TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER FACULTY OF EXACT SCIENCES
THE BLAVATNIK SCHOOL OF COMPUTER SCIENCE

Lowering STM Overhead with Static Analysis

Dissertation submitted in partial fulfillment of the requirements for the M.Sc. degree in the Blavatnik School of Computer Science, Tel-Aviv University

by

Arie Zilberstein

The research work for this thesis has been carried out at Tel-Aviv University under the supervision of Prof. Yehuda Afek

September 2010

TEL-AVIV UNIVERSITY

Abstract

Raymond and Beverly Sackler Faculty of Exact Sciences
The Blavatnik School of Computer Science

Master of Science

by Arie Zilberstein

Software Transactional Memory (STM) is an emerging paradigm for concurrent programming. Recent research showed that software that uses STM can achieve good scalability while avoiding the complexity associated with traditional lock-based synchronization mechanisms. However, code instrumented by an STM compiler suffers from a large performance overhead, which results from transforming all memory accesses inside every transaction into calls to STM library functions. This work presents a study of compiler optimization techniques that attempt to drive the instrumentation overhead lower. We introduce each optimization, describe an implementation, and evaluate its effectiveness. We use standard tools from compiler theory, such as static analysis and code motion, to implement the optimizations. We start with some well-known optimizations, that are applicable to STMs of all types and designs; for example, reads of memory locations that are immutable can be completely exempt from instrumentation. However, our focus is on STMs that employ a lazy-update protocol, and for these we devise novel optimizations that exploit some of their unique properties. For example, reads of memory locations that have not been previously written to in the same transaction, need not check the writeset for the most updated value. Our optimizations are implemented over a TL2 Java-based STM system. We evaluate their effectiveness on a subset of the STAMP benchmarks and on some data-structure microbenchmarks. Our results show that some of the new optimizations can achieve up to 38% speedup, and up to 46-53% increased throughput on the tested workloads.

Acknowledgements

First, I would like to thank Yehuda Afek, my advisor. He provided guidance, support and constant encouragement throughout the thesis.

I am particularly grateful to my mentor Guy Korland. Many of the ideas presented in this work came out of fruitful discussions with Guy. His leadership, vision, ingenuity and technical abilities were absolutely vital to the creation of this thesis.

I'd like to thank Tania, who fills my life with joy every day, for her patience, caring, and love.

Thanks to my parents, friends, and colleagues, who always kept me going.

Contents

Abstract i

Acknowledgements ii

List of Figures v

List of Tables vi

Abbreviations vii

1 Introduction 1

2 Background 4
- 2.1 Deuce, A Java-Based STM . . . 4
- 2.2 Transactional Locking II . . . 5

3 Methodology 7
- 3.1 Implementing the Analyses . . . 7
- 3.2 Experimental Testbed . . . 7
- 3.3 Benchmarks . . . 8
  - 3.3.1 Data Structures Microbenchmarks . . . 8
  - 3.3.2 STAMP Benchmarks . . . 8

4 STM Optimizations - Existing Techniques 9
- 4.1 Immutable Memory . . . 9
  - 4.1.1 Introduction . . . 9
  - 4.1.2 Analysis . . . 10
  - 4.1.3 Applicability . . . 11
- 4.2 Thread Local Memory . . . 11
  - 4.2.1 Introduction . . . 11
  - 4.2.2 Analysis . . . 11
  - 4.2.3 Applicability . . . 12
- 4.3 Transaction Local Memory . . . 13
  - 4.3.1 Introduction . . . 13
  - 4.3.2 Analysis . . . 13
    - 4.3.2.1 In-Constructor Analysis . . . 13

4.3.2.2 After-Constructor Analysis 14
4.3.3 Applicability 15

5 STM Optimizations - Novel Techniques 17
5.1 Read-Only Transactions 17
5.1.1 Introduction 17
5.1.2 Analysis 17
5.1.3 Applicability 18
5.2 Load Elimination 19
5.2.1 Introduction 19
5.2.2 Analysis 20
5.2.3 Applicability 20
5.3 Scalar Promotion 21
5.3.1 Introduction 21
5.3.2 Analysis 21
5.3.3 Applicability 22
5.4 Redundant Writeset Lookups 22
5.4.1 Introduction 22
5.4.2 Analysis 22
5.4.3 Applicability 24
5.5 Redundant Writeset Record-Keeping 24
5.5.1 Introduction 24
5.5.2 Analysis 25
5.5.3 Applicability 25
5.6 Tightening Transaction Scopes 26
5.6.1 Introduction 26
5.6.2 Analysis 26
5.6.3 Applicability 28

6 Performance Evaluation 29
6.1 Optimization Levels 29
6.2 Data Structures Microbenchmarks 29
6.3 STAMP Benchmarks 31
6.4 Discussion 32

7 Related Work 34

8 Conclusions and Further Work 36

Bibliography 38

List of Figures

5.1 Transfer function τ: $\tau(t, a)$ is the node reachable by stepping from node t across the edge a. 23

6.1 Microbenchmarks comparative results. (higher is better) 30
6.2 STAMP Benchmarks comparative results. (lower is better) 31

List of Tables

4.1 % of reads of immutable memory per benchmark. 11
4.2 % of synchronization operations removed per benchmark. 13
4.3 % of transaction-local accesses detected per benchmark. 15

5.1 % of reads from memory not changed within the transaction per benchmark. 18
5.2 % of reads eliminated after applying PRE per benchmark. 20
5.3 % of reads from locations not written to before per benchmark. 24
5.4 % of writes to locations which will not be subsequently read from per benchmark. 25
5.5 % of reads hoised outside of transactions per benchmark. 28

Abbreviations

CFG	**C**ontrol **F**low **G**raph
IIP	**I**nitial **I**nitialization **P**oint
JVM	**J**ava **V**irtual **M**achine
LCM	**L**azy **C**ode **M**otion
PRE	**P**artial **R**edundancy **E**limination
IIP	**R**ecurring **I**nitialization **P**oint
SCC	**S**trongly **C**onnected **C**omponent
STAMP	**S**tanford **T**ransactional **A**pplications for **M**ulti-**P**rocessing
STM	**S**oftware **T**transactional **M**emory
TL2	**T**ransactional **L**ocking **II**

Chapter 1

Introduction

Software Transactional Memory (STM) [1, 2] is an emerging approach that provides developers of concurrent software with a powerful tool: the *atomic block*, which aims to ease multi-threaded programming and enable more parallelism. Conceptually, statements contained in an atomic block appear to execute as a single atomic unit: either all of them take effect together, or none of them take effect at all. In this model, the burden of carefully synchronizing concurrent access to shared memory, traditionally done using locks, semaphores, and monitors, is relieved. Instead, the developer needs only to enclose statements which access shared memory by an atomic block, and the STM implementation guarantees the atomicity of each block.

In the past several years there has been a flurry of software transactional memory design and implementation work; however, with the notable exception of transactional C/C++ compilers [3], many of the STM initiatives have remained academic experiments. There are several reasons for this; major among them is the large performance overhead [4]. In order to make a piece of code transactional, it usually undergoes an instrumentation process which replaces memory accesses with calls to STM library functions. These functions handle the bookkeeping of logging, committing, and rolling-back of values according to the STM protocol. Naïve instrumentation introduces redundant function calls, for example, for values that are provably transaction-local. In addition, an STM with homogeneous implementation of its functions, while general and correct, will necessarily be less efficient than a highly-heterogeneous implementation: the latter can offer specialized functions that handle some specific cases more efficiently. For example, a homogeneous STM may offer a general `STMRead()` method, while a heterogeneous STM may offer also a specialized `STMReadThreadLocal()` method that assumes that the value read is thread-local, and as a consequence, can optimize away validations of that value.

This work presents a study of STM-specific compiler optimizations. Each optimization is described, implemented and tested on a mixed testbed of application and micro-benchmarks.

Not all STM designs can benefit equally well from all of our optimizations. For example, STMs that employ in-place updates, rather than lazy updates, will see less benefit from the optimization that removes redundant memory reads. This is because in a lazy update STM, reading from a memory location entails looking up its most updated value in the writeset, as opposed to an in-place STM where the most updated value is always stored at the memory location itself. We therefore restrict our focus to the Transactional Locking II (TL2) [5] protocol, an in-place STM which benefits from all the optimizations.

We begin with a high-level description of our novel optimizations.

- **Avoiding redundant readset insertion on read-only transactions.** Transactions which do not modify any shared memory can be exempt from inserting its read memory locations to the readset. We provide an analysis that discovers such read-only transactions.
- **Avoiding redundant reads of memory locations which have been read before.** We use load elimination, a compiler technique that reduces the amount of memory reads by storing read values in local variables and using these variables instead of reading from memory. This allows us to reduce the number of costly STM library calls.
- **Avoiding redundant writes to memory locations which will be subsequently written to.** We use scalar promotion, a compiler technique that avoids redundant stores to memory locations, by storing to a local variable. Similar to load elimination, this optimization allows us to reduce the number of costly STM library calls.
- **Avoiding redundant writeset lookups for memory locations which have not been written to.** We discover memory accesses that read locations which have not been previously written to by the same transaction. Instrumentation for such reads can avoid writeset lookup.
- **Avoiding redundant writeset record-keeping for memory locations which will not be read.** We discover memory accesses that write to locations which will not be subsequently read by the same transaction. Instrumentation for such writes can therefore be made cheaper, e.g., by avoiding insertion to a Bloom filter.

- **Tightening a too large transaction scope.** We tighten the atomic block if we discover that statements at the beginning or end of it can be safely hoisted outside of the atomic block.

In addition to the new optimizations, we have implemented the following optimizations which have been used in other STMs.

- **Avoiding instrumentations of accesses to immutable and transaction-local memory.** Such accesses can directly access the memory locations instead of going through the STM library function call.
- **Avoiding expensive synchronization operations for thread-local memory.** Thread-local memory can be read without requiring validation, and can be updated without locking.

To summarize, this thesis makes the following contributions:

- We implement a set of common STM-specific analyses and optimizations.
- We present and implement a set of new analyses and optimizations to reduce overhead of STM instrumentation.
- We measure and show that our suggested optimizations can achieve significant performance improvements - up to 38% faster, and up to 46-53% more throughput in some workloads.

We proceed as follows: Chapter 2 gives a background of the STM we optimize. Chapter 3 describes the methodology of our work. Chapters 4 and 5 describe the STM compiler optimizations. In Chapter 4 we describe common STM compiler optimizations. Chapter 5 presents our new STM compiler optimizations, and is the main contribution of this thesis. Chapter 6 presents performance evaluation of the different optimizations. Chapter 7 reviews related work. We conclude in Chapter 8.

Chapter 2

Background

2.1 Deuce, A Java-Based STM

In this section we briefly review the underlying STM protocol that we aim to optimize. We use the Deuce Java-based STM framework. Deuce [6] is a pluggable STM framework which allows different implementations of STM protocols; a developer only needs to implement the `Context` interface, and provide his own implementation for the various STM library functions. The library functions specify which actions to execute on reading a field, writing to a field, committing a transaction, and rolling back a transaction.

Deuce is non-invasive: it does not modify the JVM (Java Virtual Machine) or the Java language, and it does not require to re-compile source code in order to instrument it. It works by introducing a new `@Atomic` annotation. Java methods which are annotated with `@Atomic` are replaced with a retry-loop that attempts to perform and commit a transacted version of that method. All methods are duplicated; the transacted copy of every method is similar to the original, except that all field and array accesses are replaced with calls to the `Context` interface, and all method invocations are rewritten so that the transacted copy is invoked instead of the original.

Deuce works either in *online* or *offline* mode. In online mode, the entire process of instrumenting the program happens during runtime. A *Java agent* is attached to the running program, by specifying a parameter to the JVM (Java Virtual Machine). During runtime, just before a class is loaded into memory, the Deuce agent comes into play and transforms the program in-memory. To read and rewrite classes, Deuce uses ASM [7], a general-purpose bytecode manipulation framework.

In order to avoid the runtime overhead of the online mode, Deuce offers the offline mode, which performs the transformations directly on compiled `.class` files. In this mode, the

program is transformed similarly, and the transacted version of the program is written into new `.class` files.

In this work we focus on the Transactional Locking II (TL2) protocol implementation in Deuce.

2.2 Transactional Locking II

Transactional Locking II (TL2) [5] is an influential STM protocol. In TL2, conflict detection is done by using a combination of versioned write-locks, associated with memory locations or objects, together with a global version clock. TL2 is a lazy-update STM, so values only written to memory at commit time; therefore locks are held for a very short amount of time.

We describe Deuce's word-based TL2 implementation; other variations [5, 8] exist. TL2 maintains a *global version clock* to achieve synchronization. When a transaction begins, it first *samples* the global clock. It then proceeds executing the transaction code speculatively, that is, without affecting the actual memory locations. Additionally, with every memory location there is an associated *versioned write-lock*.

Each memory write is replaced by a write to the *writeset*, a thread-local data structure holding pairs of memory locations and last written values.

Every memory read first checks to see if the read memory address already appears in the writeset, and if so, its written value is returned. Otherwise, the read memory location is inserted to the *readset*, another thread-local data structure, holding memory locations. The read is preceded by a *pre-validation* step, which checks that the version associated with the memory location is not greater than the sample, and that the associated lock is not locked. The read is followed by a *post-validation* step, which checks that the associated version is still the same as it was in the pre-validation step.

At *commit* time, locks are acquired for all items in the writeset, then the readset undergoes re-validation, which is equivalent to the pre-validation step. If the validation passes, the global clock is incremented, the items in the writeset are written to the shared memory, and the locks are released. Failure in any of the validations, or inability to acquire the necessary locks, will both *abort* the transaction: all acquired locks will be released, the readset and writesets will be cleared, and the transaction will be retried.

Deuce's version of TL2 supports *weak isolation* [9]; this means that a program accessing the same memory location both transactionally and non-transactionally may encounter

unexpected behavior. In a weakly-isolated STM, memory locations that are accessed from within transactions, should not be concurrently accessed from without transactions.

In addition, Deuce's TL2 version supports *flat nesting* [10]; this means that a transaction B, nested inside a transaction A, shares its readset and writeset with transaction A. B's changes are visible to other threads only after A commits. If B aborts, so does A. The effect is that transactions A and B form a single, flat transaction, hence the name.

Chapter 3

Methodology

3.1 Implementing the Analyses

In order to implement and test the STM compiler optimizations, we added optimization passes that operate before Deuce's instrumentation phase. The passes collect information about optimization opportunities, and in some cases (PRE-related optimizations) transform the code. The information is transferred to Deuce which uses it when instrumenting the code. The optimization passes are implemented using Soot [11], a powerful Java optimization framework.

Deuce's STM library is homogenous. In order to allow its methods to take advantage of specific cases where optimization is possible, we enhance each of its STM functions to accept an extra incoming parameter, *advice*. This parameter is a simple bit-set representing information that was pre-calculated in the optimization passes and may help fine-tune the instrumentation. For example, when writing to a field that will not be read, the advice passed to the STM write function will have 1 in the bit corresponding to "no-read-after-write".

3.2 Experimental Testbed

Our test environment is a Sun UltraSPARC T2 Plus multicore machine with 2 CPUs, each with 8 cores at 1.2 GHz, each core with 8 hardware threads to a total of 128 threads.

3.3 Benchmarks

We experimented on a set of data structure-based microbenchmarks and several benchmarks from the Java version [12] of the STAMP [13] suite.

The results for applicability were obtained on a single-threaded run of the benchmarks. Every benchmark was executed at least 10 times, and the median value was taken.

3.3.1 Data Structures Microbenchmarks

Our microbenchmarks exercised three different data structures: **LinkedList** represents a sorted linked list implementation. **SkipList** represents a skiplist with random leveling. **Hash** represents an open-addressing hash table which uses a fixed-size array with no rehashing. Every data structure supports three atomic operations: adding an item, removing an item, and checking for containment of an item. The test consists of threads attempting to perform as many atomic operations as possible on a shared data structure; each thread chooses its next operation randomly, with 90% chance selecting the lookup operations, and 10% chance selecting addition or removal. The threads are stopped after 20 seconds.

In addition, the microbenchmarks include **Bank**, an online bank simulation, in which different threads perform either a read-all transaction with 80% chance, a write-all transaction with 10% chance, or a transfer between two accounts with 10% chance.

In the microbenchmarks, we measure throughput, that is, the total number of operations performed.

3.3.2 STAMP Benchmarks

We tested four STAMP [13] benchmarks. **K-Means** implements k-means clustering. **Vacation** simulates an on-line travel reservation system. **Ssca2** performs several graph operations. In our tests we focused on Kernel 1, which generates a graph, and Kernel 2, which classifies large sets. **MatrixMul** is part of the Java version of the STAMP suite. It performs matrix multiplication. We could not test the other benchmarks from STAMP due to technical limitations (they are written in a special dialect of Java [14] and after conversion to standard Java they run with incorrect results).

In the STAMP benchmarks we measure the time it took for each test to complete. [1]

[1]Parameters used for STAMP benchmarks: K-Means: `-m 40 -n 40 -t 0.001 -i random-n16384-d24-c16.input`; Vacation: `-n 4 -t 5000000 -q 90 -r 65536 -u 80`; Ssca2: `-s 18`; MatrixMul:`130`

Chapter 4

STM Optimizations - Existing Techniques

In this chapter and the next, we present a number of STM compiler optimizations. Each optimization is described by:

- **Introduction:** An explanation of the optimization opportunity, usually with the help of a code example.
- **Analysis:** A static analysis algorithm that discovers the optimization opportunity.
- **Applicability:** A measure of how effective the optimization is when applied to the benchmarks. This measure is optimization-specific. For example, in optimizations that reduce the amount of synchronization operations, we measure how many such operations were removed; in optimizations that remove redundant memory reads, we measure how many such reads were removed; and so on.

The performance evaluation of the optimizations is discussed in Chapter 6.

4.1 Immutable Memory

4.1.1 Introduction

A shared memory location is *immutable* if different threads always observe the same value when reading it. A common STM compiler optimization is to avoid instrumenting reads of immutable memory. As an example, consider the `sumPair()` method in Listing 4.1. If we can statically deduce that `Pair`'s `x` and `y` fields are immutable, the STM can

directly access the two fields instead of replacing access to them by a call to an expensive STM library function.

```
@Atomic public sumPair(Pair p) {
  return p.x + p.y;
}
```

Listing 4.1: Immutable field access

Many works (e.g., [15]) consider a field to be immutable if it is declared with the Java `final` modifier. This is both dangerous and sub-optimal. It is dangerous because, surprisingly, fields annotated with `final` can still appear to change their value over time. Such a situation happens if an object which is not yet fully initialized is passed around to another thread. The other thread may observe an uninitialized (zero or null) value at first, while later reads of the same field will observe the intended initialized value. Detecting immutability with `final` fields is sub-optimal because it is certainly possible for a non-final field to have an immutable value according to our definition.

4.1.2 Analysis

The following algorithm computes the set S of immutable fields:

1. Compute S, the set of all fields of all classes.
2. Remove from S all fields that are written to in a method which is not a constructor or a static initializer.
3. Remove from S all instance fields `o.f` such that an assignment of the form `o.f = x` exists where `o` is different than `this`.
4. Remove from S all static fields C.f such that an assignment of the form `C.f = x` exists in a method which is not a `C` static initializer method.
5. Remove from S all fields of classes whose constructors expose `this`.

To enable the optimization, the set of all immutable fields is computed and stored before the STM instrumentation takes place. When a field access statement is about to be instrumented, we first check if that field is immutable. If so, we avoid instrumenting the field access.

4.1.3 Applicability

To understand the applicability of the Immutable Memory optimization, we measured for every benchmark, which percentage of all memory reads occuring in a transaction is in fact a read of an immutable memory location. The results are presented in Table 4.1.

Table 4.1: % of reads of immutable memory per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
50.0%	65.3%	50.4%	0.0%	0.0%	0.0%	25.0%	7.6%

We find that as many as 65.3% of a workload's memory reads are of immutable data. For example, in LinkedList, many of the accesses are reads of a node's value, which is set at the node's construction and never changes. In SkipList and Hash, many reads are of immutable arrays or of constants.

4.2 Thread Local Memory

4.2.1 Introduction

A runtime object `o` is *thread-local* if it can be accessed only by the thread that created it. Otherwise, it is *thread-shared*. Operations on thread-local objects are not in any risk for data race, because such objects may only be accessed from a single thread. However, operations on thread-local objects still need to be logged: in case of a transaction abort, either programmer-initiated or system-initiated, the object must revert to its pre-transaction state.

Assuming a TL2 STM, there are two valuable optimizations that we can enable when accessing thread-local memory. First, when reading a field off a thread-local object, there is no need to perform expensive pre- and post-validation, since the field's value cannot have been modified by a different thread. Second, when committing, there is no need to acquire or release any locks to protect thread-local memory. Locking, unlocking, and validating are all operations on shared memory, which require synchronization primitives; by optimizing them away we can gain considerable performance boost.

4.2.2 Analysis

In order to detect thread-local objects, we use the characterization of Rountev et al. [16]: an object o is thread-local if it is not reachable (via chains of field references) from any (i)

static fields, or (ii) fields of objects of classes that implement the `java.lang.Runnable` interface. Such objects are thus accessible only by the thread that created them. We conservatively consider all other objects as thread-shared.

In order to track objects throughout the program, we employ a *points-to analysis*. The points-to analysis is computed as a preliminary step before the thread escape analysis. Our implementation of points-to analysis produces, for each reference-typed variable v, a set $pta(v) = \{n_1, n_2, \ldots, n_k\}$ of *allocation site* nodes. Each node represents a unique `new` statement in the analyzed program.

Given a node $n \in pta(v)$, which represents an allocation site of an object of type T, we can obtain $n.f$ where f is a field of type T. $n.f$ is therefore the set of all nodes that $v.f$ may point to.

This is a description of our thread-escape algorithm.

Algorithm 1 Thread Shared Analysis

1: $A \leftarrow \emptyset$
2: **for all** method M **do**
3: **for all** assignment statement s in M **do**
4: **if** s is of the form `v.f = o` such that `v` implements `java.lang.Runnable` **then**
5: $A \leftarrow A \cup pta(v)$
6: **if** s is of the form `C.f = o` **then**
7: $A \leftarrow A \cup pta(o)$
8: **repeat**
9: **for all** node n in A **do**
10: **for all** field f in n **do**
11: $A \leftarrow A \cup n.f$
12: **until** A doesn't change
13: **return** A

The algorithm generates the set of all nodes that are pointed-to from variables which are either implementors of `java.lang.Runnable`, or directly accessible from static fields. Then, it computes the closure of that set by the field-reachability relation. The returned set A is therefore a conservative approximation of the set of all thread-shared objects. Its complement is the set of thread-local objects.

4.2.3 Applicability

To understand the applicability of the Thread Local Memory optimization, we measured how many synchronization operations can be removed due to the optimization. Locking and unlocking require a CAS operation, while validating the readset require reading from the shared locks array. The results are presented in Table 4.2.

Table 4.2: % of synchronization operations removed per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	3.4%

We see that accesses to thread-local memory are quite rare. This is an indication that the benchmarks are well-designed as most transactions access only shared memory. Vacation's thread-local memory accesses are to arrays which govern what random action each thread will perform within the transaction. Other benchmarks don't access thread-local memory inside transactions.

4.3 Transaction Local Memory

4.3.1 Introduction

Consider an object `o` created inside a transaction. Immediately after its creation, `o` is *transaction-local*: access to it is possible only from within the transaction. In lazy-update STMs, `o` remains transaction-local until the transaction is committed; this is not the case in in-place STMs, where `o` may be made visible to other threads even before the transaction is committed.

Since `o` is not visible to any other thread, there can be no data race when accessing its fields. And since it is completely disposed of in case of abort, there is no need to log previous or new field values. We can therefore make use of the transaction-local property by skipping the instrumentation of all fields of such objects.

4.3.2 Analysis

We provide two complementing analyses that statically discover access operations to fields of transaction-local objects: the first finds accesses to `this` inside constructors, and the second finds accesses to newly-constructed objects inside methods.

4.3.2.1 In-Constructor Analysis

The following algorithm computes the set A of self accesses to a transaction-local object in its own constructor.

Indeed, any read or write access to the `this` reference inside of a constructor is a transaction-local memory access.

Algorithm 2 In-Constructor Transaction-Local Analysis

1: $A \leftarrow \emptyset$
2: **for all** class C **do**
3: **for all** constructor M of class C **do**
4: **for all** statement s in M of the form `this.x = v` or `v = this.x` **do**
5: $A \leftarrow A \cup s$
6: **return** A

4.3.2.2 After-Constructor Analysis

The goal of the algorithm described here is to find the set of accesses to fields of transaction-local objects after their constructor method has finished constructing them. Typically, the fields of the newly-constructed object are initialized shortly after its creation.

We use an intraprocedural, flow-sensitive data flow analysis. The analysis tracks the creation of objects and accesses to fields. It uses the results of a *points-to analysis* that was pre-calculated. A points-to analysis associates every memory access expression e (of the form `o.f` or `a[i]`) with a set of abstract memory locations, $pta(e) \subseteq L$. Points-to analysis guarantees that two expressions e_1 and e_2 do not access the same memory location if and only if $pta(e_1) \bigcap pta(e_2) = \emptyset$.

The set of tags T is composed of the following elements: $\bot$ (memory location is not transaction-local, NEW (memory location is transaction-local), and $\top$ (memory location was not accessed yet). The tags are ordered: $\bot \leq NEW \leq \top$.

Each statement s in the program induces two *program points*, $\cdot s$ and $s\cdot$. The analysis associates each program point p with a set $S(p) \subseteq L \times T$ of tagged abstract memory locations. For a statement s, $S(\cdot s)$ represents the knowledge of the analysis just before executing s, and $S(s\cdot)$ represents the knowledge of the analysis just after executing s.

Each program point is initially associated with the full set of abstract memory locations, and each memory location is tagged $\top$.

The algorithm generates information in the following way. For each statement s, if it does not read from or write to any memory location, we set $S(s\cdot) \leftarrow S(\cdot s)$. Otherwise, let e be the memory access expression. We compute the set of pointed-to abstract memory locations $pta(e)$. For every $l \in pta(e)$ and $< l, t > \in S(\cdot s)$, we compute and set

$$S(s\cdot) \leftarrow (S(s\cdot) \setminus \{< l, t >\}) \cup \{< l, \tau(s) >\} \tag{4.1}$$

The *transfer function* τ inspects s. If s is an object creation expression, it returns NEW. If s is a copy statement, or a primitive field store or load (e.g., s is of the form `o.f = i` or `i = o.f` w here `i` is a primitive), τ returns t. For all other statement types s, τ conservatively returns $\perp$.

Let $pred(s)$ be the set of statement s's predecessors. The information is propagated forward according to the data flow equations defined by:

$$S(\cdot s) \leftarrow \{< l, \arg\min_{t'} \{< l, t' > \in S(p\cdot) : p \in pred(s)\} : l \in L\}. \tag{4.2}$$

The static analysis algorithm iterates over the program's Control Flow Graph (CFG) and applies equations 4.1, 4.2 until a fixed point is reached. This concludes the analysis.

Finally, we use the results of the analysis as follows. Consider a program statement s which accesses memory access expression e. Let $t = \min \{t' :< l, t' > \in S(\cdot s), l \in pta(e)\}$ be the minimum tag found among all memory locations potentially pointed to by e. If $t = NEW$ then we can access e without any instrumentation, since the object pointed-to by it was created in the same transaction.

4.3.3 Applicability

We measured for every benchmark, what percentage out of the total number of memory accesses are discovered as transaction-local accesses. We show the total of the two complementing analyses together. The results are presented in Table 4.3.

TABLE 4.3: % of transaction-local accesses detected per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
0.03%	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	3.2%

Many benchmarks, such as MatrixMul and K-Means, do not allocate any memory inside transactions; this optimization is irrelevant for them, and they gain nothing from it. Other benchmarks allocate memory inside transactions. Vacation's transactions, for example, add nodes to a red-black tree. According to our results, 3.2% of their memory accesses can be performed without any instrumentation.

It is especially revealing to measure the amount of memory *writes* that are transaction-local: the numbers are as high as 66.6% for LinkedList, 8.9% for SkipList, and 49.3% for Vacation. The high numbers demonstrate that this optimization is indeed strongly applicable to our benchmarks and suggest that it can be effective. We note that the numbers in Table 4.3 are much lower; this is because the table measures *accesses* (reads

and writes) rather than just writes, and these benchmarks perform significantly more reads than writes. It is much more common to write to a transaction-local memory location than it is to read from it.

Chapter 5

STM Optimizations - Novel Techniques

5.1 Read-Only Transactions

5.1.1 Introduction

A *read-only transaction* is a transaction which does not modify shared memory at all. Read-only transactions can execute with less overhead than a general transaction, because they do not need to populate and validate a readset [5]. This can represent a significant saving. Consider, for example, a typical linked list's `contains()` transaction, which traverses the list while looking for an item. If we detect that `contains()` is read-only, we can skip needless insertion of many of the list's elements to the readset.

We provide a static analysis that detects read-only transactions.

5.1.2 Analysis

Let method f be *in scope* if f is marked as an atomic method, or it is reachable (via some chain of method invocations) from an atomic method. In the following analysis, we consider only methods in scope.

Our goal is to detect and mark methods which are statically read-only, so they can be exempt from readset population. By static we mean that it is safe to skip readset population in them *for every call context* in which they are invoked. In other words, a read-only method participates only in read-only transactions; otherwise it would be

wrong to skip readset population in it, since it might be called from a non-read-only transaction.

Our algorithm is based around propagating the "non-read-only" property of methods in scope across the program's call graph. We begin by assigning the non-read-only property to all *locally* non-read-only methods: those methods which directly write to memory. The propagation is done according to the two following rules:

1. If a method f is not read-only, then so are all methods g that it invokes.

2. If a method f is not read-only, then so are all methods g which invoke it.

The reason for rule 1 is that any transaction which invokes f might invoke g, thereby making g participate in a non-read-only transaction. Similarly, the reason for rule 2 is that any transaction which invokes g might invoke f, thereby making g participate in a non-read-only transaction.

We use a simple worklist to do the propagation, starting from locally non-read-only methods. The analysis ends when the propagation hits a fixed-point. At that point, all the methods which are not marked as non-read-only are considered read-only, and can be marked as methods for which readset population can be avoided.

5.1.3 Applicability

To understand the applicability of the Read Only Transaction optimization, we optimized each of the benchmarks, and measured how many memory reads are reads that do not have to log their value to the readset. The percentages are displayed in Table 5.1.

Table 5.1: % of reads from memory not changed within the transaction per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
50.6%	85.9%	0.0%	82.6%	0.0%	0.0%	0.0%	0.0%

Our analysis was able to detect LinkedList's `contains()` method, SkipList's `contains()` method, and Bank's `computeTotal()` method as read-only methods. The three methods do not alter shared memory and can therefore skip inserting elements to the readset. We could not detect Hash's `contains()` method as read-only, even though it is in fact read-only, because it invokes an auxiliary method `index()` that searches the hash table. `index()` is used by other Hash methods, and could not be inlined in `contains()`.

5.2 Load Elimination

5.2.1 Introduction

Consider the following code fragment that is part of an atomic block (derived from the Java version of the STAMP suite):

```
for (int j = 0; j < nfeatures; j++) {
  new_centers[index][j] = new_centers[index][j] + feature[i][j];
}
```

A naïve STM compiler will instrument every array access in this fragment. However, the memory locations `new_centers[index]` and `feature[i]` are loop-invariant. We can calculate them once, outside the loop, and re-use the calculated values inside the loop. The technique of re-using values is a form of Partial Redundancy Elimination (PRE) optimization and is common in modern compilers. When PRE is applied to memory loads, it is called *Load Elimination*. The optimized version of the code will be equivalent to:

```
if (0 < nfeatures) {
  nci = new_centers[index];
  fi = feature[i];
  for (j = 0; j < nfeatures; j++) {
        nci[j] = nci[j] + fi[j];
  }
}
```

Many compilers refrain from applying the technique to memory loads (as opposed to arithmetic expressions). One of the reasons is that such a code transformation may not be valid in the presence of concurrency; for example, the compiler must make sure that the `feature[i]` memory location cannot be concurrently modified by a thread other than the one executing the above loop. This constraint, however, does not exist inside an atomic block, because the atomic block guarantees isolation from other concurrent transactions. An STM compiler can therefore enable PRE optimizations where they would not be possible with a regular compiler that does not support atomic blocks.

We note that this optimization is sound for all STM protocols that guarantee isolation. The performance boost achieved by it, however, is maximized with lazy-update STMs as opposed to in-place-update STMs. The reason is that lazy-update STMs must perform an expensive writeset lookup for every memory read, while in in-place-update STMs, memory reads are relatively cheap since they are done directly from memory. In fact, in such STMs, reads of memory locations that were read before can be optimized (Wang

et al. [17], Eddon and Herlihy [18]) to perform just a direct memory read together with a consistency check. By using load elimination of memory locations, memory reads are transformed into reads of a local variable; as a result, such reads are made much faster and a lazy-update STM operates at in-place STM speeds.

5.2.2 Analysis

We follow the classic Lazy Code Motion (LCM) ([19]) algorithm. This algorithm computes a few data flow analyses, and uses them to remove redundancies by assigning computed values into temporary variables and reusing them later. We apply LCM to field and array references, as well as to scalar expressions. For field and array references, we use a *side-effect analysis* to discover if the values do not change from one read to another. For example, consider the fragment `v1 = o1.f; o2.f = ...; v2 = o1.f;`. Can we eliminate the load into `v2` by reusing the value of `o1.f` that was previously loaded into `v1`? The answer depends on whether the store to `o2.f` may modify `o1.f` (that is, whether `o1` and `o2` may point to the same memory location); If it can, then it is not safe to eliminate the load to `v2`. The side-effect analysis provides us with answers to such questions.

5.2.3 Applicability

Table 5.2 shows the reduction in the number of instrumented memory reads as a result of applying our algorithm.

Table 5.2: % of reads eliminated after applying PRE per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
0.0%	27.6%	7.8%	1.2%	58.9%	0.0%	25.1%	1.8%

We see that some benchmarks (SkipList, K-Means, MatrixMul) contain many redundancies which can be removed. The PRE technique is ineffective on small and loop-less transactions (Ssca2) and on tightly written code with no redundancies (LinkedList). The advantage of PRE is that it does not require any modifications to the STM library functions.

5.3 Scalar Promotion

5.3.1 Introduction

The dual to load elimination is *Scalar Promotion.* Consider the following code fragment, also from STAMP:

```
for (int i = 0; i < num_elts; i++) {
  moments[0] += data[i];
}
```

If this fragment appeared inside an atomic method, an STM compiler could take advantage of the isolation property to eliminate the multiple writes to the same memory location. An optimized version of the code would be equivalent to:

```
if (0 < num_elts) {
  double temp = moments[0];
  try {
        for (int i = 0; i < num_elts; i++) {
          temp += data[i];
        }
  } finally {
      moments[0] = temp;
  }
}
```

The advantage of the optimized version is that multiple memory writes are replaced with just one.

5.3.2 Analysis

Our scalar promotion algorithm searches for loops in which a field or array reference is written to. For each written reference, it attempts to discover whether it is *promotable.* We will present the algorithm for field references; promoting array references is done in a similar manner. Consider, for example, a statement `o.f = ...` contained inside a loop. `o.f` is promotable if the two following conditions are satisfied: 1. `o` is never modified inside the loop, and 2. All writes to `o.f` are assignment statements of the exact form `o.f = ...`. The second condition is needed to invalidate the promotion in case `o.f` is modified by a method call or by a reference that points to the same object as `o`. In case `o.f` is found promotable, we transform the loop as follows: we add a preheader, in which we read `o.f`'s value into a new variable, `t`; we add a tail in which we store `t`'s value to `o.f`; and we replace all occurrences of `o.f` in the loop by `t`.

5.3.3 Applicability

Our analysis could not locate any instances where scalar promotion can be used. By examining the benchmarks code, we could not find examples where it was obvious that the same memory location was written to more than once. We expect that real production code will contain at least a few instances where this optimization is applicable; however this will occur at a much lower frequency than instances where PRE and other optimizations are possible.

5.4 Redundant Writeset Lookups

5.4.1 Introduction

Consider a field read statement `v = o.f` inside a transaction. The STM must produce and return the most updated value of `o.f`. In STMs that implement lazy update, there can be two ways to look up `o.f`'s value: if the same transaction has already written to `o.f`, then the most updated value must be found in the transaction's writeset. Otherwise, the most updated value is the one in `o.f`'s memory location. A naïve instrumentation will conservatively always check for containment in the writeset on every field read statement. With static analysis, we can gather information whether the accessed `o.f` was possibly already written to in the current transaction. If we can statically deduce that this is not the case, then the STM may skip checking the writeset, thereby saving processing time.

5.4.2 Analysis

We use an interprocedural, flow-sensitive data flow analysis. The analysis tracks the memory locations that are accessed by transactions, and their usage patterns. It can be seen as a compile-time simulation of the readset and writeset run-time activity.

The analysis uses the results of a *points-to analysis* that was pre-calculated. A points-to analysis associates every memory access expression e (of the form `o.f` or `a[i]`) with a set of abstract memory locations, $pta(e) \subseteq L$. Points-to analysis guarantees that two expressions e_1 and e_2 do not access the same memory location if and only if $pta(e_1) \bigcap pta(e_2) = \emptyset$.

The set of tags T is composed of the following elements: $\perp$ (memory location was written to), RO (memory location was read at least once, but not written to thereafter), and $\top$ (memory location was not accessed yet). The tags are ordered: $\perp \leq RO \leq \top$.

Each statement s in the program induces two *program points*, $\cdot s$ and $s\cdot$. The analysis associates each program point p with a set $S(p) \subseteq L \times T$ of tagged abstract memory locations. For a statement s, $S(\cdot s)$ represents the knowledge of the analysis just before executing s, and $S(s\cdot)$ represents the knowledge of the analysis just after executing s.

Each program point is initially associated with the full set of abstract memory locations, and each memory location is tagged $\top$.

The algorithm generates information in the following way. For each statement s, if it does not read from or write to any memory location, we set $S(s\cdot) \leftarrow S(\cdot s)$. Otherwise, let e be the memory access expression. Let $a \leftarrow \mathit{Read}$ if s reads from e, otherwise $a \leftarrow Write$. We compute the set of pointed-to abstract memory locations $\mathit{pta}(e)$. For every $l \in \mathit{pta}(e)$ and $< l, t > \in S(\cdot s)$, we compute and set

$$S(s\cdot) \leftarrow (S(s\cdot) \setminus \{< l, t >\}) \cup \{< l, \tau(t, a) >\} \tag{5.1}$$

where the *transfer function* τ is defined in the Figure 5.1.

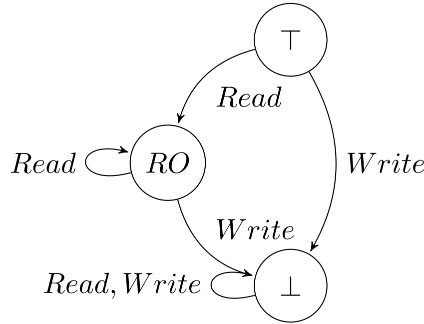

FIGURE 5.1: Transfer function τ: $\tau(t, a)$ is the node reachable by stepping from node t across the edge a.

Let $\mathit{pred}(s)$ be the set of statement s's predecessors. The information is propagated forward according to the data flow equations defined by:

$$S(\cdot s) \leftarrow \{< l, \arg\min_{t'} \{< l, t' > \in S(p\cdot) : p \in \mathit{pred}(s)\} : l \in L\}. \tag{5.2}$$

There is one exception to this propagation equation. if $\cdot s$ is a program point just before starting a new transaction, and all its predecessors are not part of a transaction, then we propagate $\top$ tags instead of the minimum tag seen over all predecessors. The reason is that when starting a transaction, the readset and writeset and empty.

The static analysis algorithm iterates over the program's Control Flow Graph (CFG) and applies equations 5.1, 5.2 until a fixed point is reached. This concludes the analysis.

Finally, we use the results of the analysis as follows. Consider a program statement s which reads memory access expression e. Let $t = \min \{t' :< l, t' >\in S(\cdot s), l \in pta(e)\}$ be the minimum tag found among all memory locations potentially pointed by e. If $t = RO$ then we know that in the transaction surrounding s, no memory location pointed to by e has been written to before the execution of s. Therefore when executing s there is no need to lookup e's value in the writeset.

5.4.3 Applicability

We measured the percentage, for each benchmark, of reads from memory locations, such that these memory locations have not been written to before in the same transaction. The results are presented in Table 5.3.

TABLE 5.3: % of reads from locations not written to before per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
100.0%	98.0%	100.0%	88.4%	80.2%	100.0%	100.0%	3.9%

Our analysis was successful on almost all benchmarks. Indeed, transactions usually read memory before writing to it. As an example of this optimization, consider how `add()` and `remove()` work in LinkedList, SkipList and Hash: They first locate the position to insert or remove the element, do the insertion or removal and finally perform some structural fixes. The first step (locating the position) encompasses the majority of memory reads, and since it precedes any memory writes, the optimization is applicable to it.

5.5 Redundant Writeset Record-Keeping

5.5.1 Introduction

Consider a field write statement `o.f = v` inside a transaction. An STM with lazy-update protocol must update the writeset with the information that `o.f` has been written to. One of the design goals of the writeset is that it should be fast to search it; this is because subsequent reads from `o.f` in the same transaction must use the value that is in the writeset. But, some memory locations in the writeset will never be actually read in the same transaction. We can exploit this fact to reduce the amount of record-keeping that the writeset data-structure must handle. As an example, TL2 suggests implementing the writeset as a linked-list (which can be efficiently added-to and traversed) together with a Bloom filter (that can efficiently check whether a memory location exists in the writeset). If we can statically deduce that a memory location is written-to but will not

be subsequently read in the same transaction, we can skip updating the Bloom filter for that memory location. This saves processing time, and is sound because there is no other purpose in updating the Bloom filter except to help in rapid lookups.

5.5.2 Analysis

Consider a statement s that writes to a memory access expression e. In order to find out whether there is no subsequent statement which reads from any of the memory locations pointed to by e, we create an "opposite" version of the previous analysis. The opposite version propagates information backwards rather than forwards. Its set of tags consists of: $\bot$ (memory location was read), WO (memory location was written to at least once, but not read thereafter), and $\top$ (memory location was not accessed yet). The tags are ordered: $\bot \leq WO \leq \top$. We omit the rest of the details since they are very similar to the previous analysis.

5.5.3 Applicability

We measured the percentage, for each benchmark, of writes to memory locations, such that these memory locations will not be subsequently read from in the same transaction. The results are presented in Table 5.4.

Table 5.4: % of writes to locations which will not be subsequently read from per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
100.0%	4.2%	100.0%	10.0%	5.3%	100.0%	100.0%	0.3%

We see that a majority of writes could in fact be detected as writes to memory locations which will not be read from in the same transaction. As a counter-example, consider Bank's `addInterest()` method, which atomically updates all the accounts. Each memory write, that updates the balance, is indeed not followed by a read of the same balance; however, the points-to analysis cannot distinguish between the multiple accounts. The result is that the analysis thinks that the same balance is read, written, re-read and re-written again and again; so the optimization is not applicable to the balance update.

5.6 Tightening Transaction Scopes

5.6.1 Introduction

The goal of this optimization is to shorten the lifetime of transactions. Consider the `minPair()` method at listing 5.1. This method atomically reads the two items of the incoming pair, and then computes their minimum. Assume that this method is never called from an atomic context. The statements where the minimum is computed need not, in fact, belong in a transaction: they contain no field accesses, array accesses, or method invocations. A *transaction boundaries* analysis will deduce that the instructions following `int y = p.y;` can be hoisted outside of the transaction.

```
@Atomic public void minPair(Pair p) {
  int x = p.x; int y = p.y;
  return x < y ? x : y;
}
```

Listing 5.1: Tightening transaction scope

Using transaction boundaries information, we can shorten the lifetime period when a transaction is active, by committing earlier. In the example, we can commit immediately after `int y = p.y;`, instead of waiting for the method to return to its caller and then commit (recall that in the naïve instrumentation, every atomic method is wrapped by a retry-loop, where the actual commit is performed). Shortening transactions helps in reducing contention: conflicts can be detected earlier, and less amount of wasted work would be done if the transaction is doomed to abort.

Symmetrically, we can find opportunities where we can shorten the transaction by initializing it later. This can happen, for example, if a transaction's first action is to allocate memory (e.g. allocate a new node for a linked-list). This allocation can happen outside the scope of the transaction. In TL2, a transaction begins by reading the shared global clock. Initializing the transaction later means that we may read a later value of the clock, thereby possibly avoiding collisions with other committing transactions.

5.6.2 Analysis

Let m be an atomic method. We seek to find all *initialization points (IPs)*: program points in m where it is safe to start the transaction. (Finding *commit points* requires minor modifications that "reverse" to algorithm; we omit the details.) We require that: 1. all (runtime) paths through m go through exactly one IP, 2. all (runtime) paths through m go through an IP before going through any other STM library calls, and 3.

the IPs are placed as late as possible. Once we have calculated the set of initialization points, we can optimize the instrumentation of m as follows: 1. remove the STM library's initialization call from m's retry-loop, and 2. insert STM library initialization calls in all IPs.

To make the "as late as possible" point concrete, we list the program statements that require no STM instrumentation and therefore can appear on a path before an IP. These are: 1. statements that do not access memory directly or indirectly; 2. statements that access immutable memory locations; 3. statements that access memory locations which were allocated in the same transaction as m; 4. statements which invoke methods which require no STM instrumentation.

The algorithm that finds IPs is composed of three major steps and operates on the strongly-connected components (SCC) graph of m. We use SCCs because IPs may never be placed inside loops.

In the first step, we detect all statements that must be instrumented in the program. Then, for every SCC of size larger than 1 that contains a must-instrument statement, we mark all its SCC predecessors as requiring instrumentation. This process repeats until we reach a predecessor SCC of size 1 (this is possibly m's head).

The second step is a simple forward intraprocedural data flow analysis over m's SCC graph. For every SCC s, it associates its two program points ($\cdot s$ and $s\cdot$) with a tag $t \in T$. T's elements are: $\bot$ (possibly initialized already), DI (definitely initialized already), and $\top$ (not initialized yet). Every SCC program point starts associated with $\top$. Every must-instrument SCC generates DI. Flow merge points generate $\bot$ if the tags corresponding to the incoming branches are not all the same, otherwise their common value. We propagate until convergence.

In the final step, let $S(p)$ be the tag associated with a program point p. Consider all SCCs s such that $S(s\cdot) = DI$. If $S(\cdot s) = \top$ then we mark s as an *initial initialization point (IIP)*. If $S(\cdot s) = \bot$ then we mark s as a *recurring initialization point (RIP)*.

After IIPs and RIPs have been calculated, the optimization proceeds by injecting calls to the STM library transaction initialization method at the beginning of every SCC marked IIP. The first step guarantees that such SCCs contain exactly one statement, so the injection placement is unambiguous. The second step guarantees that IIP program points are reached only by (runtime) paths that did not already initialize the transaction.

RIPs represent program points such that some paths leading to them may have initialized the transaction and some others may have not; therefore at every RIP we inject a *conditional* initialization statement, that initializes the transaction only if it was not

already initialized. This is accomplished by using a local boolean flag. In the presence of RIPs we must set this flag at all IIPs; if the analysis did not turn up any RIPs, the flag is not needed. As before, RIPs are only possible at SCCs of size 1.

5.6.3 Applicability

To measure the applicability of the transaction scope tightening optimization, we examined only cases where *memory access operations* could be hoisted outside of transactions – we ignored cases where only other instructions (i.e., arithmetic, branching or memory allocations) could be hoisted. This allows us to meaningfully measure and compare this optimization to other optimizations. Also, for this optimization to be effective, it must use information about immutable and transaction-local; so these two analyses were performed as well. We measure which percentage of memory reads could be hoised outside transactions. The results appear in Table 5.5.

Table 5.5: % of reads hoised outside of transactions per benchmark.

LinkedList	SkipList	Hash	Bank	K-Means	Ssca2	MatrixMul	Vacation
0.03%	7.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.03%

The most visible gain was found in SkipList, where we discovered 3 initiaization points and 1 commit point. Other benchmarks gained next to nothing. The low applicability of this optimization is due to two reasons: 1. Transactions with lengthy loops gain very little from optimizing just few instructions outside the loops. 2. The transactions that appear in the benchmarks are already tight. We note that our benchmarks did not contain any instances of RIP or recurring commit points.

Chapter 6

Performance Evaluation

In this chapter we present a performance evaluation of the different optimizations.

6.1 Optimization Levels

We compared 9 levels of optimizations. The levels are **cumulative** in that every level includes all the optimizations of the previous levels. The *None* level is the most basic code, which blindly instruments every memory access. The *+Immutable* level avoids instrumentation of accesses to immutable memory (Section 4.1). The *+TransactionLocal* level avoids instrumentation of accesses to transaction-local memory (Section 4.3). The *+ThreadLocal* level avoids synchronization instructions when accessing thread-local memory (Section 4.2). The *+ReadOnlyMethod* level avoids readset population in read-only transactions (Section 5.1). The *+PRE* level performs a load elimination optimization (Section 5.2). The *+WritesetLookup* level avoids redundant writeset lookups for memory locations which have not been written to (Section 5.4). The *+WritesetStorage* level avoids redundant writeset record-keeping for memory locations which will not be read (Section 5.5). Finally, the *+ScopeTightening* level shortens transactions by hoisting out instructions (Section 5.6).

6.2 Data Structures Microbenchmarks

The data structure benchmarks consist of LinkedList, SkipList, Hash and Bank (Section 3.3.1). The graphs in Figure 6.1 depict normalized throughput of these benchmarks. The graphs show normalized throughput as a function of both the number of threads

(the x-axis) and the optimization level applied (the different columns for the same x mark). Since we measure throughput, the higher the bar is, the better.

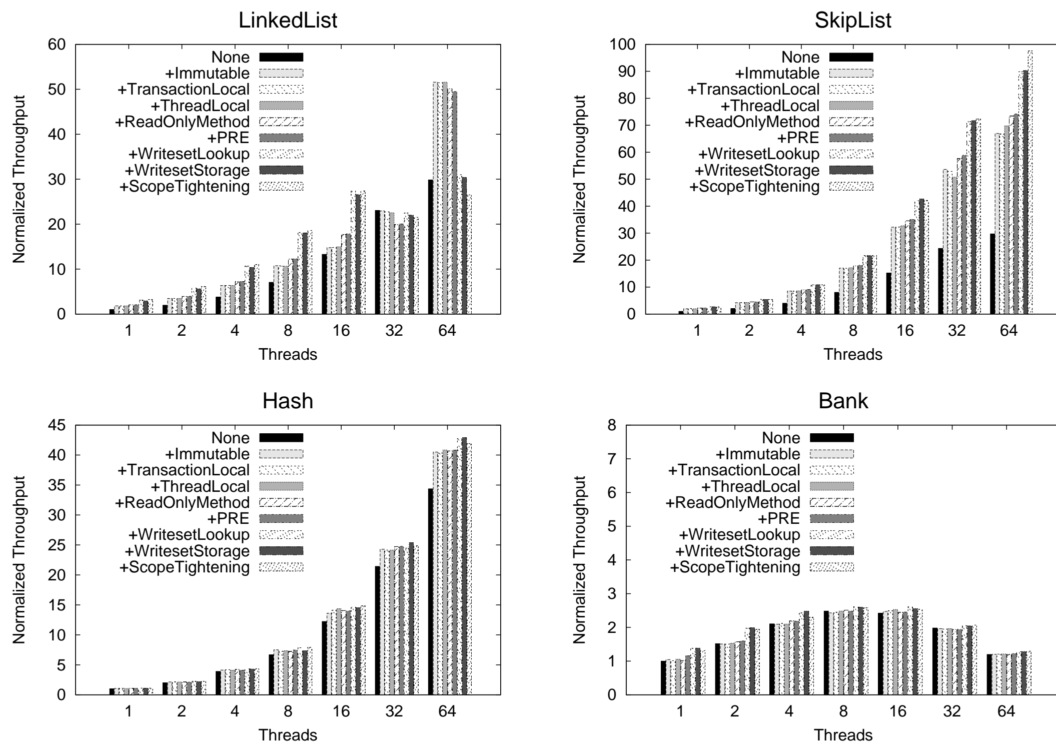

Figure 6.1: Microbenchmarks comparative results. (higher is better)

LinkedList benefited mostly from 3 optimizations. First, skipping instrumentation of immutable memory increased throughput dramatically: up to 82% in the single-threaded run and up to 73% in the 64-thread run. The ReadOnlyMethod optimization, which skips population of the readset in the `contains()` method, increased throughput by up to 14% (16 threads). Finally, the WritesetLookup optimization, which is applicable to all of LinkedList's transactioned memory reads, increased throughput by 46-53% (4-16 threads).

SkipList also benefited mostly from these 3 optimizations. Immutable boosted throughput by 95-125% (1-64 threads). ReadOnlyMethod increased throughput by 5-13% (32-64 threads). The WritesetLookup optimization increased throughput by around 20%. The effect of the ScopeTightening optimization was noticeable only in the 64-thread run, where it increased throughput by 8%. PRE increased throughput by 1-3%.

Hash saw almost no benefit from the optimizations at 1-4 threads. At higher amounts of threads, the most effective optimization was Immutable - up to 12-17% improvement.

Bank did not expose many opportunities for optimizations. The highest gain was from WritesetLookup (6-19% improvement at 1-16 threads). Even though the `computeTotal()` function was discovered as a ReadOnlyMethod, there was not much improvement (up

to 4% at 8 threads) because the readset size was very low (equal to the number of threads). Bank is the least scalable benchmark because even one transaction performing `addInterest()` will conflict and stall all other transactions.

6.3 STAMP Benchmarks

The STAMP benchmarks consist of K-Means, Vacation, Ssca2 and MatrixMul (Section 3.3.2). The graphs in Figure 6.2 depict normalized running time of these benchmarks. Similar to the data structure benchmarks, the graphs show normalized running time as a function of both the number of threads and the optimization level applied. In this case we measure running time, so the *lower* the bar is, the better.

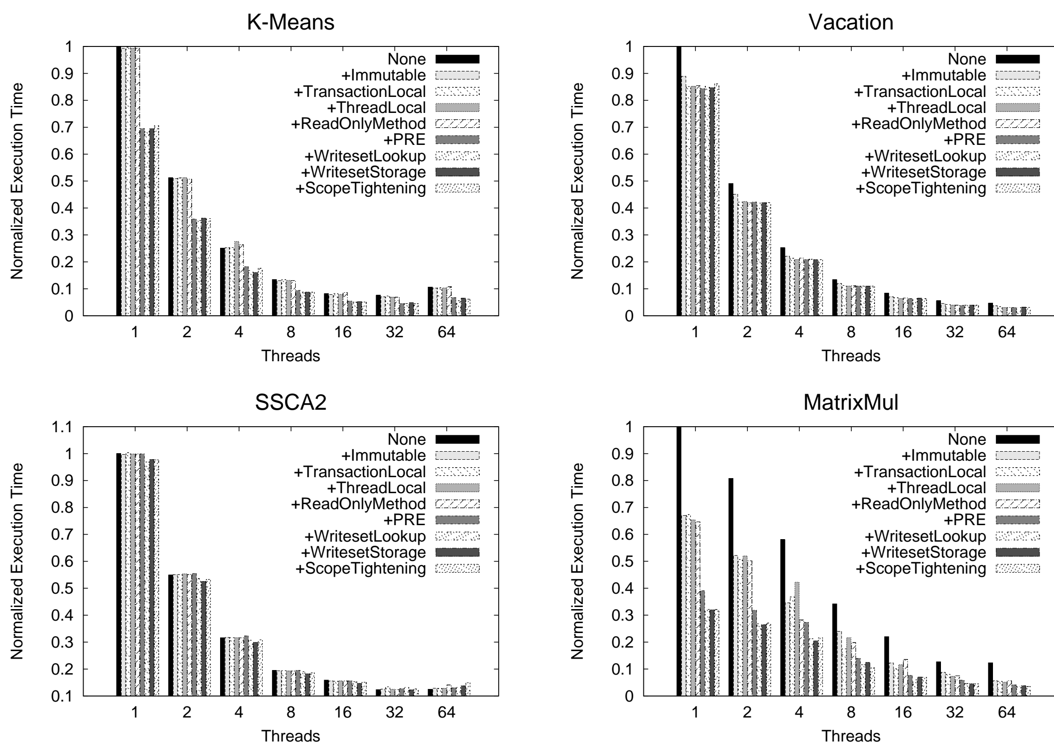

FIGURE 6.2: STAMP Benchmarks comparative results. (lower is better)

K-Means benefited greatly (28-38% speedup on all thread amounts) from PRE; the example from section 5.2 was taken directly from K-Means. Other optimizations were not successful in improving the runtime.

Vacation benefited from 2 main optimizations. The Immutable optimization sped up the benchmarks by 8-19% at all thread amounts. The TransactionLocal optimization, which avoids instrumentation of accesses to transaction-local memory, sped up the benchmarks by 3-9%. This makes sense, since this optimization was applicable to only 3.2% of the memory accesses. The ThreadLocal optimization, which removed 3.4% of

the synchronization operations, was effective only at high thread counts, where it sped up the tests by 3-9% (16-64 threads).

Ssca2 benefited mostly from two optimizations. The WritesetLookup optimization, which as we saw was applicable to all the read accesses, shortened the benchmark times by 2-5% (1-8 threads). The WritesetStorage optimization, applicable to all the write accesses, shortened the times by 0-8% (1-32) threads. Even though all the reads and writes were optimized, Ssca2 showed only modest improvement; this is because in Ssca2, transactions are short-lived with relatively small readsets and writesets [13].

MatrixMul, the most scalable benchmark, was amenable to 4 optimizations: Immutable (30-54% faster), PRE (3-43% faster), WritesetLookup (12-22% faster), and WritesetStorage (up to 4% faster).

6.4 Discussion

By examining the results we see that, among the well-known optimizations, the most impressive gains came from Immutable. The TransactionLocal and ThreadLocal optimizations have low applicability and therefore rarely had a noticable effect. These results are consistent with previous research [15].

Among the new optimizations, the three most powerful optimizations were WritesetLookup (up to 53% more throughput), PRE (up to 38% speedup), and ReadOnlyMethod (up to 14% more throughput).

The impact of the WritesetStorage optimization was relatively small. The reason that this optimization is less effective is that the insertion to the Bloom filter is already a very quick operation. In addition, our tested workloads have a relatively low amount of writes. The impact of ScopeTightening was also small, due to less applicability – transactions in the benchmarks were already tight.

The effectiveness of the optimizations varied widely with the different workloads and benchmarks. For example, the fully optimized LinkedList was 223% faster than the unoptimized version on 1 thread, and 106% faster on 16 threads. MatrixMul was 68% faster on a single-threaded run. However, Ssca2 showed only modest improvements on any number of threads due to short and small transactions.

Finally, we note that at some workloads, the optimizations produced a negative impact on performance. For example, the optimized version of LinkedList at 64 threads performed worse than the non-optimized version. We suspect that, on some specific workloads, making some transactions faster could generate conflicts with other advancing

transactions. Despite such isolated cases, the results are encouraging: new optimizations like WritesetLookup, PRE and ReadOnlyMethod were successful at significantly saving time and increasing throughput on most workloads.

Chapter 7

Related Work

The literature on compiler optimizations that are specific to transactional memory implementations revolves mostly around in-place-update STMs [20]. Harris et al. [21] presents the baseline STM optimizations that appear in many subsequent works. Among them are: Decomposition of STM library functions to heterogeneous parts; code motion optimizations that make use of this decomposition to reduce the number of "open-for-read" operations; early upgrade of "open-for-read" into "open-for-write" operation if a write-after-read will occur; suppression of instrumentation for transaction-local objects; and more. In [15], immutable objects are also exempt from instrumentation. In addition, standard compiler optimization techniques (see [22] for a full treatment), such as loop peeling, method inlining, and redundancy elimination algorithms are applied to atomic blocks.

Eddon and Herlihy [18] apply fully interprocedural analyses to discover thread-locality and subsequent accesses to the same objects. Such discoveries are exploited for optimized "fast path" handling of the cases. Similar optimizations also appear in Wang et al. [17], Dragojevic et al. [23]. We note that the above works optimize for in-place-update STMs. In such an STM protocol, once a memory location is "open-for-write", memory accesses to it are nearly transparent (free), because the memory location is exclusively owned by the transaction. Our work is different because it targets lazy-update STMs, where subsequent instrumented accesses to memory cannot be made much cheaper than initial accesses; e.g., the writeset must still be searched on every read. We solve this problem by transforming instrumented reads and writes, that access shared memory, into uninstrumented reads and writes that access local variables.

Spear et al. [20] proposes several optimizations for a TL2-like STM: 1. When multiple memory locations are read in succession, each read is instrumented such that the location is pre-validated, read, and then post-validated. By re-ordering the instructions such that

all the pre-validations are grouped together, followed by the reads, and concluded by the post-validations, they increase the time window between memory fences, such that the CPU could parallelize the memory reads. 2. Post-validation can sometimes be postponed as long as working with "unsafe" values can be tolerated; This eliminates or groups together expensive memory barrier operations. Shpeisman et al. [24]'s *barrier aggregation* is a similar, but simpler, optimization that re-uses barriers inside a basic block if they guard the same object.

Beckman et al. [25]'s work provides optimizations for thread-local, transaction-local and immutable objects that are guided by *access permissions*. These are Java attributes that the programmer must use to annotate program references. For example, the `@Imm` attribute denotes that the associated reference variable is immutable. Access permissions are verified statically by the compiler, and then used to optimize the STM instrumentation for the affected variables.

Partial redundancy elimination (PRE) [19, 26] techniques are widely used in the field of compiler optimizations; however, most of the focus was at removing redundancies of arithmetic expressions. Fink et al. [27] and Hosking et al. [28] were the first to apply PRE to Java access path expressions, for example, expressions like a.b[i].c . This variant of PRE is also called *load elimination*. As a general compiler optimization, this optimization may be unsound because it may miss concurrent updates by a different thread that changes the loaded value. Therefore, some works [29, 30] propose analyses that detect when load elimination is valid. *Scalar promotion*, which eliminates redundant memory writes, was introduced by Lu and Cooper [31], and improved by later works (e.g. [32]).

Chapter 8

Conclusions and Further Work

We described, implemented and tested a selection of STM-specific optimizations. Most of the optimizations are suitable for every STM protocol, but their effectiveness is greater on lazy-update STMs.

In addition to implementing well-known optimizations, we presented some new ones. **First**, we provided an analysis to discover read-only transactions. **Second**, we showed that two pre-existing optimizations, load elimination and scalar promotion, can be used in an optimizing STM compiler. Where standard compilers need perform an expensive cross-thread analysis to enable these optimizations, an STM compiler can rely on the atomic block's isolation property to enable them. **Third**, we highlighted two redundancies in STM read and write operations, and showed how they can be optimized. **Last**, we showed how to shorten transactions by hoisting statements outside of atomic blocks where it is safe to do so.

We implemented a compiler pass that performs these STM-specific code motion optimizations, and another pass that uses static analysis methods to discover optimization opportunities for redundant STM read and write operations. We have augmented the interface of the underlying STM compiler, Deuce, to accept information about which optimizations to enable at every STM library method call, and modified the STM methods themselves to apply the optimizations when possible.

The combined performance benefit of all the optimizations presented here varies with the workload and the number of threads. While some benchmarks see little to no improvement (e.g., Ssca2), we have observed speedups of up to 38% and throughput increases of up to 46-53% in other benchmarks.

There are many ways to improve upon this research. For example, a drawback of the optimizations presented here is that they require full interprocedural analysis to make

sound decisions. It may be interesting to research which similar optimizations can be enabled with less analysis work, for example, with running only intraprocedural analyses, or with partial analysis data that is calculated at runtime. In addition, it may be possible to improve the applicability of our optimizations, by basing them on stronger general-purposes analyses. For example, many optimizations that make use of a points-to analysis, can possibly benefit by employing a more precise (e.g., object-sensitive) points-to analysis techniques.

Bibliography

[1] Maurice Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[2] Nir Shavit and Dan Touitou. Software transactional memory. In *"Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)"*, pages "204–213", 1995.

[3] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA '08: Proceedings of the 23rd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 195–212, 2008.

[4] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1400214.1400228.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[6] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with Java STM. In *MultiProg '10: Programmability Issues for Heterogeneous Multicores*, January 2010. Further details at `http://www.deucestm.org/`.

[7] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[8] Hillel Avni and Nir Shavit. Maintaining consistent transactional states without a global clock. In *SIROCCO '08: Proceedings of the 15th international colloquium*

on Structural Information and Communication Complexity, pages 131–140, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69326-0. doi: http://dx.doi.org/10.1007/978-3-540-69355-0_12.

[9] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006. ISSN 1556-6056. doi: http://dx.doi.org/10.1109/L-CA.2006.18.

[10] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 70–81, New York, NY, USA, 2006. ACM. ISBN 1-59593-578-9. doi: http://doi.acm.org/10.1145/1178597.1178610.

[11] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[12] Brian Demsky and Alokika Dash. Evaluating contention management using discrete event simulation. In *Fifth ACM SIGPLAN Workshop on Transactional Computing*, April 2010.

[13] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[14] URL `http://demsky.eecs.uci.edu/software.php`.

[15] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1133985.

[16] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. pages 43–55. ACM Press, 2001.

[17] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: http://dx.doi.org/10.1109/CGO.2007.4.

[18] Guy Eddon and Maurice Herlihy. Language support and compiler optimizations for stm and transactional boosting. In *ICDCIT*, pages 209–224, 2007.

[19] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Not.*, 27(7):224–234, 1992. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/143103.143136.

[20] Michael F. Spear, Maged M. Michael, Michael L. Scott, and Peng Wu. Reducing memory ordering overheads in software transactional memory. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 13–24, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0. doi: http://dx.doi.org/10.1109/CGO.2009.30.

[21] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Conference on Programming Language Design and Implementation*, ACM SIGPLAN, pages 14–25, June 2006.

[22] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[23] Aleksandar Dragojevic, Yang Ni, and Ali-Reza Adl-Tabatabai. Optimizing transactions for captured memory. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 214–222, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: http://doi.acm.org/10.1145/1583991.1584049.

[24] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. *SIGPLAN Not.*, 42(6):78–88, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1273442.1250744.

[25] Nels E. Beckman, Yoon Phil Kim, Sven Stork, and Jonathan Aldrich. Reducing STM overhead with access permissions. In *IWACO '09: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pages

1–10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-546-8. doi: http://doi.acm.org/10.1145/1562154.1562156.

[26] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359060.359069.

[27] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174, London, UK, 2000. Springer-Verlag. ISBN 3-540-67668-6.

[28] Antony L. Hosking, Nathaniel Nystrom, David Whitlock, Quintin I. Cutts, and Amer Diwan. Partial redundancy elimination for access path expressions. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 138–141, London, UK, 1999. Springer-Verlag. ISBN 3-540-66954-X.

[29] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 41–52, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. doi: http://dx.doi.org/10.1109/PACT.2009.32.

[30] Christoph von Praun, Florian Schneider, and Thomas R. Gross. Load elimination in the presence of side effects, concurrency and precise exceptions. In *In Proceedings of the International Workshop on Compilers for Parallel Computing (LCPC03*, 2003.

[31] John Lu and Keith D. Cooper. Register promotion in C programs. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258943.

[32] Byoungro So and Mary W. Hall. Increasing the applicability of scalar replacement. In *CC*, pages 185–201, 2004.