Optimizing Transactions for Captured Memory

Aleksandar Dragojević*
EPFL, Lausanne
Switzerland
aleksandar.dragojevic@epfl.ch

Yang Ni
Intel Labs
Santa Clara, CA 95054
yang.ni@intel.com

Ali-Reza Adl-Tabatabai
Intel Labs
Santa Clara, CA 95054
ali-reza.adl-tabatabai@intel.com

ABSTRACT

In this paper, we identify transaction-local memory as a major source of overhead from compiler instrumentation in software transactional memory (STM). Transaction-local memory is memory allocated inside a transaction, which cannot escape (i.e., is *captured* by) the allocating transaction. Accesses to such memory do not require calls to STM memory access functions (i.e., *STM barriers*). A compiler unaware of that may translate accesses to captured memory into expensive STM barriers. This presents us opportunities to improve STM performance. Our measurements with the STAMP benchmark suite (version 0.9.9) revealed that as many as 60% of the STM barriers generated by our baseline compiler access captured memory, including 90% of the write barriers and 45% of the read barriers. We propose runtime and compiler optimizations to elide STM barriers to captured memory. These techniques can also elide barriers for accesses to thread-local and read-only data. We implemented those optimizations in the Intel C++ STM compiler. Our experiments with the STAMP benchmark suite on a Intel Dunnington system (with 24 cores in a 4-node SMP system) show that these optimizations can improve performance by to 18% at 16 threads.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures

General Terms

Algorithms, Languages, Performance

Keywords

Software Transactional Memory, Runtime Optimizations, Compiler Optimizations.

*Work done during his internship at Intel.

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

1. INTRODUCTION

Transactional Memory (TM) [10] is an important technology with potential to simplify programming of proliferating multi-core processors. TM avoids or alleviates issues associated with locks, such as deadlocks and poor composition of software modules. Software Transactional Memory (STM) [9] implements TM purely in software. It allows smooth transition from lock-based programming to TM before hardware support becomes available, and it provides backward compatibility and a fallback mechanism [13, 17] once hardware support becomes available. Language and compiler support are necessary to deliver on TM's promise of improved programmability [5], so a full STM system consists of both a compiler and a runtime.

An STM compiler translates programs written using transactional language constructs into executables that run with the STM runtime. The compiler inserts runtime calls for starting and committing transactions, and it converts memory accesses inside a transaction into runtime calls for transactional memory reads and writes (called read and write *barriers*). We refer to the conversion of adding barriers as compiler instrumentation.

In our previous work [19], we identified overheads of transactional barriers as a major performance bottleneck in our STM system. There are two reasons for this: (1) STM barriers require about 10 or more instructions each, much more costly than simple memory load and store instructions. (2) STM barriers can hurt scalability. Write barriers typically acquire locks and may increase the probability of false conflicts. Read barriers either perform expensive lock instructions (for visible readers) or incur costs due to the overheads of read-set validation and privatization-safety (for invisible readers). To make the situation worse, a compiler often adds unnecessary STM barriers, which are pure overhead without any benefit; we call this problem *compiler over-instrumentation* [19].

STM compilers may add unnecessary barriers for three reasons: (1) Static compiler analysis is often imprecise and conservative, and thus cannot remove all unnecessary barriers. Whole-program information is often missing because program modules are dynamically loaded, for example, and it is impossible to perform whole-program compiler analysis. (2) The compiler lacks knowledge about application semantics available to programmers. (3) Lastly, and most interestingly, as we note in this paper, traditional compiler optimizations do not fully exploit transaction semantics, and they miss some opportunities to optimize away unnecessary barriers.

Inside STM barriers, the runtime detects conflicts in order to guarantee atomicity and isolation of transactions. Interestingly, the isolation of a transaction in turn makes some barriers unnecessary. In an in-place update STM system, for example, a write barrier on an address makes barriers on succeeding accesses to that same address in the same transaction unnecessary. Common subexpres-

sion elimination and partial redundancy elimination can optimize a transaction with such accesses [1]. As a more complicated example, a write barrier on a pointer that points to some data allocated inside the current transaction makes barriers on dereferences of the pointer in the same transaction unnecessary. No traditional compiler optimization targets cases like this.

In this paper, we identify *transaction-local memory* as a major source of compiler over-instrumentation. Transaction-local memory is allocated inside a transaction (either on the stack or on the heap). The compiler often generates barriers for accesses to such memory even though these barriers are not needed. Because a transaction is guaranteed to be isolated from other transactions, newly allocated memory cannot escape its allocating transaction. (Or more precisely, it cannot escape the current thread until the allocating transaction commits.) For this reason, we say that transaction-local memory is *captured* by the allocating transaction. We propose optimizations using runtime and compiler techniques to determine whether a read or write barrier accesses captured memory. This allows us to elide certain STM overheads.

The same runtime techniques for eliding barriers for captured memory can also elide barriers for thread-local or read-only accesses to memory. We propose user APIs (which could be transformed into language extensions in a straightforward manner) that programmers can use to annotate certain address ranges as thread-local or read-only. These data annotations allow the runtime to elide STM barriers when accessing memory in the annotated address ranges. We chose manual annotations because automatic optimizations are impossible due to the lack of application-level knowledge by the compiler and the runtime. Previously, this problem was solved using escape actions [19]. Data annotations solve the same problems with cleaner semantics, and therefore make the program more readable and less error-prone. It should be noted that, similarly to escape actions, incorrect use of data annotations can introduce data races.

In the rest of paper, we describe the performance problems associated with compiler instrumentation (Section 2); we present our solutions based on runtime and compiler techniques (Section 3); we then evaluate our compiler and runtime optimizations (Section 4); and finally we present related work and conclude (Sections 5 and 6).

2. BACKGROUND

In this section, we present a brief overview of the Intel C++ STM used in our study, followed by a description of the compiler over-instrumentation problem. (For more information about Intel's STM and STM in general see [14, 1, 16, 11].) We then identify three categories of data, for which an STM compiler may generate unnecessary instrumentation code.

2.1 Software Transactional Memory (STM)

A transaction in Intel C++ STM is represented as an atomic statement, i.e., a statement with a leading `__tm_atomic` keyword. The STM guarantees that transactions are executed atomically and that they are isolated from effects of other concurrent transactions. In other words, single lock atomicity (SLA) semantics is provided—transactions execute as if they were protected by a single program-wide lock. (Note that speculative concurrent execution of transactions is allowed by SLA as long as the illusion of serial execution is preserved.)

When the STM compiler generates code for an atomic block, it instruments the code region with calls to the underlying STM runtime. STM runtime exposes an interface—the application binary interface (ABI)—to the compiler that includes functions for starting and ending transactions and for performing transactional reads and writes of memory locations (*STM barriers*). Internally, the runtime also implements *contention managers* of various policies to resolve conflicts. The STM runtime maps each memory address to a piece of metadata—called *transaction record*—using a hash function. Such mapping is from a cache-line, a word, or a C++ object. Our system is cache-line-based. A system-wide transaction record table is maintained. Each entry of the table tracks ownership of the corresponding memory location. Ownership is usually claimed by locking the transaction record. An STM can be eager-locking or lazy-locking, i.e., locking the transaction record upon memory access or upon transaction commit. An STM can either do in-place updates or write buffering, i.e., directly writing to memory (with eager-locking) or writing to a buffer before flushing it to global memory on commit (with either eager-locking or lazy-locking). Finally, an STM can either be an optimistic-reader or pessimistic-reader, i.e., not locking for reads or locking for reads. Intel C++ STM does optimistic-reading and in-place updates, but may do pessimistic reads in certain cases.

2.2 Compiler Over-Instrumentation

A naive STM compiler translates every memory access inside a transaction into a read or a write barrier. While this does not influence correctness of generated code, it does degrade its performance. STM read/write barriers are expensive and they slow down single-threaded execution because they perform checking, locking, and write buffering or restoring of memory locations. More importantly, excessive use of STM barriers hurts scalability because they access (read or write) shared memory locations (e.g. shared transaction records). In some cases, excessive barriers can introduce false data conflicts—data conflicts that should not exist according to application semantics, but occurs because different addresses are mapped to the same transaction record, either due to a suboptimal hash function or the limited size of the transaction record table. False conflicts degrade system performance by causing unnecessary aborts.

Read/write barriers are necessary only in two cases: 1) Read and write barriers are necessary for memory accesses to locations that are shared and can therefore conflict with other memory accesses;[1] 2) Write barriers (more precisely, undo logs) are necessary for memory accesses to locations that hold live values when the transaction begins (even if the locations are not shared). In other cases, read/write barriers are not necessary and can be omitted by the STM compiler. It is the goal of this paper to identify cases in which STM barriers are not needed and to develop optimizations that would elide read/write barriers in those cases.

2.2.1 Transaction-Local Memory

Memory allocated inside a transaction is not accessible to other transactions and this is why we call it *transaction-local memory*. Both local variables defined inside an atomic block and memory blocks dynamically allocated (using `malloc` or `new`) inside an atomic block are transaction-local. Because transaction-local memory is private, accesses to it cannot cause conflicts with other transactions and they do not require STM barriers. Instead, regular CPU load/store instructions can be used.

Conventional wisdom tells us that if the address A of a memory location M is written to a shared pointer P_{SHARED} , it "escapes" the current thread T (i.e., it becomes shared) because other threads T_i can access it through P_{SHARED} and thus mutual exclu-

[1]Two memory accesses, executed by concurrent transactions, conflict when they access the same memory location and at least one of them is a write operation.

sion is required to access M [4]. However, this is not the case in a transactional system when P_{SHARED} is updated by a transaction t in T using a write barrier. Because transactions are isolated, the new value of P_{SHARED} does not become accessible to any other transaction t_i in thread T_i until current transaction t in thread T commits and, therefore, any newly allocated memory pointed to by P_{SHARED} is also not accessible to any t_i until t commits. In summary, the isolation for pointers indirectly guarantees the isolation for newly allocated memory that they point to. It is interesting to note that, since the isolation is a fundamental property of a transactional system, memory allocated inside transaction t is private to t regardless of any particular STM implementation. For example, this observation holds for both in-place-update and write-buffering STMs regardless of whether they perform eager or lazy locking.

Because transaction-local memory can not escape the transaction that allocates it, we say it is *captured* by the transaction. We use the term *capture analysis* (similar to escape analysis) to refer to a compile- or runtime-time algorithm that determines if a memory location is captured by a transaction or not.

The code snippet in Figure 1(a) (from STAMP benchmark `bayes`) is a typical example of transactions accessing transactional-local stack. In this example, a list iterator `it` is allocated on stack and then used to navigate a shared list inside a transaction.

Memory location M local to a transaction t, might be live-in for its child transaction t' (i.e., t' is nested inside the dynamic scope of t). If that is the case, and if STM supports partial abort of nested transactions (e.g., through user abort in our system), undo logging for M is necessary in the nested transaction t' although it is not needed for the outer transaction. In systems that support nested parallelism—allowing threads to be created inside a transaction, memory locations local to the parent transaction become shared among transactions in child threads and require full STM barriers.

2.2.2 *Thread-Local Memory*

Obviously, no read/write barrier is needed for accessing memory local to a thread, including variables on stack and static or global variables in thread-local storage (TLS). These accesses require undo logging if the memory locations hold live values on transaction begin.

Even if TLS is clearly declared in a program, there are still cases where a compiler can not determine whether a memory access is to thread-local memory unless it performs whole-program analysis. For example, the address of thread-local memory could be passed to an external function through a pointer-type argument. In addition, thread-local memory may escape [4] and stop being thread-local. A compiler that does not perform the whole-program analysis has to make conservative decisions if a thread-local memory location is address-taken and has to generate STM barriers for accesses to it.

What is worth noting is that the memory region can dynamically change from being thread-local to being shared and vice versa. For example, some large piece of data could be split among different threads and processed in parallel, where each thread would work on its own part of the data, and later, after the processing is finished, it could be freely accessed by all threads.

We show a piece of code (from STAMP benchmark `bayes`) that accesses thread-local data in Figure 1(b). The code first allocates a thread-local vector `queryVectorPtr` and subsequently uses it to pass arguments to functions `TMpopulateQueryVectors` and `computeLocalLogLikelihood`. The same vector is used in different transactions and does not get accessed outside the allocating thread.

```
list_iter_t it;
TMLIST_ITER_RESET(&it, taskListPtr);

if(TMLIST_ITER_HASNEXT(&it, taskListPtr)) {
  taskPtr = (learner_task_t*)TMLIST_ITER_NEXT(
  &it, taskListPtr);
  bool_t status = TMLIST_REMOVE(taskListPtr,
  (void*)taskPtr);
}
```

(a) Transaction-Local Stack

```
vector_t* queryVectorPtr = PVECTOR_ALLOC(1);

...

TMpopulateQueryVectors(..., queryVectorPtr, ...);
newBaseLogLikelihood = computeLocalLogLikelihood(
                          ..., queryVectorPtr, ...);
```

(b) Thread-Local Data

```
nodePtr = (list_node_t*)TM_SHARED_READ_P(
prevPtr->nextPtr);

if((listPtr->compare(nodePtr->dataPtr, dataPtr) != 0)
|| (nodePtr == NULL)) {
  return NULL;
}

return (nodePtr->dataPtr);
```

(c) Read-Only Data

Figure 1: Examples of where certain memory accesses do not require STM barreris. Each code snippet above is part of a transaction, the beginning and ending of which is not shown here.

2.2.3 *Read-Only Memory*

Accesses to read-only memory do not require read barriers, even if the memory is shared among threads. As with thread-local data, a memory region can dynamically change between being read-only and read-write. For example, some data may be updated at the beginning of a program during initialization, but never changed afterwards. Declaring read-only variables constant using C/C++ keyword `const` does not fully solve the problem of over-instrumentation for read-only data, for reasons similar to thread-local data. Furthermore, the `const` qualifier could simply be cast away when the data is accessed.

If not for these cases, a `const` qualifier could be used for global variables to inform the STM compiler that the variable is read-only and that barriers are not required when accessing it. However, it is hard for the compiler to tell if a pointer really points to some read-only memory. For example, if the address of a `const` global is passed to a function as a pointer argument, it is impossible for the compiler compiling that function to know that the pointer points to read-only memory, unless a complicated whole-program analysis is performed. In C++, defining an argument `p` to function `foo` as `const <type> *p` only guarantees that function `foo` does not change the memory pointed to by `p`, but not that `p` points to immutable data. And there is no way to express that in C++.

A function in Figure 1(c) (from STAMP implementation of a linked list) searches for a particular element in the linked list. It can access both `listPtr->compare` and `nodePtr->dataPtr` directly as both of these fields do not change after the data structure is created.

```
type_t read_barrier(transaction_t *td, type_t *address) {
    if(is_captured(td, address))
        return *address;

    return full_read_barrier(td, address);
}
```

Figure 2: Pseudo-code for read barrier that uses runtime capture analysis

3. RUNTIME AND COMPILER OPTIMIZATIONS

In this section we describe runtime and compile-time techniques that determine whether memory locations being accessed are captured (transaction-local) or not. Runtime capture analysis techniques are described in Section 3.1. We describe how the same techniques used for runtime capture analysis can be used to elide barriers for accesses to thread-local and read-only memory. We describe several implementations of runtime optimizations and discuss their respective tradeoffs. Compiler capture analysis is presented in Section 3.2.

3.1 Runtime Optimizations

Runtime capture analysis cannot fully elide barriers—instead it aims to reduce their cost. Figure 2 shows the pseudo-code for the read barrier that performs runtime capture analysis. It first checks whether the data being accessed is captured by the current transaction. If so, the data is read from memory directly and returned. In this way, costs of logging, locking and maintaining read and write sets are eliminated for unnecessary barriers. If the memory locations are not captured, the standard read barrier algorithm is invoked and the resulting value is returned. The pseudo-code for write barriers is similar and is omitted.

If runtime capture analysis is faster than a regular STM barrier and it succeeds often enough to elide a sufficient number of barriers, the average cost of a barrier in an application will be reduced. How often the regular STM barrier can be elided (i.e., how often the accessed memory location is captured) mainly depends on the application itself. The main implementation problem to solve is how to efficiently implement the runtime capture analysis (the `is_captured()` function in Figure 2).

Transaction-local memory can be either a part of the stack or the heap. Due to the different natures of these two allocation mechanisms, runtime capture analysis also differs.

3.1.1 Capture Analysis for Transaction-Local Stack

A stack is a very simple and fast last-in-first-out allocation mechanism. The transaction-local stack (Figure 3) is a contiguous range of memory locations between (1) the top of the stack at the beginning of the transaction and (2) the current top of the stack. The transaction descriptor contains a pointer to the beginning of transaction's stack and the CPU stack pointer register points to the current stack top. Runtime capture analysis for transaction-local stack is relatively simple and fast—it requires a single range check (Figure 4.)

3.1.2 Capture Analysis for Transaction-Local Heap

Allocating data on the heap is much more flexible than allocating it on the stack, as memory blocks can be allocated and deallocated in an arbitrary order. This, however, introduces additional complexity to memory allocation/deallocation algorithms [12, 2, 11]. As the data allocated on the heap does not occupy a contiguous range of memory locations, our runtime capture analysis for transaction-local heap is more complex than the one used for the transaction-local stack.

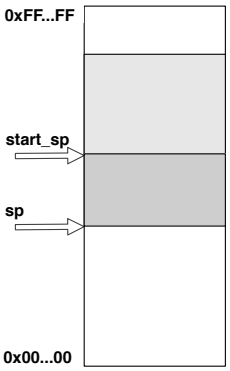


Figure 3: A transaction-local stack (stack grows downwards). Shaded memory represents stack used by a single thread. Darker shaded memory is transaction-local stack. Its beginning is pointed to by a field from transaction descriptor and its end by processor stack pointer register.

```
int is_captured_on_stack(void *addr) {
    return (start_sp<addr && addr <= sp);
}
```

Figure 4: Pseudo-code for runtime checking for transaction-local data on stack

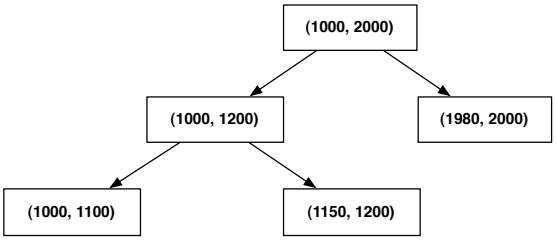


Figure 5: A search tree that logs memory blocks allocated inside a transaction, populated with three memory ranges: (1000,1100), (1150,1200) and 1980,2000).

In order to perform capture analysis for transaction-local heap, all transactional allocations are logged in a transaction-local allocation log. We extended the existing transactional memory allocator in our STM runtime to keep a log of all memory blocks allocated in a transaction. The runtime searches the allocation log for the address being accessed. If the address belongs to the log, it means that memory starting at the address was allocated (and is captured) by the current transaction.

Efficiency of the data structure used to implement allocation log is crucial for fast transaction-local heap capture analysis. We implemented the allocation log using three different data structures: a search tree, an array, and a filter of memory ranges.

Search tree.

The search tree (Figure 5) allows insertions and removals of memory ranges and search operations to determine if a data item (described as the starting address and the data size) belongs to a

memory range stored in the tree. The tree has the following structure:

- Each leaf represents the range of an allocated memory block.
- Each internal node represents a range with the lower bound as the minimum of those of its child nodes, and the upper bound as the maximum.

This way, searches that result in misses are relatively fast, because they usually terminate at an internal node at higher levels of the tree. This satisfies the design principle of optimizing for common cases, which aims at lowering the performance cost suffered by barriers that do not benefit from runtime capture analysis.

Array.

The array implementation of the log simply keeps all memory ranges allocated inside a transaction as an unsorted array. The array is of cache-line size, so that logged memory ranges can be brought into cache all at once. On a 32-bit CPU that has 64-byte cache-lines the array has the layout depicted in Figure 6. In the figure, $start_i$ is the starting address of a memory range, and end_i is its ending address. The runtime capture analysis simply traverses the array and checks whether the address of the memory location belongs to any of the ranges stored in the array. If the accessed memory falls into any of those ranges, the regular barrier can be elided. Otherwise, the full STM barrier has to be used.

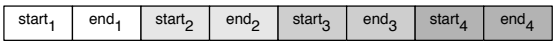


Figure 6: An array of memory ranges implementation of allocation log on a 32-bit CPU.

Our array-based capture analysis takes advantage of the fact that capture analysis does not have to be accurate as long as it is conservative. Inaccurate capture analysis misses some opportunities for barrier elision and is, therefore, less effective, but it does not cause any correctness issues. While capture analysis can be arbitrarily inaccurate for direct update STMs (such as our STM) it must provide consistent results for deferred update STMs (such as e.g. [6]). (In a deferred-update (or write-buffering) STM, if an STM barrier is used to buffer writes to a location, then STM barriers have to be used to read that same location.) We also assume that most transactions do not perform many memory allocations, so the limited size of our array does not pose a problem. Results presented in Section 4 show that this approximation is good enough as the performance results of array-based capture analysis compare well to the results obtained using tree-based technique.

Filtering.

Our third implementation uses a hash table as a filter, a approach similar to the runtime filtering technique described in [8], but extended to handle memory ranges larger than a single data item. In this scheme, the allocation log is implemented as a hash table. When a block of memory gets allocated, all memory locations belonging to the block are hashed and the corresponding hash table entries are marked with the exact addresses of the corresponding memory locations (unless they have already been marked by a previous allocation in the same transaction). Upon deallocation, all hash table entries belonging to the being-deallocated memory block are cleared. To perform capture analysis, the system hashes the address of the accessed memory location and checks whether the corresponding hash table entry contains it. Our filtering scheme allows false negatives, but is conservative (similarly to the array implementation) in the sense that it never generates falso positives.

The check on memory accesses using the filter is fast, as it requires only a hash and a compare. However, allocation and deallocation is more expensive especially with large allocation sizes.

```
void addPrivateMemoryBlock(void *addr, size_t size);
void removePrivateMemoryBlock(void *addr, size_t size);
```

Figure 7: APIs to annotate data safe for direct accesses

Simplifying Heap Checks.

It might look as if it would be possible to simplify and speed up the heap checks by structuring the heap similar to the stack with a special memory allocator. For example, every thread could have its own memory pool for transaction-local heap allocation, and use an allocation algorithm similar to a stack, in which a pointer to available memory monotonically increases and allocated memory stays in a continuous range. Capture analysis would be simple and fast as using a stack. Unfortunately, this approach is not going to work well, as it may lead to severe memory fragmentation after memory is freed inside a transaction, and requires extra copy out operations from the thread-local memory pool to shared memory when the tranaction commits and allocated data becomes shared.

3.1.3 *Capture Analysis for Thread-Local and Read-Only Memory*

It is impossible to decide whether certain data is thread-local or read-only at runtime without help from the programmer, as this relies on the application-level knowledge. Instead of trying to automatically detect which locations are thread-local or read-only, we expose new API calls to the programmer (Figure 7). Using these API calls, the programmer can annotate memory regions to be (or stop being) safe for accessing without STM barriers. When a memory block becomes private (via a call to `addPrivateMemoryBlock()`), it is inserted into the log tracking thread-local data, which uses the same data structure and general algorithms as the allocation log. The main difference between these two logs is that allocation log gets emptied on every transaction end (commit or abort), while the thread-local data log does not. This is why we did not reuse the allocation log to track thread-local data. When a memory block becomes shared again (via a call to `removePrivateMemoryBlock()`), it is removed from the log. The read/write barrier code (depicted in Figure 2) stays the same as before.

An alternative approach that would use dedicated memory pools for read-only and thread-local data to speed up the runtime checks would suffer from similar issues as the memory allocation based scheme for the transaction-local heap. In particular, it could not efficiently cope with data that is read-only (thread-local) in some parts of the program's lifetime and read-write (shared) in others, because it would require moving data between different pools.

3.2 Compiler Optimization

While runtime optimizations may reduce average cost of barriers by avoiding full barriers (see Section 4), additional runtime checks introduce new costs. In some cases, these additional costs could even lead to performance degradation if the cost of runtime capture analysis outweighs the potential savings from barrier elision. An alternative to performing capture analysis at runtime is to do it at compile-time, and completely elide (some of) those unnecessary STM barriers, without paying additional runtime costs.

We implemented our compiler capture analysis using pointer analysis, which determines whether a pointer points to memory allocated inside the current transaction. If it does, dereferences of the pointer do not require STM barriers. Similarly to runtime techniques, the compiler capture analysis can tolerate false negatives, and thus can use pointer analysis that is not completely accurate, as long as it is conservative.

Our compiler capture analysis is based on the standard pointer analysis implemented in the Intel C++ compiler. It only uses intra-procedural pointer analysis, and relies on function inlining to extend the analysis results across function calls. We opted for not using inter-procedural pointer analysis (which is also supported by Intel C++ compiler) as it would increase compilation time and would not be fully applicable when dynamically linked libraries are used. Experimental measurements from Section 4 show that, even though we use a simple compiler technique, the capture analysis is still quite effective.

Instead of detecting data that is safe to access directly, the compiler could detect which accesses are to shared data. When accessing shared data, STM barriers have to be used, but runtime filtering need not be used in these cases. This would improve the speed of runtime techniques, by avoiding runtime checks when the compiler can detect that they are not beneficial. We have not explored this approach and left it as future work.

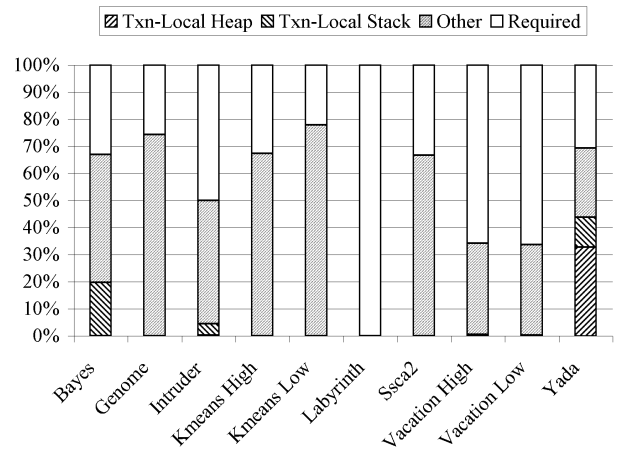
4. EVALUATION

We use the STAMP 0.9.9 [3] benchmark suite to evaluate our runtime and compiler optimizations. All the measurements were performed on a Dunnington machine with four Intel processors—six cores each at 2.66GHz—and 16GB of RAM running Red Hat Linux version 7. However, since STAMP only allows executions with a power-of-two number of threads, we were limited to using at most 16 threads in our experiments.

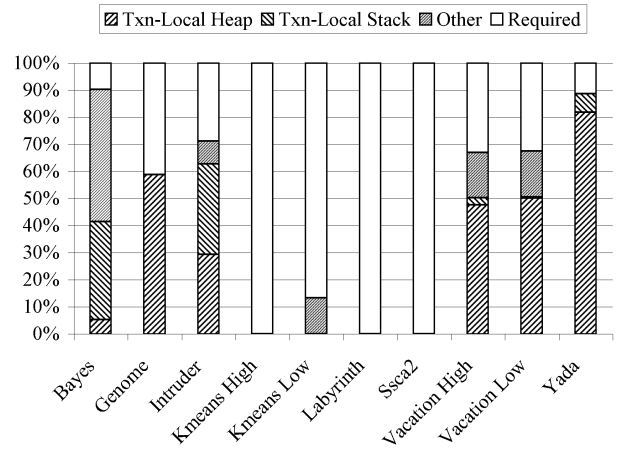
4.1 Barrier Elision Opportunities

In order to estimate opportunities for optimization, we ran the benchmarks at a single thread and counted the numbers of STM barriers that are (1) accessing transaction-local heap, (2) accessing transaction-local stack, (3) not required for other reasons, and (4) required. To estimate the number of required barriers, we counted manually instrumented accesses in the original STAMP benchmark programs. Although this might not be the absolutely minimal number of barriers that are required for correct code—as the manual instrumentation may have added unnecessary barriers too, it is a good basis to estimate of the number of barriers that the STM compiler added. We used our tree-based runtime algorithm to count the total number of transaction-local heap and stack accesses. Because the tree-based runtime algorithm is precise, it detects all accesses to transaction-local heap and stack. The rest of the STM barriers generated by the compiler are not required, but are not transaction-local either. They might be, for example, accesses to read-only or thread-local data. These barriers cannot be elided without the help of the programmer and we did not elide those barriers in the following experiments.

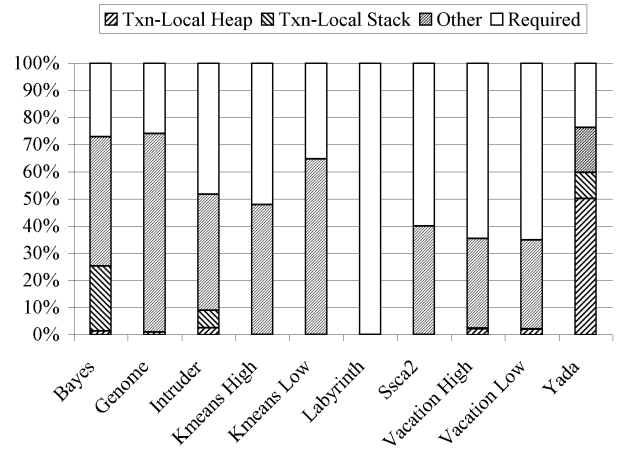
Figure 8 depicts a breakdown of compiler-inserted STM barriers for all STAMP programs, for reads, writes, and all memory accesses (reads and writes combined). The numbers in Figure 8 suggest significant opportunities with most STAMP programs to improve the performance of our STM by eliding barriers. The only program where we did not see any redundant barrier is `labyrinth`. Also, the number of barriers that can be elided automatically (accesses to transaction-local stack and heap) is much higher for write than for read barriers. This is encouraging as write barriers are more expensive than read barriers. Finally, the figure conveys that the number of reads is typically much higher than the



(a) Read breakdown



(b) Write breakdown



(c) Breakdown of all accesses

Figure 8: Memory access breakdown

(a) Read barriers

(b) Write barriers

Figure 9: Portion of barriers removed by different optimization techniques

number of writes for all benchmarks (except for `yada`) and that the overall number of barriers that can be elided automatically can be pretty high (e.g. 60% for `yada`).

Figure 9 shows the number of barriers that different capture analysis techniques removed. The figure conveys two interesting points: (1) the only program for which array runtime implementation removes fewer barriers than the others is `yada`, which suggests that almost the full potential of capture analysis can be achieved by tracking only a few memory allocations; (2) despite using a simple algorithm, the compiler analysis is quite effective identifying and removing unnecessary barriers—the only case for which it removed significantly fewer barriers than the runtime techniques using search tree and filtering hashtable is `yada`, while still removed more than the filtering technique. It is worth noting that, for all programs, the compiler optimization removed at least as many barriers as the runtime optimization using an array.

4.2 Performance Impact

Table 1 shows the abort-to-commit ratio for all benchmarks running at 16 threads, for all optimizations and the baseline (to which no optimization is applied). This ratio is defined as the ratio of number of transactions aborted and retried to the number of transactions committed. The table reveals that our optimizations significantly reduce the number of aborts for both high- and low-contention configurations of `vacation` benchmark. This results in a performance improvement, as we show next.

To evaluate the performance improvement, we first measured the execution time of all benchmarks running at a single thread, and compared the results for all runtime and compiler optimizations to the baseline. In experiments for runtime optimizations, we used the tree data structure for allocation log and performed experiments in three different configurations: (1) checking for both transaction-local stack and heap accesses in both read and write barriers, (2) checking for both transaction-local stack and heap accesses only in write barriers, and (3) checking for only transaction-local heap accesses only in write barriers. The results presented in Figure 10 show that runtime optimizations do not degrade single-thread performance, except for `kmeans` and `yada`. For `kmeans`, this was due to the lack of barrier elision opportunities (Figure 8). Since there was no barrier elision opportunity, runtime checks for transaction-local accesses only added overheads, without benefits. Because the number of barriers is very high for `kmeans`, the overhead was significant. For `yada`, most of the redundant barriers were already caught by cheap write-after-write checks in the baseline system. Because of this, our runtime optimizations did not improve performance, but added overheads and degraded performance. It is worth noting that when we disabled runtime optimizations for read and transaction-local stack accesses, the overhead of our runtime checks was reduced significantly. The compiler optimization, on the other hand, did not cause any performance degradation, as expected, because it did not introduce any runtime overheads.

	Baseline	Tree	Array	Filtering	Compiler
Bayes	0.95	0.86	1.2	0.94	0.67
Genome	0.05	0	0	0	0
Intruder	0.78	0.79	0.78	0.78	0.78
Kmeans High	1.6	1.6	1.6	1.6	1.6
Kmeans Low	0.66	0.67	0.68	0.65	0.67
Labyrinth	0.18	0.17	0.17	0.17	0.17
Ssca2	0	0	0	0	0
Vacation High	0.28	0.01	0.01	0.01	0.02
Vacation Low	0.24	0	0	0.01	0.01
Yada	1.7	1.6	1.7	1.6	1.6

Table 1: Abort-to-commit ratio at 16 threads

We repeated the same experiments at 16 threads. The performance for some of the benchmarks showed great fluctuations at 16 threads despite us repeating the experiments multiple times for each data point. The fluctuations were mostly caused by contention and the simple exponential-back-off contention manager that we used. The relative standard deviation for all benchmarks at 16 threads for 5 repeated runs is shown in Table 2. Figure 11(a) shows the performance of `vacation` improved by 14% for the high-contention configuration and 18% for the low-contention configuration. The reasons for these improvements are: (1) a relatively high number of barrier elisions (Figure 9) and (2) the reduction of false conflicts (Table 1). The figure also shows that our compiler optimization, although very simple, performed close to or better than runtime op-

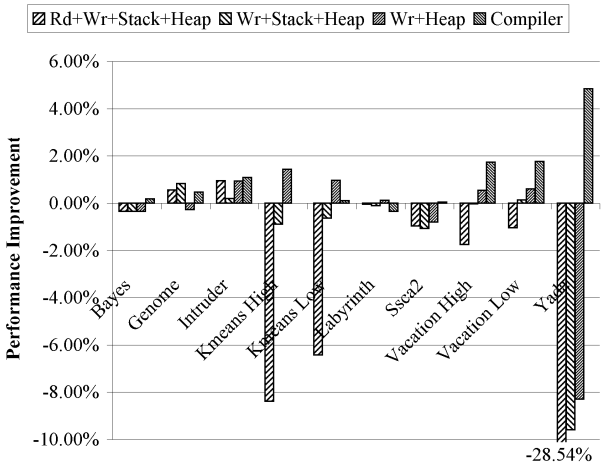


Figure 10: Performance improvement from runtime and compiler optimizations at single thread. Different configurations of runtime were tried and measured.

	Baseline	Tree	Array	Filtering	Compiler
Bayes	35	29	10	31	15
Genome	4.8	3.0	1.8	1.1	1.5
Intruder	1.5	0.95	1.3	0.98	1.3
Kmeans High	8.5	10	8.4	9.2	7.0
Kmeans Low	28	23	27	13	19
Labyrinth	5.7	18	15	11	12
Ssca2	1.8	1.1	0.96	0.36	1.4
Vacation High	1.0	0.33	0.89	0.51	0.78
Vacation Low	1.2	0.88	1.9	0.69	1.4
Yada	2.2	1	1.9	0.43	2.9

Table 2: Percent relative standard deviation at 16 threads(%)

timizations in almost all cases. However, other benchmarks were indifferent to the optimizations, given the variance we saw in our measurements.

Figure 11(b) compares three different implementations of the runtime optimization and the compiler optimization. In these experiments, we ran runtime checks only in write barriers and only for transaction-local heap accesses. The figure shows that, while all three runtime techniques resulted in comparable performance improvements, `tree` and `array` implementations performed slightly better than the `filter`, due to their lower costs.

5. RELATED WORK

Improving STM performance by reducing cost of read/write barriers or eliminating unnecessary barriers completely has been previously done in contexts of both managed and unmanaged languages.

For unmanaged languages (C/C++), data partitioning is used in [15] to fine-tune the STM algorithm for different parts of program data, based on runtime workload characteristics. Special types of data partitions are thread-local and transaction-local, which correspond to thread-local and transaction-local heap in our system. These partition types also use the same optimization techniques. However, the approach in [15] relies on compiler data structure analysis (DSA) and does not account for transactional stack accesses. In [14], the burden of eliminating unnecessary barriers is placed on the programmer who can annotate certain functions as `tm_pure`, thus avoiding all instrumentation inside these functions. Also, [14] introduces STM barrier optimizations that allow the underlying STM runtime to invoke specialized versions of STM barriers for memory locations that have already been accessed by the same transaction. Some STM barriers can effectively be eliminated this way (read-after-write access for STMs that use update-in-place, for example). While barrier optimizations, similarly to optimizations proposed in this paper, aim to reduce STM barrier costs, these two techniques are orthogonal.

Performing compiler analysis that eliminates unnecessary STM barriers (or object opening for read or write) in managed languages, like Java, is simpler and can be more precise, due to restrictions that these languages impose on data pointers (references). In [1], authors describe both optimizations similar to barrier optimizations in [14] and elimination of transactional operations for transaction-local objects in context of a Java system. Dataflow analysis of Java programs is used in [7] to determine when the transactional accesses can safely be replaced with non-transactional ones. The system described in [8] uses flow-sensitive inter-procedural compiler analysis and runtime log filtering to identify accesses to transaction-local heap objects. This analysis is used to eliminate undo-logging of accessed fields, thus improving performance. Dynamic escape analysis and static not-accessed-in-transaction analysis are used in [18] to eliminate non-transactional barriers used to enforce isolation and consistent ordering between transactional and non-transactional accesses in a Java STM system.

6. CONCLUSIONS AND FUTURE WORK

We identified captured memory (transaction-local memory) as a major source of compiler over-instrumentation, one of the performance bottlenecks identified in our previous work [19]. We proposed and implemented runtime and compiler optimizations to elide STM barriers for captured memory. Our experimental results showed such optimizations achieved as high as 18% performance improvement for programs in the STAMP 0.9.9 benchmark suite at 16 threads. This suggested that while analyzing and optimizing performance of an STM system, we should look beyond traditional compiler and runtime optimizations, take into consideration the nature of a transactional system, and take advantage of the special properties of transactions in order to optimize their performance.

Directions for future work include designing and implementing compiler analysis to identify memory accesses that definitely require STM barriers and avoid runtime checks trying to elide them. Also in general, understanding and optimizing STM performance requires better understanding of contention and improving contention management.

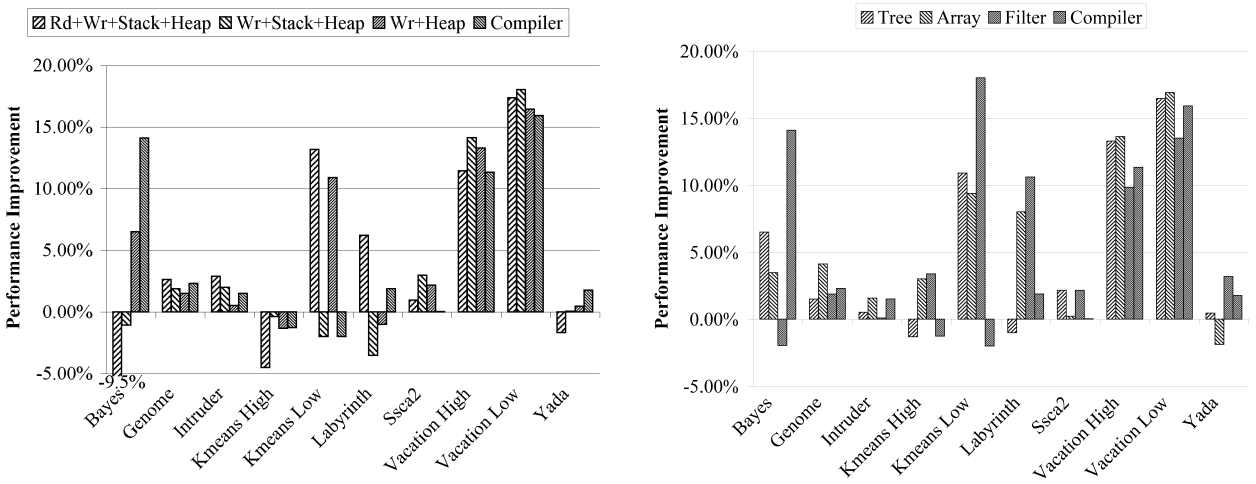
Acknowledgments

We are grateful to Xinmin Tian, Ravi Narayanaswamy, Sergey Kozhukow, and Serguei Preis for their help with implementation of the compiler optimization and to anonymous reviewers for their helpful comments.

7. REFERENCES

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay S. Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, 2000.



(a) Using different configurations of runtime (using tree of memory ranges for heap checks) and compiler optimizations

(b) Using different data structures for runtime optimizations and compiler optimization

Figure 11: Performance improvement from runtime and compiler optimizations at 16 threads

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[4] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *OOPSLA 1999*.

[5] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *TRANSACT 2007*.

[6] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC 2006*.

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

[8] Timothy Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI 2006*.

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

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

[11] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *ISMM 2006*.

[12] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley Professional, July 1997.

[13] Yosef Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT 2007*.

[14] 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 ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 195–212, New York, NY, USA, 2008. ACM.

[15] Torvald Riegel, Christof Fetzer, and Pascal Felber. Automatic data partitioning in software transactional memories. In *20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.

[16] Bratin Saha, Ali-Reza Adl-Tabatabai, Rick Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006*.

[17] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 2006*.

[18] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Saha Bratin. Enforcing isolation and ordering in STM. In *PLDI 2007*.

[19] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *SPAA 2008*.