srđan Stipić, Tim Harris, Osman Unsal, Adrian Cristal and Mateo Valero. "Software Transactional Memory Implementation". In Advanced Computer Architecture and Computation for Embedded Systems (ACACES 2009), Poster Session, pp. 101-103, Terrassa (Spain), Jul 2009.

- Ferad Zyulkyarov, Srđan Stipić, Tim Harris, Osman Unsal, Adrian Cristal, Ibrahim Hur, Mateo Valero. "Discovering and Understanding Performance Bottlenecks in Transactional Applications". PACT'10: Proc. 19th International Conference on Parallel Architectures and Compilation Techniques, September 2010. (Best Paper)

- Vladimir Gajinov, Srđan Stipić, Osman Unsal, Tim Harris, Eduard Ayguade and Adrian Cristal. "Supporting Stateful Tasks in a Dataflow Graph. In 21st International Conference on Parallel Architectures and Compilation Techniques (PACT-2012)". Poster Session, pp. 435-436, Minneapolis (United States), Sep 2012.

- Vladimir Gajinov, Srđan Stipić, Osman Unsal, Tim Harris, Eduard Ayguade and Adrian Cristal. "Integrating Dataflow Abstractions into the Shared Memory Model". In 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2012), pp. 243-251, New York (United States), Oct 2012.

- Vesna Smiljković, Srđan Stipić, Osman Unsal, Adrian Cristal and Mateo Valero. "Transaction Coalescing - Lowering Transactional Overheads by Merging Transactions". In Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2013), Berlin (Germany), Jan 2013.

7.2 Related publication not included in the thesis:

- Srđan Stipić, Vasileios Karakostas, Vesna Smiljkovic, Vladimir Gajinov, Osman Unsal, Adrian Cristal, and Mateo Valero. "Dynamic Transaction Coalescing". ACM International Conference on Computing Frontiers, Cagliari (Italy), May 2014.
- Vladimir Gajinov, Igor Erić, Srđan Stipić, Osman Unsal, Eduard Ayguade, and Adrian Cristal. "DaSH: A Benchmark for Hybrid Dataflow and Shared Memory Programming Models. With Comparative Evaluation of Three Hybrid Dataflow Models". ACM International Conference on Computing Frontiers, Cagliari (Italy), May 2014.

Bibliography

Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B. & Shpeisman, T. (2006a). Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, 26–37. 15

Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B. & Shpeisman, T. (2006b). Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, **41**, 26–37. 59

Afek, Y., Korland, G. & Zilberstein, A. (2011). Lowering STM overhead with static analysis. *Languages and Compilers for Parallel Computing*, 31–45. 14, 15, 62

Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E. & Lie, S. (2005). Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, 316–327. 8, 9

Bader, D. & Madduri, K. (2005). Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. *High Performance Computing–HiPC 2005*, 465–476. 12, 95

Baugh, L., Neelakantam, N. & Zilles, C. (2008). Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, 115–126, IEEE Computer Society, Washington, DC, USA. 59

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. & O'Neil, P. (1995). A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, **24**, 1–10. 22

Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G. & Reinhardt, S.K. (2006). The m5 simulator: Modeling networked systems. *IEEE Micro*, **26**, 52–60. 53

Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M. & Wood, D.A. (2007). Performance pathologies in hardware transactional memory. In *ISCA*, 81–91. 9

Cachopo, J. & Rito-Silva, A. (2006). Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, **63**, 172–185. 25

Cao Minh, C., Chung, J., Kozyrakis, C. & Olukotun, K. (2008). STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. 11th IEEE International Symposium on Workload Characterization*, 35–46. 53

Chen, S., Kozuch, M., Strigkos, T., Falsafi, B., Gibbons, P.B., Mowry, T.C., Ramachandran, V., Ruwase, O., Ryan, M. & Vlachos, E. (2008). Flexible hardware acceleration for instruction-grain program monitoring. *SIGARCH Comput. Archit. News*, **36**, 377–388. 59

Chung, J., Dalton, M., Kannan, H. & Kozyrakis, C. (2008). Thread-safe dynamic binary translation using transactional memory. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 279–289, IEEE. 14, 62, 87

Cormen, T. (2001). *Introduction to Algorithms*. MIT Press. 23

Diniz, P. & Rinard, M. (1997). Synchronization transformations for parallel computing. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 187–200, ACM. 75

Dragojevic, A., Ni, Y. & Adl-Tabatabai, A. (2009). Optimizing transactions for captured memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 214–222, ACM. 14, 15, 62

Felber, P., Fetzer, C., Müller, U., Riegel, T., Süsskraut, M. & Sturzrehm, H. (2007). Transactifying applications using an open compiler framework. In *TRANSACT*. 47, 64

Felber, P., Fetzer, C. & Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 237–246, ACM. 78, 95

Fraser, K. & Harris, T. (2004). Practical Lock-Freedom. *University of Cambridge Computer Laboratory, Technical Report number*, **579**. 23

Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C. & Olukotun, K. (2004a). Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 102, IEEE Computer Society. 8

Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C. & Olukotun, K. (2004b). Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, 102–, IEEE Computer Society, Washington, DC, USA. 58

Harris, T. (2001). A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, vol. 2180, 300–314, Springer. 27

Harris, T., Marlow, S. & Jones, S. (2005a). Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, 49–61, ACM New York, NY, USA. 39

Harris, T., Marlow, S., Peyton-Jones, S. & Herlihy, M. (2005b). Composable memory transactions. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 48–60. 21, 32

Harris, T., Plesko, M., Shinnar, A. & Tarditi, D. (2006). Optimizing memory transactions. In *Proceedings of the 2006 PLDI Conference*, vol. 41, 14–25, ACM New York, NY, USA. 23

Harris, T., Tomic, S., Cristal, A. & Unsal, O. (2010). Dynamic filtering: multi-purpose architecture support for language runtime systems. *SIGARCH Comput. Archit. News*, **38**, 39–52. 59

Herlihy, M. & Moss, J.E.B. (1993). Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 289–300. 3

Herlihy, M., Luchangco, V., Moir, M. & Scherer III, W. (2003). Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 92–101, ACM Press New York, NY, USA. 25

http://gcc.gnu.org (????). GCC home page. http://gcc.gnu.org/. 73

Knight, T. (1986). An architecture for mostly functional languages. *Proceedings of the 1986 ACM conference on LISP and functional programming*, 105–112. 3

Larus, J. & Rajwar, R. (2007). Transactional Memory. *Synthesis Lectures On Computer Architecture*, **1**, 1–226. 18

Michael, M. (2002). High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 73–82, ACM New York, NY, USA. 27

Minh, C., Chung, J., Kozyrakis, C. & Olukotun, K. (2008). STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 35–46, IEEE. 9, 63, 95

Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C. & Olukotun, K. (2007a). An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, 69–80. 14, 62

Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C. & Olukotun, K. (2007b). An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proc. 34th International Symposium on Computer architecture*, 69–80. 58

Moir, M., Moore, K. & Nussbaum, D. (2008). The Adaptive Transactional Memory Test Platform: A tool for experimenting with transactional code for Rock. *The third annual ACM SIGPLAN Workshop on Transactional Computing, February*. 9

Moore, K., Bobba, J., Moravan, M., Hill, M. & Wood, D. (2006a). LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*. 21

Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D. & Wood, D.A. (2006b). Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 254–265. 8, 9

Necula, G., McPeak, S., Rahul, S. & Weimer, W. (2002). CIL: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*, 209–265, Springer. 73

Ni, Y., Menon, V., Adl-Tabatabai, A., Hosking, A., Hudson, R., Moss, J., Saha, B. & Shpeisman, T. (2007). Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 68–78, ACM Press New York, NY, USA. 20, 25

Ruppert, J. (1995). A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of algorithms*, **18**, 548–585. 12

Saha, B., Adl-Tabatabai, A. & Jacobson, Q. (2006a). Architectural Support for Software Transactional Memory. *International Symposium on Microarchitecture: Proceedings of the 39 th Annual IEEE/ACM International Symposium on Microarchitecture*, **9**, 185–196. 9

Saha, B., Adl-Tabatabai, A. & Jacobson, Q. (2006b). Architectural support for software transactional memory. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 185–196, IEEE. 14, 62

Saha, B., Adl-Tabatabai, A.R. & Jacobson, Q. (2006c). Architectural support for software transactional memory. In *MICRO '06: Proc. 39th IEEE/ACM International Symposium on Microarchitecture*, 185–196. 58

Scherer III, W.N. (2006). *Synchronization and concurrency in user-level software systems*. Ph.D. thesis, Citeseer. 29

Schindewolf, M., Biliari, B., Gyllenhaal, J., Schulz, M., Wang, A. & Karl, W. (2012). What scientific applications can benefit from hardware transactional memory? In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 1–11, IEEE. 95, 96

Shavit, N. & Touitou, D. (1995). Software transactional memory. *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, 204–213. 3

Shivakumar, P. & Jouppi, N. (2001). Cacti 3.0: An integrated cache timing, power, and area model. Tech. rep., Technical Report 2001/2, Compaq Computer Corporation. 57

Shriraman, A., Marathe, V., Dwarkadas, S., Scott, M., Eisenstat, D., Heriot, C., Scherer III, W. & Spear, M. (2006). Hardware Acceleration of Software Transactional Memory. *ACM SIGPLAN Workshop on Transactional Computing, Ottawa, ON, Canada, June*. 9

Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarkadas, S. & Scott, M.L. (2007). An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, 104–115. 14, 62

Stipic, S., Tomic, S., Zyulkyarov, F., Cristal, A., Unsal, O. & Valero, M. (2012). TagTM - accelerating STMs with hardware tags for fast meta-data access. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 39 –44. 62

Stipic, S., Smiljkovic, V., Cristal, A., Unsal, O. & Valero, M. (2014). Profile-guided transaction coalescing - lowering transactional overheads by merging transactions. *The ACM Transactions on Architecture and Code Optimization, TACO 2014*. 87, 92, 93

Venkataramani, G., Roemer, B., Solihin, Y. & Prvulovic, M. (2007). Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 273–284, IEEE Computer Society, Washington, DC, USA. 59

Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R. & Michael, M. (2012). Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 127–136, ACM. 14, 87

Wang, C., Chen, W., Wu, Y., Saha, B. & Adl-Tabatabai, A. (2007). Code generation and optimization for transactional memory constructs in an unmanaged language. In *Code Generation and Optimization, 2007. CGO'07.*, 34–48, IEEE. 14, 62

Wu, P., Michael, M., von Praun, C., Nakaike, T., Bordawekar, R., Cain, H., Cascaval, C., Chatterjee, S., Chiras, S., Hou, R. *et al.* (2009). Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience*, **21**, 7–23. 14, 15, 62

Yoo, R.M., Hughes, C.J., Lai, K. & Rajwar, R. (2013). Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Supercomputing 2013.*. 9, 15, 87

Zeldovich, N., Kannan, H., Dalton, M. & Kozyrakis, C. (2008). Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, 225–240, USENIX Association, Berkeley, CA, USA. 59

Zilles, C. & Rajwar, R. (2007). Transactional memory and the birthday paradox. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, 303–304, ACM Press New York, NY, USA. 23

Zyulkyarov, F., Stipic, S., Harris, T., Unsal, O., Cristal, A., Hur, I. & Valero, M. (2010a). Discovering and understanding performance bottlenecks in transactional applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 285–294, ACM. 12

Zyulkyarov, F., Stipic, S., Harris, T., Unsal, O.S., Cristal, A., Hur, I. & Valero, M. (2010b). Discovering and understanding performance bottlenecks in transactional applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, 285–294, ACM, New York, NY, USA. 51