Lowering STM overhead with Static Analysis

Yehuda Afek, Guy Korland, and Arie Zilberstein

Computer Science Department, Tel-Aviv University, Israel
afek@cs.tau.ac.il,guy.korland@cs.tau.ac.il,zilbers@post.tau.ac.il

Abstract. Software Transactional Memory (STM) compilers commonly instrument memory accesses by transforming them into calls to STM library functions. Done naïvely, this instrumentation imposes a large overhead, slowing down the transaction execution. Many compiler optimizations have been proposed in an attempt to lower this overhead. In this paper we attempt to drive the STM overhead lower by discovering sources of sub-optimal instrumentation, and providing optimizations to eliminate them. The sources are: (1) redundant reads of memory locations which have been read before, (2) redundant writes to memory locations which will be subsequently written to, (3) redundant write-set lookups of memory locations which have not been written to, and (4) redundant writeset record-keeping for memory locations which will not be read. We describe how static analysis and code motion algorithms can detect these sources, and enable compile-time optimizations that significantly reduce the instrumentation overhead in many common cases. We implement the optimizations over a TL2 Java-based STM system, and demonstrate the effectiveness of the optimizations on various benchmarks, measuring up to 29-50% speedup in a single-threaded run, and up to 19% increased throughput in a 32-threads run.

Key words: Transactional Memory, Optimization, Static Analysis

1 Introduction

Software Transactional Memory (STM) [12, 20] 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 [19], many of the STM initiatives have remained academic experiments. There are several reasons for this; major among them is the large performance overhead [6]. 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 book-keeping 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.

Previous work has presented many compiler and runtime optimizations that aim to reduce the overhead of STM instrumentation. In this work we add to that body of knowledge by identifying additional new sources of sub-optimal instrumentation, and proposing optimizations to eliminate them. The sources are:

- **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.
- **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.
- **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.
- **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.

Not all STM designs can benefit equally well from all the optimizations listed; For example, STMs that employ in-place updates, rather than lazy updates, will see less benefit from the redundant memory access optimization. From here on, we restrict the discussion to the TL2 protocol, which benefits from all of the optimizations.

In addition to the new optimizations, we have implemented the following optimizations which have been used in other STMs: 1. Avoiding instrumentations of accesses to immutable and transaction-local memory; 2. Avoiding lock acquisitions and releases for thread-local memory; and 3. Avoiding readset population in read-only transactions.

To summarize, this paper 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 29-50% speedup in some workloads.

We proceed as follows: Section 2 gives a background of the STM we optimize. In Section 3 we describe the optimization opportunities that our analyses expose. In Section 4 we measure the impact of the optimizations. Section 5 reviews related work. We conclude in Section 6.

2 Background - 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 [16] 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 take 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 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. 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 [4], 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.

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 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".

In this work we focus on the Transactional Locking II (TL2) protocol implementation in Deuce. TL2 is word-based, lock-based and uses the lazy update strategy. Full details can be found at [7].

3 Optimization Opportunities

The following are optimization opportunities that we have detected.

3.1 Preventing Redundant Memory Accesses

Load Elimination 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 instrument every memory access, while in in-place-update STMs it suffices to instrument just the first memory access. Repeated accesses to the same memory locations are transparent.

Using PRE of memory locations with a lazy-update STM, evens the ground: the first memory access is instrumented, and its value is stored in a temporary variable. This variable is later re-used, thereby achieving "transparent" memory access, that does not warrant another costly access through the STM library function.

Scalar Promotion 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.

3.2 Preventing Redundant Writeset Operations

Redundant Writeset Lookups 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.

Redundant Writeset Record-Keeping Consider a field write statement `o.f = v` inside a transaction. According to the TL2 protocol, the STM 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.

4 Experimental Results

In order to test the benefit of the above optimization opportunities, we used Deuce [16], a Java-based STM framework.

PRE optimizations (section 3.1) require no change to the actual Deuce runtime; they only require an extra preliminary optimization pass.

The optimization of preventing redundant writeset operations (section 3.2)) needs to actually change the instrumentation. To do it, we enhance each of Deuce's STM library methods to accept an extra bit-set parameter, *advice*, every bit of which denotes an optimization opportunity. Our compile-time analyses discover the opportunities and supply the *advice* parameters to the STM library method calls. The STM library methods detect the enabled bits in the *advice* parameters and apply the relevant optimizations. Specifically, the STM read method, upon seeing a 1 in the bit corresponding to "no-write-before-read", will avoid looking up the memory location in the writeset. Similarly, the STM write function, upon seeing a 1 in the bit corresponding to "no-read-after-write", will avoid updating the Bloom filter.

Out 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.

4.1 Optimization Levels

We compared 5 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 *Common* level adds several well-known optimizations that are common in STMs. These include 1. Avoiding instrumentations of accesses to immutable and transaction-local memory; 2. Avoiding lock acquisitions and releases for thread-local memory; and 3. Avoiding readset population for read-only transactions. The *PRE* level consists of load elimination and scalar promotion optimizations. The *ReadOnly* level avoids redundant readset lookups for memory locations which have not been written to. Finally, the *WriteOnly* level avoids redundant writeset record-keeping for memory locations which will not be read.

4.2 Benchmarks

We experimented on a set of data structure-based microbenchmarks and several benchmarks from the Java version of the STAMP [5] suite.

Data Structures Microbenchmarks In the following microbenchmarks, we exercised three different data structures, with separate threads adding, removing and looking up items. Each test ran for 20 seconds and performed 10% write operations. *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.

In the microbenchmarks, we measure throughput, that is, the total number of operations performed. Each value is normalized relative to the results of a single-threaded run with no optimizations. The results appears in Figure 1. Each bar represents the median value of at least 10 runs.

STAMP Benchmarks We tested four STAMP [5] 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.

In the STAMP benchmarks we measured the time it took for each test to complete. [1] As before, the values are normalized relative to the single-threaded, no-optimizations run. The results appears in Figure 2.

[1] Parameters used for the 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`

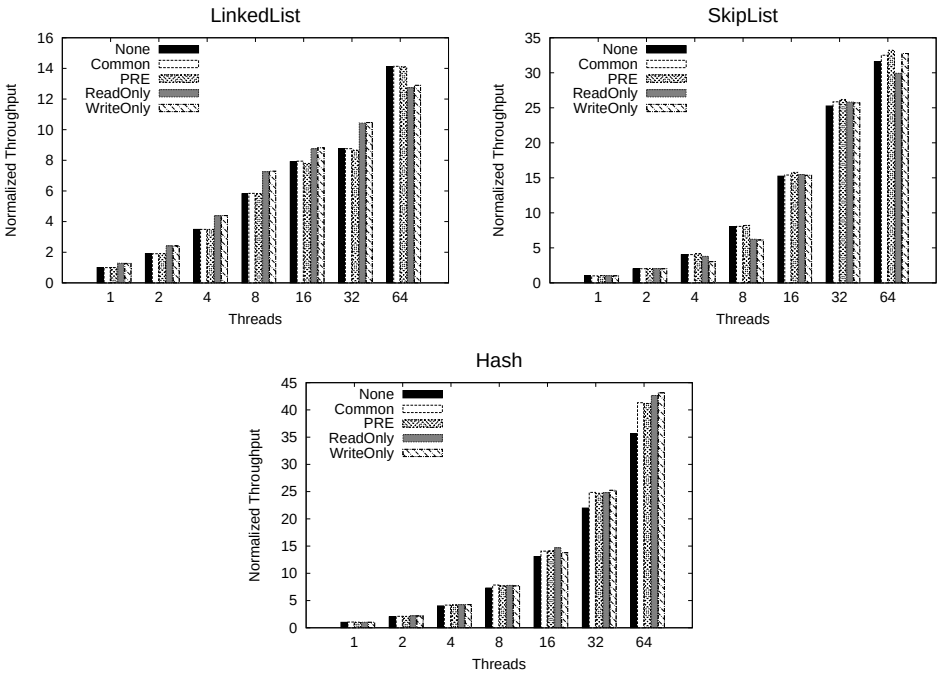

Fig. 1. Microbenchmarks comparative results. (higher is better)

Table 1. Optimization opportunity measures per microbenchmark on a single-threaded run.

	LinkedList	SkipList	Hash
PRE (% of reads eliminated)	0.17%	0.48%	0%
ReadOnly (% of reads from locations not written to before)	100%	3.56%	100%
WriteOnly (% of writes to locations not read from thereafter)	100%	0%	100%

4.3 Optimization Opportunities Breakdown

To understand to what extent optimizations are applicable to the benchmarks, we compared optimization-specific measures on single-threaded runs. The results appear in tables 1, 2. The measure for PRE is the percent of reads eliminated by load elimination and scalar promotion (compared to the Common level). The measure for ReadOnly is the percent of read statements that access memory locations which have not been written to before in the same transaction. The measure for WriteOnly is the percent of write statements that write to memory which will not be read in the same transaction. All numbers are measured dynamically at runtime. High percentages represent more optimization opportunities. Low percentages mean that we could not locate many optimization opportunities, either because they do not exist, or because our analyses were not strong enough to find them.

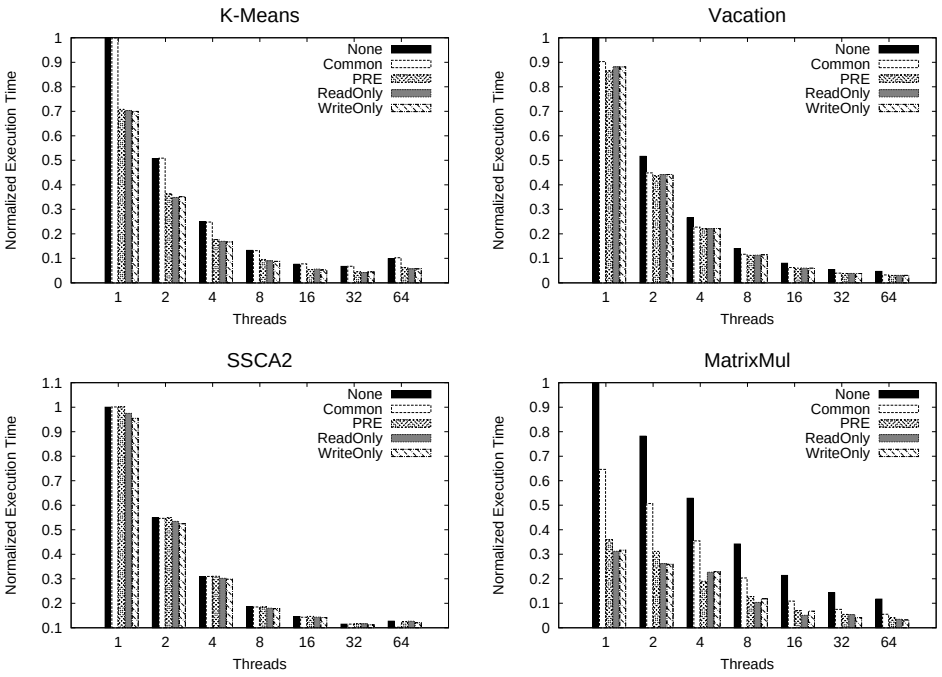

Fig. 2. STAMP Benchmarks comparative results. (lower is better)

Table 2. Optimization opportunity measures per STAMP benchmark on a single-threaded run.

	K-Means	Vacation	SSCA2	MatrixMul
PRE (% of reads eliminated)	56.86%	4%	0%	33.46%
ReadOnly (% of reads from locations not written to before)	56.05%	1.56%	100%	100%
WriteOnly (% of writes to locations not read from thereafter)	5.26%	0.02%	100%	100%

4.4 Analysis

Our benchmarks show that the optimizations have improved performance to varying degrees. The most noticeable performance gain was due to PRE, especially in tight loops where many memory accesses were eliminated.

PRE K-Means benefits greatly (up to 29% speedup) from load elimination: the above example (section 3.1) is taken directly from K-Means. MatrixMul also benefits from PRE due to the elimination of redundant reads of the main matrix object. Vacation achieves 4% speedup in the single-threaded run, but sees little to no speedup as the number of threads rises; this is because the eliminated loads exist outside of tight loops.

Our Scalar Promotion analysis, which focuses on finding loops where the same memory location is re-written in every iteration, was not able to find this pattern

in any of the tested benchmarks. A more thorough analysis, that also considers writes outside of loops, may have been able to detect some opportunities for enabling the Scalar Promotion optimization.

ReadOnly LinkedList benefits (throughput increased by at up 28%) from the ReadOnly optimization, which applies to reading the next node in the list. This optimization is valid since traversal is done prior to updating the next node. We note that the read of the head of the list is also a read-only memory access; however this is subsumed by the Common optimizations because the head is immutable. Hash's throughput is increased by up to 4% due to ReadOnly opportunities in the `findIndex()` method, which is called on every transaction. SSCA2 and MatrixMul see modest benefits.

Our analysis discovered that in 4 benchmarks: LinkedList, Hash, SSCA2 and MatrixMul, all reads are from memory locations which have not been written to before in the same transaction. We suspect that reading before writing is the norm in almost all transactions, but our analyses could prove it only in these 4.

WriteOnly The WriteOnly optimization is effective on transactions with a high number of successful writes. Hash, which makes over 11 million writes in the single-threaded run, shows up to 1.7% improvement, while SSCA2 with a similar number of writes, is up to 4% faster. Other benchmarks with 100% optimization of writes, MatrixMul and LinkedList, perform much less writes (only around 15,000) and therefore see very little benefit from the WriteOnly optimization.

We conclude that PRE shows the most impressive gains while ReadOnly follows next. The impact of the WriteOnly optimization is relatively small. The reason that the WriteOnly 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 effectiveness of the optimizations varies widely with the different workloads and benchmarks. For example, the fully optimized LinkedList is 27% faster (compared to Common) on 1 thread, and 19% faster on 32 threads. MatrixMul is 50% faster on a single-threaded run. However, SkipList and Vacation shows no more than 1% gain on any number of threads due to lack of optimization opportunities.

While generally the optimizations improve throughput and save time, at some workloads their effects are detrimental. For example, the optimization versions of LinkedList on 64 threads, or SkipList on 8 threads, perform worse than their non-optimized versions. We suspect that, on some specific workloads, making some transactions faster could generate conflicts with other advancing transactions.

5 Related Work

The literature on compiler optimizations that are specific to transactional memory implementations revolves mostly around in-place-update STMs [22]. Harris

et al. [11] 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 [1], immutable objects are also exempt from instrumentation. In addition, standard compiler optimization techniques (see [14] for a full treatment), such as loop peeling, method inlining, and redundancy elimination algorithms are applied to atomic blocks. Eddon and Herlihy [9] apply fully interprocedural analyses to discover thread-locality and subsequent accesses to the same objects. Such discoveries are exploited for "fast path" handling of the cases. Similar optimizations also appear in Wang et al. [24], Dragojevic et al. [8]

We note that the above works optimize for in-place-update STMs. In such an STM protocol, once an object is "open-for-write", memory accesses to its fields are transparent (free), because the object is exclusively owned by the transaction. Our work is different because it targets lazy-update STMs, where this form of optimization is invalid. A lazy-update STM keeps a writeset where it gathers all the memory location writes that occur within the transaction. It does not update the memory in-place; this happens only at commit time. Therefore, we still need to instrument even subsequent memory accesses, because they cannot transparently access the memory locations themselves. We solve this problem by working with local copies of memory locations, which require no instrumentation.

Spear et al. [22] proposes several optimizations for a TL2-like STM: 1. When multiply 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.

Beckman et al. [3]'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 ([15, 18]) (PRE) 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. [10] and Hosking et al. [13] 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 [2, 23] propose analyses that detect when load elimination is valid. *Scalar promotion*, which eliminates redundant memory writes, was introduced by Lu and Cooper [17], and improved by later works (e.g. [21]).

6 Conclusions and Further Work

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. We also highlighted two redundancies in STM read and write operations, and showed how they can be optimized.

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 and SkipList), we have observed speedups of up to 50% and 29% in other benchmarks (single-threaded MatrixMul and K-Means, respectively).

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.

Acknowledgments. This paper was supported in part by the European Union grant FP7-ICT-2007-1 (project VELOX) and by grants from Intel Corporation and Sun Microsystems.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. 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.

[2] R. Barik and V. 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.

[3] N. E. Beckman, Y. P. Kim, S. Stork, and J. 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.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[6] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. 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.

[7] 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.

[8] A. Dragojevic, Y. Ni, and A.-R. 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.

[9] G. Eddon and M. Herlihy. Language support and compiler optimizations for stm and transactional boosting. In *ICDCIT*, pages 209–224, 2007.

[10] S. J. Fink, K. Knobe, and V. 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.

[11] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Conference on Programming Language Design and Implementation*, ACM SIGPLAN, pages 14–25, June 2006.

[12] M. 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.

[13] A. L. Hosking, N. Nystrom, D. Whitlock, Q. I. Cutts, and A. 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.

[14] K. Kennedy and J. 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.

[15] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *SIGPLAN Not.*, 27(7):224–234, 1992. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/143103.143136.

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

[17] J. Lu and K. 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.

[18] 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.

[19] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. 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.

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

[21] B. So and M. W. Hall. Increasing the applicability of scalar replacement. In *CC*, pages 185–201, 2004.

[22] M. F. Spear, M. M. Michael, M. L. Scott, and P. 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.

[23] C. von Praun, F. Schneider, and T. 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.

[24] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. 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.