Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware

Martín Abadi[⋆†] Tim Harris[⋆] Mojtaba Mehrara[⋆‡]

Microsoft Research[⋆] University of California, Santa Cruz[†] University of Michigan[‡]

abadi@microsoft.com tharris@microsoft.com mehrara@eecs.umich.edu

Abstract

This paper introduces a new way to provide strong atomicity in an implementation of transactional memory. Strong atomicity lets us offer clear semantics to programs, even if they access the same locations inside and outside transactions. It also avoids differences between hardware-implemented transactions and software-implemented ones. Our approach is to use off-the-shelf page-level memory protection hardware to detect conflicts between normal memory accesses and transactional ones. This page-level mechanism ensures correctness but gives poor performance because of the costs of manipulating memory protection settings and receiving notifications of access violations. However, in practice, we show how a combination of careful object placement and dynamic code update allows us to eliminate almost all of the protection changes. Existing implementations of strong atomicity in software rely on detecting conflicts by conservatively treating some non-transactional accesses as short transactions. In contrast, our page-level mechanism lets us be less conservative about how non-transactional accesses are treated; we avoid changes to non-transactional code until a possible conflict is detected dynamically, and we can respond to phase changes where a given instruction sometimes generates conflicts and sometimes does not. We evaluate our implementation with C# versions of many of the STAMP benchmarks, and show how it performs within 25% of an implementation with weak atomicity on all the benchmarks we have studied. It avoids pathological cases in which other implementations of strong atomicity perform poorly.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Algorithms, Languages, Performance

1. Introduction

Many implementations of `atomic` blocks in programming languages provide "weak atomicity" which means that there is no concurrency control between the operations being done inside an `atomic` block and the operations being done outside [3]. It is often difficult to define the semantics of such systems independently of the details of a particular implementation, so programs that seem to work on some implementations may not work on others.

The "privatization problem" [6, 7, 26, 28] illustrates several of the questions that arise in implementations with weak atomicity:

```
// Initially x==0, x_shared==true

// Thread 1              // Thread 2
atomic {                 atomic {
  x_shared = false;        if (x_shared) {
}                            x ++;
x = 100;                 } }
```

A programmer might reason that Thread 1's update to `x_shared` allows its subsequent update to `x` to be made as a normal non-transactional store. After these fragments have run, a programmer might expect that `x==100` whichever order the `atomic` blocks ran in. However, implementations over software transactional memory (STM [24]) can lead to other results, e.g., `x==1` if the implementation of Thread 2's `atomic` block was still writing back a buffered update to `x` concurrently with Thread 1's non-transactional store.

In this paper we examine the implementation of `atomic` blocks that provide "strong atomicity", that is, with concurrency control between transactional and non-transactional accesses to the same memory location. With strong atomicity `x==100` is the only possible result in our example. Furthermore, since strong atomicity is the semantics offered by hardware TM implementations, it is attractive to be able to provide the same semantics in software.

Existing software implementations of strong atomicity are based on modifying code outside `atomic` blocks to detect conflicts with concurrent transactions. These accesses are conservatively expanded to optimized short transactions. Naïve implementations perform poorly because of this expansion, although whole-program analyses can be used to reduce the number of memory accesses that must be expanded in this way [25].

In this paper we examine an alternative approach that lets us modify code dynamically in response to a possible conflict. When compared with conservative expansion, this dynamic approach can reduce the number of non-transactional operations that have been expanded but do not actually encounter conflicts during execution.

Our key implementation technique is to use off-the-shelf memory protection hardware to detect possible conflicts between transactional accesses and normal accesses. We detect conflicts by organizing the process's virtual address space so that its heap is mapped twice; one mapping is used when executing inside a transaction, and the other mapping is used during normal execution. This organization lets us selectively prevent normal access to pages while they remain accessible transactionally. We use this mechanism to detect possible conflicts between transactional accesses and normal accesses at the granularity of pages; we use an existing STM to detect conflicts between transactions at the granularity of objects.

We introduce our design in several stages. We discuss the basic approach in Section 3, showing how we organize the process's virtual address space, and how we wrap the STM implementation

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.
Copyright © 2009 ACM 978-1-60558-397-6/09/02. . . $5.00

with page-level concurrency control. We discuss how we handle access violations (AVs) triggered by normal accesses. We sketch an informal correctness argument.

This basic design provides a foundation that is sound but slow: on conventional hardware it is costly to modify the memory protection settings, and to handle any AVs that arise. In Section 4 we introduce a number of techniques to mitigate these costs. We allow the language runtime system to operate without triggering AVs. We use static analysis to identify non-transactional operations that are guaranteed not to conflict with transactions (these can bypass the page-level checks) and to identify transactions that are guaranteed not to conflict with normal accesses (these transactions need not revoke normal-access permissions on the pages involved).

As we discuss in Section 5, we dynamically identify operations that trigger large numbers of AVs and update them to use short transactions. These updates limit the number of times that any given instruction can trigger an AV, letting us avoid the resulting costs in programs that include sharing between transactional accesses and normal ones (whether actual sharing of the same object, or false sharing of different objects on the same page).

We present low-level implementation details in Section 6. We show how we use dynamic code updating to reduce the D-TLB pressure introduced by mapping a process's heap twice. We also explain how we reduce false sharing when transactionally accessed and non-transactionally accessed data is placed on the same page. All of our work has been implemented in user mode over Windows Vista and Windows Server 2003. We did not make any modifications to the operating systems.

We examine the performance of our implementation in Section 7. We show how, despite its complexity, it does not harm the scaling of our benchmarks. We also show that the overhead of using strong atomicity is less than 25% over weak atomicity, even when we do not use any whole-program analyses to reduce conflicts.

We discuss related work in Section 8 and conclude in Section 9.

2. Programming Model

We base our work on the Bartok compiler and runtime system in which we have developed earlier STM implementations [10]. In this section we summarize this existing work and the programming model that we are implementing.

Our starting point is the C# language augmented with block-structured `atomic` sections. These `atomic` blocks are intended only for concurrency control on shared memory data structures; it is an error to invoke native code from them. Code within an `atomic` block must be written in the safe verifiable subset of C#; unsafe constructs, such as pointer arithmetic, are not permitted. The programmer does not need to indicate whether or not a particular piece of data is accessed in `atomic` blocks, or whether or not a particular method may be called from them.

2.1 STM Implementation

The `atomic` blocks are implemented using STM following the approach of Harris and Fraser [9]. STM operations are introduced by the compiler for concurrency control on each of the objects accessed inside an `atomic` block and then static analyses are used to identify operations that are redundant (for example because a transaction re-reads an object it has already read).

Bartok-STM makes in-place updates as a transaction runs; this technique is also known as "eager updates" or "eager versioning" [18]. The STM uses encounter-time locking between writers, so that at most one transaction may be writing to any given object. The STM uses lazy conflict detection based on per-object version numbers for reads. The compiler adds periodic validation checks inside transactions. Our earlier paper provides pseudo-code for these operations [10]. The techniques in this paper could readily apply to other STM designs.

The runtime system implements services like memory allocation, garbage collection (GC), and the STM all in C#. Ordinary type-safe C# is used for the majority of these services, with unsafe loop-holes used in the low levels. A similar approach is taken in the Jikes RVM implementation of the Java Virtual Machine.

The STM implementation is integrated with the memory allocator and GC. Different threads can concurrently allocate storage space without introducing conflicts between their transactions. GC can occur during a transaction without requiring the transaction to be aborted, and the GC can reclaim objects that a transaction has allocated but which will be unreachable whether or not it commits.

We use Bartok in an ahead-of-time whole-program mode, generating native x86 code from the bytecode of the application compiled along with the bytecode of the runtime system.

2.2 Memory Management Requirements

In this section we summarize the requirements that we make of the operating system's memory management APIs and the processor's memory management hardware.

- We require a mechanism to allocate storage space and to map the same storage space at multiple locations in a process's virtual address space. In Win32 this mechanism is provided by `CreateFileMapping` and `MapViewOfFile`. In Unix it is provided by `mmap`.
- We assume that a process has a mechanism to control the access permissions on the pages in its virtual address space. In Win32 this is provided by `VirtualProtect`. In Unix it is provided by `mprotect`. We assume that a change to a page's protection settings takes effect before a call to `VirtualProtect` or `mprotect` returns. To achieve this, the implementation of these calls may require an inter-processor "TLB-shootdown" interrupt to be used to flush any other processors' TLB mappings for the page.
- We assume that the threads within a process see a coherent view of multiple mappings of the same data: an update through one virtual address for a given piece of data should be immediately reflected through any other virtual addresses for the same data.
- We require a mechanism for a process to receive notifications of AVs that it triggers—e.g., via structured exception handling in C++ on Win32, or via a signal handler on Unix.

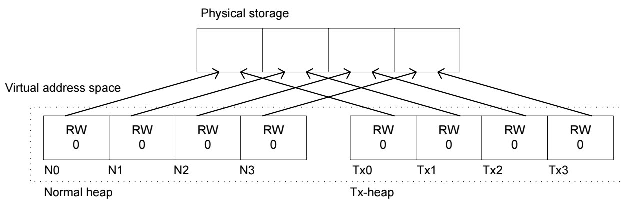
All of these requirements are conventional and met by current desktop and server operating systems and off-the-shelf processors.

3. Strong Atomicity via Page Protection

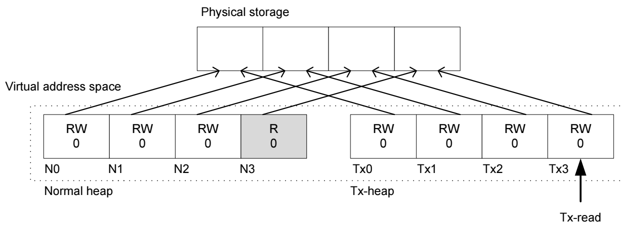
In this section we introduce our basic approach for building strong atomicity over off-the-shelf memory protection hardware. This provides a sound—but slow—foundation on which to develop a practical implementation in Sections 4–6. We introduce our basic design (Section 3.1). We discuss a number of design alternatives and the rationale for the choices we made (Section 3.2). We sketch an informal correctness argument (Section 3.3).

3.1 Implementation Overview

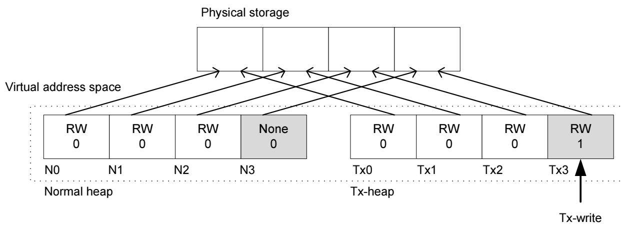
Figure 1(a) shows the heap structure that we use. The process's virtual address space initially contains two identical read-write views of the same heap; we call these the "normal heap" (pages N0..N3 in the figure) and the "tx-heap" (pages Tx0..Tx3). The normal heap is used during non-transactional execution. The tx-heap is used inside transactions. Page protection settings are associated with pages in



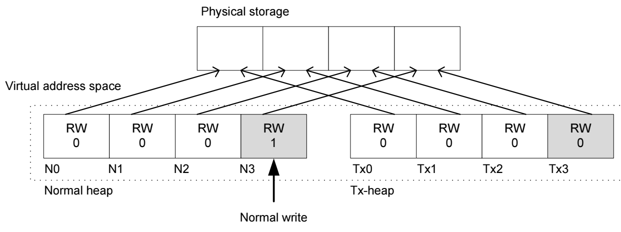
(a) Initial heap structure.



(b) Transactional reads disable normal writes to the same page.



(c) Transactional writes increment the writer count and disable normal access to the same page.



(d) An AV from a normal write must wait for any writing transactions to finish. It then increments the transition count and restores normal access to the page.

Figure 1. Heap structure during a transaction and in response to an AV from normal code. Changes are shaded at each step.

the virtual address space, and so, for example, Tx0 may have different protection settings than N0. In order to detect conflicts between transactional and non-transactional accesses, we modify page protections so that a possible conflict triggers an AV.

For simplicity we show both heaps referring to contiguous physical memory. In practice the physical storage might not be contiguous and might also be paged out to disk in the usual way. We place the two heaps at a constant offset from one another. This makes it easy to translate between addresses. On a 32-bit machine we use a 1GB offset. A larger offset could be used on 64-bit machines. We use normal 4KB pages.

At runtime we maintain three meta-data values for each physical page. First, a status field indicates which kinds of non-transactional access are currently permitted ("None", "R", or "RW"). This field lets the runtime system check a page's current status without needing to make a system call. Second, a "writer count" records how many transactions are currently writing to the page. This is shown on each tx-heap page in the figure, with its initial value of zero. The writer count is used when handling an AV: the AV handler waits until the writer count is back to zero before restoring normal access to the page. Finally, a "transition count" records how many times the page has had normal read-write access restored to it. This is shown on each normal-heap page in the figure, initialized to zero. As we illustrate below, the transition count is used to detect page-level conflicts between transactional readers and non-transactional writers: a change to the transition count indicates, to a transactional reader, that the page has reverted to normal access at least once during the transaction (so the transaction must abort because of the possible conflict).

Figure 1(b)-(d) illustrate our page-based technique using the privatization example of Section 1. Suppose that Thread 1 is just about to execute `x=100` non-transactionally, and Thread 2 is just about to execute `x++` inside a transaction. With weak atomicity this conflict on `x` is not detected and various implementation-dependent results are possible (e.g., `x==101`).

Our implementation of strong atomicity can detect the conflict as follows:

1. Figure 1(b): Thread 2 requests transactional read access to page Tx3 that holds variable `x`. This changes the protection on N3 to be read-only, preventing any normal updates being made to that page. The transaction records the transition count for page N3 (currently 0). The transaction proceeds with its read once the page protection has been changed.
2. Figure 1(c): Thread 2 requests transactional write access to page Tx3. This has two effects: the writer count is incremented on Tx3, and all normal access to the page is revoked.
3. Figure 1(d): Thread 1 attempts `x=100`, which triggers an access violation on page N3. The AV handler must wait until there are no concurrent transactional writers: in this case it must wait for Thread 2's transaction to roll back. When there are no transactional writers, Thread 1 increments N3's transition count (to cause any concurrent transactional readers to be aborted), and restores normal read-write access to the page. An AV triggered by a read is similar, except that (i) read-only access can be granted instead of read-write access, and (ii) the transition count need not be incremented because the normal read will not conflict with a transactional read.

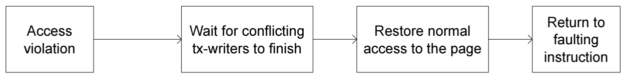
3.2 Design Choices

There are many variants of this general technique. The particular design choices in our implementation are motivated by two goals. Our first goal is to avoid overhead on non-transactional code unless a page-level conflict is detected; we assume that execution is usually non-transactional, and so we want to avoid slowing it down. Our second goal is to reduce the overhead that supporting strong atomicity (rather than weak atomicity) imposes to the STM implementation.

We identified a number of design choices, and used our goals to help select between them:

When to revoke access. We considered whether to revoke transactional access to pages that are being used outside transactions, or whether to retain transactional access at all times, and to revoke only non-transactional access. We chose the latter option because it avoids needing to modify non-transactional code to manipulate page table entries. This choice is motivated by our goal to avoid overhead on non-transactional code.

When to restore access. Normal access permission can either be restored eagerly (i.e., as soon as there are no transactions accessing objects on a page), or it can be restored lazily (i.e., only when an AV occurs). We chose to make these changes lazily to reduce the cost that supporting strong atomicity adds to code inside



Access violation
Wait for conflicting tx-writers to finish
Restore normal access to the page
Return to faulting instruction

Figure 2. Response to AVs: we restore normal access in the AV handler and then re-execute the instruction.

an `atomic` block. Restoring access lazily avoids ping-ponging between access modes if the same data is accessed repeatedly inside transactions without being accessed non-transactionally.

Which permissions to restore on an AV triggered by a read. We considered whether to restore read-only access to the page, or whether to restore read-write access. Restoring read-only access is preferable if the page is also being read by transactions. On the other hand, restoring read-write access is preferable if the page is subsequently written non-transactionally. In practice, in our experiments, our dynamic code update mechanism (Section 5) reduces the number of AVs to such an extent that the choice of how to respond to AVs is not critical. We currently restore read-only access in response to an AV triggered by a read.

Visible or invisible page-level reading. As in transactional memory designs we can select between using "visible" or "invisible" reading, that is, we can select whether or not the presence of a transactional reader is directly visible to other threads [11]. Page-level visible reading allows an AV handler to determine if a normal write may conflict with a concurrent reading transaction. Conversely, invisible reading does not allow an AV handler to detect concurrent transactional readers.

We initially experimented with both approaches. However, the contention introduced by maintaining per-page reader information led us to focus on invisible reading, given our goal of reducing the overhead added to the STM.

Response to AVs. We identified two basic responses to AVs: either to leave the page protection unchanged and emulate the effect of the failing memory access as a short transaction ("emulate"), or to wait until it is safe to restore normal access before re-executing the instruction ("wait").

We chose the latter approach for two reasons. First, the information needed to emulate an individual instruction as a transaction is not immediately available at runtime—we would need to map the address being accessed back to an object-base/offset pair. Furthermore, so that the GC could trace the STM logs, we would need to determine whether or not the address being accessed holds a reference. Neither of these look-ups is designed to be fast in the Bartok runtime system. The second reason for using the "wait" strategy is that it leaves the page accessible to subsequent non-transactional reads without further AVs. Note that "wait" avoids deadlock because a transaction never needs to wait for non-transactional code.

Figure 2 summarizes this basic "wait" approach; we build on this in Sections 4–6.

Format of object references. Introducing a second mapping of the process's heap raises the question of how objects are identified: is one form of reference used throughout, or can an object be identified by either of its virtual addresses?

For simplicity we always represent object references using the address in the normal heap. Within a transaction, the translation from these normal references to addresses in the tx-heap is deferred until the actual native instructions that read or write the objects. In practice, a final peephole optimization pass can often combine this translation with another operation, e.g., when accessing an object's field, the translation becomes an adjustment to the offset. This approach means that, outside the STM, the language's runtime system does not need to be aware of the changes to the structure of the heap and that there is no need to marshal between different forms of object reference when data enters or leaves a transaction.

3.3 Correctness Argument

Our informal correctness argument is based on the notion that the page protection settings established before transactional memory accesses and checked after transaction validation are sufficient to detect conflicts with normal accesses.

The cases of pages that have been written by a transaction are straightforward: each transaction increments the writer count for the pages it writes, normal access is revoked before the transactional write is made, and an AV handler waits until the writer count is back to 0.

The cases of pages that have been read by a transaction are slightly more complex. Before the transaction's first read from a page it must have recorded the page's transition count and observed that the page was either inaccessible to normal code, or that it was read-only to normal code. After the transaction's normal validation work it must have observed that the transition count was unchanged. This means that the page was never writable by normal code at any point since the read. Note how it is necessary that the page-level validation be done after the STM-level validation. Otherwise, if the page-level validation were to be done first, a conflict would be missed if a normal write were to occur after the page-level checks but before the STM-level checks.

4. Avoiding AVs

The basic technique of Section 3 isolates transactions from normal memory accesses. In this section we introduce the techniques we use to develop a practical implementation from this foundation. The main idea is to identify "safe" memory accesses that cannot trigger AVs. We call an access "safe" if either:

- It is a normal access that cannot conflict with a concurrent transaction (that is, there will be no transaction that has started, but not yet finished committing or aborting, whose implementation will make a conflicting access).
- It is a transactional access whose implementation cannot conflict with a normal access (that is, there will be no conflicting normal access between the time of the transactional access and the time when the transaction finishes its commit or abort).

We employ static analyses in order to establish safety. So far we have used extensions to fairly standard techniques (so we do not describe them in detail). Our experiments indicate that these techniques may well be sufficient (Section 7). We discuss three kinds of safe access: safe accesses by the language implementation and runtime system (Section 4.1), safe accesses by non-transactional code (Section 4.2), and safe accesses by transactions (Section 4.3).

4.1 Safe Accesses by the Language Implementation and Runtime System

Many memory accesses made by the language implementation and runtime system are safe:

- Access to virtual method tables and array lengths. These are initialized when an object is allocated and are then immutable.
- Loads of values into dead registers. These are used to implement explicit null reference tests which are executed for their possible side effect of triggering an AV; the actual value loaded is not required.
- Access to the data structures used by the memory allocator and the STM. The memory allocator and STM never run transactionally, and their internal data structures are disjoint from those accessed by the application.

```
class Sequencer {
   ...
   private int ComputeUniqueSegments(int nthreads) {
     int nUniqueSegment = 0;
     for (int i = 0; i < nthreads; i++)
         nUniqueSegment += this.uniqueSegments[i].Count;
     return nUniqueSegment;
   }
   ...
}
```

(a) Source code.

```
Genome_Sequencer_ComputeUniqueSegments::
    ... // Prologue omitted for brevity
    mov edi,ecx
    mov esi,edx
    xor ebp,ebp
    xor ebx,ebx
    test edx,edx
    jle done
  loop:
    mov eax,dword ptr [edi+0x20]          // I1
    cmp ebx,dword ptr [eax+0x40000004]    // I2
    jae outOfRange
    mov ecx,dword ptr [eax+ebx*4+0x08]    // I3
    mov eax,dword ptr [ecx+0x40000000]    // I4
    call dword ptr [eax+0x40000088]       // I5
    add ebp,eax
    add ebx,1
    cmp ebx,esi
    jl loop
  done:
    mov eax,ebp
    ... // Epilogue omitted for brevity
    ret
  outOfRange:
    call throwNewIndexOutOfRangeException
```

(b) Native code exploiting safe run-time system accesses.

Figure 3. Exploiting safe memory accesses during compilation: `I2`, `I4`, and `I5` access immutable data managed by the runtime system, so access checks are not required on them. We exploit this by having them access the tx-heap at an offset of `0x40000000` from the normal heap. We show the actual code generated by Bartok except that, for clarity, we disable loop optimizations and add descriptive label names.

- The GC implementation. We use a stop-the-world GC. As in our earlier work [10], the GC is aware of the STM's data structure formats and traverses them if a GC occurs at a time when any threads are executing transactions.

These safe memory accesses cannot conflict with transactions, so do not require page-level access checks. We eliminate these page-level checks by compiling safe memory accesses so that they use the tx-heap mapping; as we discussed in Section 3 we never revoke access to pages in the tx-heap.

Figure 3 illustrates this point. The code fragment is taken from our C# version of the STAMP Genome benchmark [17]. It iterates over an array `uniqueSegments`. Each element of the array refers to a hashtable. `ComputeUniqueSegments` computes the total number of elements in these tables. It executes non-transactionally, but the tables themselves are manipulated by transactions.

The main work in the example is in the block following the label `loop`. `I1` loads the register `eax` with a reference to the array. `I2` compares the index being accessed with the array's bounds. `I3` loads a reference to a particular hashtable from the array. `I4` loads the virtual-method-table pointer from the hashtable, and `I5` performs the virtual call. The access made by `I2`, `I4`, and `I5` can all be made to the tx-heap because they access immutable data.

4.2 Safe Accesses by Normal Code

We identify normal memory accesses that are safe because they can never conflict with transactions. If a location is never accessed transactionally then all accesses to it are safe. Furthermore, if a location is never written to transactionally, then all read accesses to it are safe.

We identify such locations using a simple NAIT analysis based on Shpeisman *et al.*'s approach [25]. Concretely, we use Steensgaard's points-to analysis [27] to identify objects that are never accessed in transactions, and objects that are read-only in transactions. Each element in the points-to set of a load or store is marked as transactional or normal depending on the kind of access being performed. We can then transform normal memory accesses into direct accesses to the tx-heap if, after the analysis, none of the elements in the access's points-to set are accessed transactionally. Similarly, we can transform a read if none of these elements are written transactionally.

We perform a simple static escape analysis to identify objects that remain thread-local even though they are accessed transactionally as well as normally. All accesses to these objects are safe. We use a simple intra-procedural forward data-flow analysis to identify safe accesses during object initialization.

Returning to the code fragment from Genome in Figure 3, the above analyses enable the accesses at `I1` and `I3` to use the tx-heap directly. The access at `I1` is safe because there are no transactional stores to the `uniqueSegments` field (as detected by the first static analysis). Similarly, the access at `I3` is safe because the array that `this.uniqueSegments` refers to is never updated inside a transaction. Therefore, these instructions can be rewritten as:

```
mov eax,dword ptr [edi+0x40000020]       // I1
...
mov ecx,dword ptr [eax+ebx*4+0x40000008] // I3
```

4.3 Safe Accesses by Transactions

We identify transactional memory accesses that are safe because they can never conflict with normal accesses. These transactional accesses can be performed without needing to revoke normal access to the locations involved. This may allow fewer page protection changes to be made. It may also avoid some false conflicts.

We identify safe accesses by transactions using the same analyses that we use for safe accesses by non-transactional code. Statically, the results are promising: around 85% of the transactional memory-access instructions are safe in our experiments. However, dynamically, very few page protection changes are avoided. There are two reasons for this. First, array accesses are less frequently safe but, dynamically, form a substantial fraction of the accesses in our benchmarks. Second, a transaction can avoid changing a page's protection only if all of the data accesses it makes to the page are safe: a single non-safe access will trigger a protection change.

5. Dynamic Code Update on AVs

The techniques of Section 4 reduce the frequency of AVs. In this section we discuss how to accommodate workloads with genuine sharing between transactional and non-transactional code. Our technique patches non-transactional code that triggers frequent AVs, replacing the faulting instructions with ones to perform a short transactional access (Figure 4).

Our current approach is that, on an AV, we both update the source of the AV and we restore access to the page in question. This simple heuristic seems to work well in practice; the update avoids immediately subsequent AVs from accesses to nearby locations.

Dynamic code update also lets us bound the number of AVs that a process may incur: if a memory access is always updated upon triggering an AV then the number of AVs is limited by the number

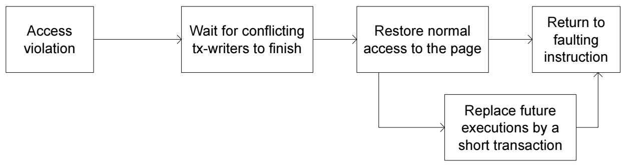

Access violation
Wait for conflicting tx-writers to finish
Restore normal access to the page
Return to faulting instruction
Replace future executions by a short transaction

Figure 4. Response to AVs: before re-executing the original instruction we consider updating the code to use a short transaction.

of memory access instructions. In practice, as we show in the results in Section 7, the number of AVs is vastly less than this bound. However, the bound provides assurance against pathological cases which might otherwise trigger vast numbers of AVs.

Since we are building on an ahead-of-time compiler we use binary patching to make code updates. Figure 5 illustrates this point using the `Hashtable.Count` property that is called from the earlier example in Figure 3 if it is compiled to use standard library classes. Figure 5(a) shows the original source code. We modify each basic block that may trigger an AV:

1. We generate an alternate version of the basic block with each possibly faulting operation replaced by a call onto a library function that performs the operation as a short transaction. These blocks are generated early during compilation when type information is available. Furthermore, the implementation of the short transactions can be inlined, STM meta-data can be re-used from one short transaction to the next, and so on. The blocks are placed out-of-line at the end of each function and control-flow edges to them are considered rarely taken for the purpose of register allocation. In the example in Figure 5(b) this is the block labeled `alternate`, reading an integer-valued field at an offset of 16 from the object reference held in `ecx`. (Our calling convention places the result in register `eax` and the first two parameters in `ecx` and `edx`.)
2. We generate a table listing the locations to patch: an AV after `primary` will be patched by atomically replacing the first instruction of `primary` with a branch to `alternate`. The patch is applied as a relative branch, selecting a 1-byte or 4-byte offset according to the distance between the blocks.
3. If necessary, particularly with 4-byte offsets, we pad the start of a block with a `nop` instruction so that only one instruction will be overwritten when patching.

In an implementation using JIT compilation we would recompile the method that we wish to update. Recompilation could give slightly better code quality than patching; the alternate basic blocks would not need to be present in memory ahead of time, and we would not need to pad code with `nop` instructions to overwrite. As our results show in Section 7 the scope for any speed-up is small.

6. Implementation

In this section we discuss implementation details from our prototype. The baseline Bartok-STM design is described in our earlier paper [10]. We discuss how our implementation avoids some cases of false sharing (Section 6.1), how we reduce the D-TLB pressure added by our optimizations for "safe" accesses (Section 6.2), how the page-level operations are combined with the STM implementation (Section 6.3), how we handle AVs (Section 6.4), how we support heap accesses from native code (Section 6.5), and various implementation ideas that proved ineffective (Section 6.6).

```
class Hashtable {
   ...
   public int Count {
     get { return this.count; }
   }
   ...
}
```

(a) Source code for the `Count` property that is used by the `ComputeUniqueSegments` fragment from Figure 3.

```
System_Hashtable_Count::
    push ebp
    mov ebp,esp
    sub esp,4
    mov dword ptr [ebp+-4],ecx
primary:                            // Patch to alternate
    mov eax,dword ptr [ecx+16]  // I10
done:
    mov esp,ebp
    pop ebp
    ret
alternate:
    mov edx,16
    call DirectReadInt32Obj
    jmp done
```

(b) Native code with support for patching AVs generated at `I10` where the field access is performed.

Figure 5. Example dynamic patch sequence.

6.1 Controlling Object Placement

We attempt to avoid transactional and non-transactional objects being located on the same page. We do this at allocation-time by using separate memory pools during normal execution and transactional execution. This avoids an AV when one object is allocated and initialized inside a transaction (revoking normal access to the page holding it) and then another object is allocated after the transaction.

We originally hypothesized that we would need to develop further heuristics to reduce false sharing—for example based on profiling object usage, or re-locating objects based on reachability during GC. Surprisingly, such extensions have proved unnecessary for the workloads we have studied. As we show in Section 7, the conservative implementation of strong atomicity using short transactions performs reasonably well so long as it is enabled selectively in response to AVs; performance is tolerant to occasional mis-expansion caused by false conflicts. Concurrent work by Schneider *et al.* seems to confirm the intuition that expanding a small number of accesses conservatively is unlikely to harm performance [22].

6.2 Pay-to-Use Implementation

The basic implementation of Section 3 avoids any change to how non-transactional code is compiled. However, the handling of safe instructions in Section 4 tends to degrade the quality of non-transactional code by modifying safe memory accesses to use the tx-heap instead of the normal heap: these accesses add pressure to the D-TLB, and the quality of the generated code is harmed by the address translations. The overall result is a slow-down of up to 35%, mainly due to the additional D-TLB pressure.

To mitigate this slow-down we use a dynamic-update mechanism once more: we initially run the code without the optimizations from Section 4 and, upon an AV, update the code to a version that includes the optimizations. We patch each block at most once: we use a single alternate block that uses the tx-heap for safe memory accesses and includes short transactions for any non-safe accesses.

There is one subtle problem that occurs when updating code from the runtime system. After updating a basic block we cannot branch immediately to the replacement if we are mid-way through

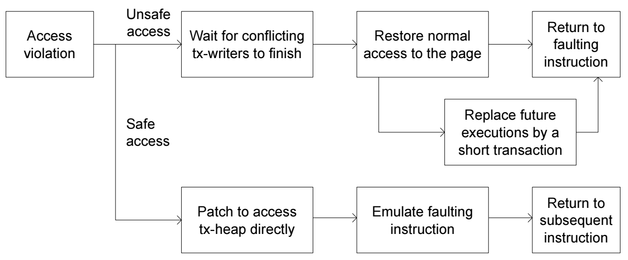

Access violation
Unsafe access
Wait for conflicting tx-writers to finish
Restore normal access to the page
Return to faulting instruction
Replace future executions by a short transaction
Safe access
Patch to access tx-heap directly
Emulate faulting instruction
Return to subsequent instruction

Figure 6. Response to AVs: safe instructions are handled by emulation to avoid deadlock.

it rather than at its entry point. This is no problem for AVs from application code: we wait for any conflicting transaction to finish, update the page permissions, and re-execute the instruction that triggered the AV. However, waiting within AV handlers triggered from runtime system functions can cause deadlock: e.g., the waiting thread may hold a lock in the storage manager, and the thread being waited for may subsequently try to allocate storage. We deal with this problem by handling AVs from runtime system functions differently from those from application code. For runtime system functions we emulate the instruction that faulted, using the tx-heap for any addresses that it accesses, before returning to the subsequent instruction. Figure 6 illustrates this approach.

6.3 STM Changes

We use a global array to store the per-page meta-data—i.e., the page's status field, its writer count, and its transition count. The array is indexed by the page's offset in the tx-heap, and each entry holds the three fields packed into 31-bits of an integer. The final bit is used as a spin-lock: a thread must hold the spin-lock when invoking a system call to change the page's protection settings.

We perform the page-level operations after the STM's existing test to eliminate object-level duplicate log entries. We experimented with a further layer of filtering to detect page-level duplicates that are not object-level duplicates. We could not find a way to do this in which the cost of the additional checks made the space saving worthwhile.

We inline fast-path versions of the page-level operations into the existing object-level STM code paths. The fast-path page-level read comprises: (i) reading the per-page meta-data, (ii) checking that the page's spin-lock is not held, (iii) checking that the page is not writable from normal code, (iv) logging the transition count from the meta-data. The fast-path page-level write replaces these last two steps with: (iii) checking that the page is not accessible from normal code, (iv) incrementing the page's writer count with an atomic compare and swap.

We must be careful when handling objects that span page boundaries; the address of the object header may be on a different page to the address later accessed. We use a bit in the object's STM meta-data to flag which objects span page boundaries and, if the bit is set, we use slow-path versions of the STM functions. The bit is initialized when an object is allocated. The common-case object allocation functions can always set the bit to 0 because they allocate memory from page-size pools, so the objects that they allocate cannot span page boundaries.

Our implementation of the short transactions that are introduced by dynamic updates broadly follows the design of Shpeisman *et al.* [25]: we read the per-object STM meta-data before and after reading the field's value from the tx-heap. We test that the meta-data is unchanged and that it indicates that there was no concurrent transaction. If there may have been a concurrent transaction then we branch to a slow-path version.

As in our earlier work [10], we make short transactions safe against version numbers overflowing in the STM meta-data. We do this by ensuring that at least one GC occurs before any given version number is re-used in a given object. During GC we validate all running transactions and roll back any that we find to be invalid. This means that a version number that was used before the GC can be re-used after the GC. In practice GC occurs sufficiently often that we never need to force additional collections to guard against version number overflow.

6.4 Handling AVs

We use C++ structured exception handling to register a user-mode handler for AVs. The handler is provided a copy of the thread's register context (which it can update before resuming the thread), along with information about the faulting address. These handlers run within the thread that incurred the AV.

Integrating this handler with the C# STM is complicated. The complexity comes from needing to allow the handler to wait for a conflicting transaction to finish while, in the meantime, the GC may need to traverse the stack of the thread that is running the handler. Consequently, we treat each site of a possible AV as a GC safe-point (generating tables to allow the GC to identify references from that thread's stack). However, unlike at normal GC safe-points, the thread's caller-saved registers may be live at instructions that trigger AVs. To avoid changing the format of the GC tables we treat these registers as conservative roots—i.e., if a register holds an address in the heap then any object at that address is prevented from being relocated or reclaimed. (In practice we have not seen the GC prolong object lifetimes in this way, but the technique is needed for correctness.)

6.5 Interaction with Native Code and System Calls

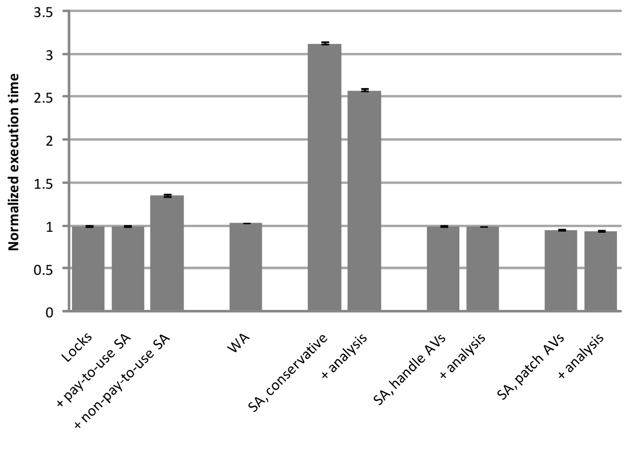
We support workloads in which heap objects may be passed to native code (or to system calls). The difficulty in doing this is that native code may trigger AVs when accessing data on pages in the normal heap. However, unlike code generated by Bartok, it is not typically possible to patch the native code, and it may not even be possible to handle the AV (e.g., the native code may have registered its own handler).

Our solution is to ensure that addresses passed to native code refer to an object's address in the tx-heap, so that the object remains accessible to the native code during its execution. This translation is done in the `GCHandle` library that is used to protect objects from being moved by the GC during native code accesses. We could extend this solution to prevent conflicts between native code and transactions by updating the object's STM meta-data when creating and destroying `GCHandle` structures.

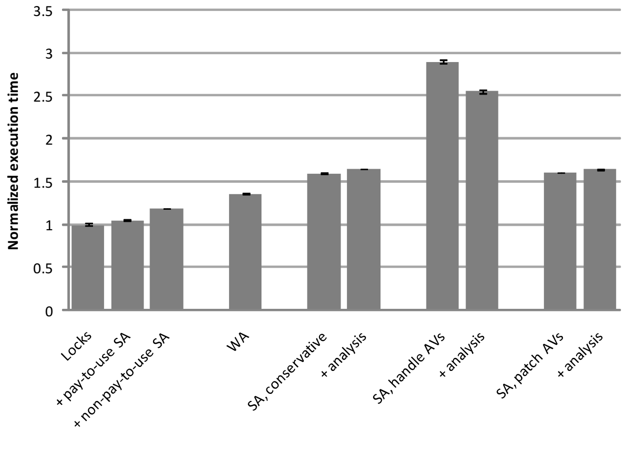
6.6 Other Experiments

We experimented with a number of techniques which are not used in our current implementation:

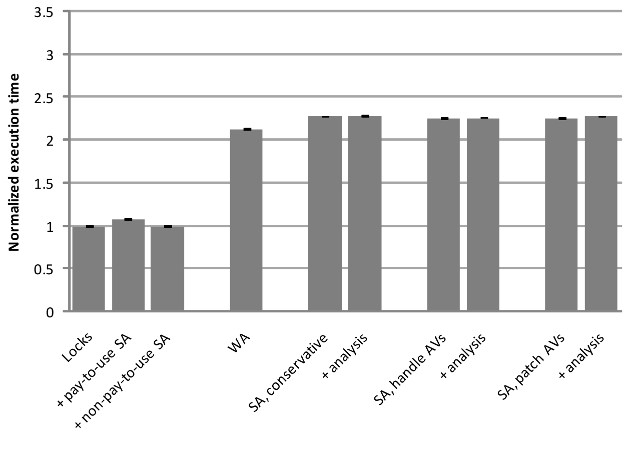
- We tried using x86 segmentation when accessing the tx-heap. We set the `gs` segment so that an address `gs:X` would access the tx-heap version of the normal address `X`. Using segmentation improved code density by replacing many 4-byte constants with 1-byte segment-override prefixes. However, it had no effect on performance.
- We hoped to use 4MB "large pages" for the tx-heap while using small 4KB pages for the normal heap. This approach could reduce the D-TLB overhead of using small pages for both heaps. However, although current processors support such heap mappings, the Win32 interface does not.



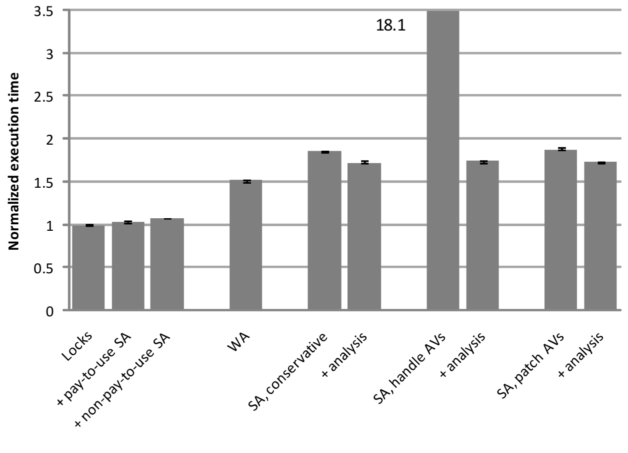
(a) Delaunay



(b) Genome



(c) Labyrinth



(d) Vacation

Figure 7. 1-thread performance. The first three bars show performance of the benchmark using locks, and the overhead of adding support for strong atomicity ("SA"). "WA" shows the original STM performance with weak atomicity. The final three pairs of bars show the performance of strong atomicity with conservative patching, with handling of AVs, and with patching in response to AVs.

7. Results

We studied the performance of our implementation with a set of four benchmarks. Three of these (Genome, Labyrinth, Vacation) were derived from the STAMP 0.9.9 benchmark suite [17]. We translated them from C to C#. This translation was straightforward; the original C code has a clear modular structure based on sets of `struct` definitions with accompanying functions. We made each `struct` a C# `class` exposing the functions as methods. We generally used C# library data structures (lists, vectors, and so on) in place of STAMP's internal library. The exception was that, as in STAMP, we used a hashtable without a shared counter field so that we get the same scaling in C and C#. We used the HTM versions of functions in STAMP, turning each hardware transaction into an `atomic` block. Our fourth benchmark is a Delaunay triangulation algorithm implemented following the description by Scott *et al.* [23]. We used Vacation when developing heuristics and tuning them. We report results for all four benchmarks.

We measured our implementation's performance using Windows Server 2003 Enterprise x64 Edition on a machine with 2 quad-core Xeon 5300-series Intel processors with 4GB physical memory. In addition to our main results on this 2*4-core machine, we confirmed that our 1-thread results were consistent with a set obtained on a machine running Windows Vista Enterprise SP1 on an Intel Core2 Duo T7300 CPU (2.0GHz) with 2GB physical memory. Making a virtual-protection change to a page takes 2-15K cycles in a multi-threaded process. Delivering an AV to a user-mode handler takes around 6K cycles. Our C# runtime system uses a generational copying GC.

Figure 7 shows the performance of 10 different 1-thread configurations of each program. We plot the median-of-9-runs wall-clock timings reported by the benchmark. Error bars show the min/max of the 9 runs. Timings are normalised against a lock-based implementation in which each atomic block acquires and releases a single global lock without using STM. On each graph, the first cluster of three bars shows the performance of lock-based implementations. The single "WA" bar shows the performance of the Bartok-STM implementation with weak atomicity. The pairs of "SA" bars show the performance of the Bartok-STM implementation with strong atomicity with and without the whole-program analyses from Section 4.2. We discuss each of these clusters of bars in turn:

Overhead of enabling strong atomicity in non-transactional programs. The first cluster of bars shows the performance of lock-based implementations and consequently the cost of supporting strong atomicity in programs that do not use transactions. "Locks" is the baseline implementation with coarse-grained locking and no runtime support for transactions. "Locks + pay-to-use SA" shows the cost of having runtime support for strong atomicity while still using locks for synchronization. The difference between this bar and "Locks" reflects the cost of compiling methods to enable dynamic code updates. The median cost varies from -1% to 8%. This could be avoided in an environment supporting JIT recompilation. "Locks + non-pay-to-use SA" shows the cost when safe accesses always use the tx-heap. The median cost varies from 0% to 35%. CPU performance counters indicate this is due to the increased D-TLB pressure caused by the tx-heap accesses.

Cost of STM with weak atomicity. The "WA" bar shows the overhead of an STM implementation with weak atomicity over the coarse-grained lock-based implementation. The overhead varies between benchmarks, largely dependent on the amount of time that the benchmark spends executing transactions. Delaunay uses occasional short transactions, so the overhead is low. Conversely, Labyrinth spends almost all of its time within transactions: the STM-based implementation executes 514 transactions, making a total of 9.0M memory writes. The STM-based implementation is 2.1x slower than the lock-based implementation.

These STM overhead numbers are substantially lower than those reported by Cascaval *et al.* for other implementations [4]. There are a number of significant differences between the implementations. First, Cascaval *et al.*'s implementations require memory-fence instructions in situations where they are not needed on the Intel processors that we use. Second, Cascaval *et al.*'s implementations use an update-log rather than an undo-log; their STM algorithms must therefore test for read-after-write accesses, rather than reading values in-place.

Cost of strong atomicity with conservative expansion. The "Conservative" pair of bars is for an implementation that conservatively expands non-transactional accesses that may conflict with transactions, following the approach of Shpeisman *et al.* [25]. This approach does not work well on the Delaunay benchmark. Delaunay involves substantial phases of private computation which access the same objects that are sometimes accessed by transactions: all of the normal accesses become expanded, even when no conflicts occur dynamically. Static analysis does not help here because most of the normal accesses are to objects that may be accessed by transactions; whether or not these transactional accesses occur depends on the input data set. (Scott *et al.* similarly observed that STMs that use indirection between object references and payloads perform poorly for this Delaunay workload because of the overhead they add to non-transactional code [23].)

Cost of handling AVs, without patching. The "Handle AVs" pair is for an implementation that handles each AV; this implementation updates the page's protection settings upon AV but does not patch the program. The results from Vacation illustrate the poor performance achieved without patching. Vacation involves large numbers of transactions mixed with non-transactional access to thread-local data. Without whole-program static analysis, "Handle AVs" performs poorly because of false conflicts when the GC locates thread-local data on the same page as shared data. It takes 18.1x longer to run than the lock-based implementation. It incurs 0.9M AVs and makes 2.2M page protection changes.

Patching AVs. The final pair of bars, "Patch AVs", shows the performance of an implementation using dynamic code update on AVs. This is the implementation that we would recommend using in practice.

This implementation performs well, when compared with the other implementations of strong atomicity, even without using whole-program static analysis. Unlike the "Conservative" implementations, it avoids the large number of transactional memory accesses incurred in Delaunay. Unlike the "Handle AVs" implementation, it avoids the large number of AVs incurred in Vacation.

In total, without using the static analysis, there are 59 AVs in Delaunay, 239 in Genome, 126 in Labyrinth, and 330 in Vacation. There are 82 page protection changes in Delaunay, 30K in Genome, 99 in Labyrinth, and 40K in Vacation. The difference between the number of AVs and the total number of page protection changes reflects the number of protection changes made when a transaction initially revokes normal access to a page.

Scaling. Figure 8 shows the performance of each configuration for 1..8 threads normalized against the program's lock-based performance with 1 thread. These show wall-clock timings, so perfect scaling would show a falling line, decreasing proportionally to the addition of threads. We plot lines for two representative lock-based configurations ("Locks" and "Locks + pay-to-use SA" from Figure 7) and two STM-based configurations ("WA" and "SA, patch AVs + analysis"). For clarity we omit error bars and the remaining six configurations: in each case the lock-based configurations would form one bundle of lines, and the STM-based configurations form a second bundle. The error bars from each line would cover the whole bundle. We see that, enabling strong atomicity, or using it, does not affect the scaling of these benchmarks with this STM.

Summary. Our results show that our implementation of strong atomicity over page-based memory protection hardware performs well when used along with dynamic patching of instructions that trigger AVs. It avoids the poor performance of conservative expansion on Delaunay. It avoids the vast number of AVs that occur without patching on Vacation. The overhead of this form of strong atomicity over weak atomicity is less than 25% on all these programs, even without static analysis.

8. Related Work

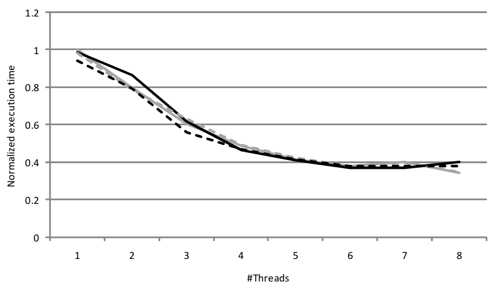
Hardware implementations of TM have typically supported strong atomicity [14], both in implementations based on extensions of MESI cache protocols (as in Herlihy and Moss' original design [12]), and in signature-based approaches [29]. Chuang *et al.* perform page-level buffering of transactional state in an HTM implementation if it overflows the cache [5].

Baugh *et al.* use memory protection hardware to separate transactional and non-transactional data in a hybrid TM [2]. Their approach inspired ours, although the designs ultimately differ in a number of ways. We use off-the-shelf page-based hardware, while Baugh *et al.* use fine-grained memory protection. We use separate virtual address mappings to prevent triggering AVs inside transactions, while Baugh *et al.*'s hardware provides per-thread flags to disable the delivery of access faults. Finally, we make dynamic code updates to reduce the high cost of AVs in off-the-shelf hardware. Baugh *et al.*'s hardware would make this unnecessary.

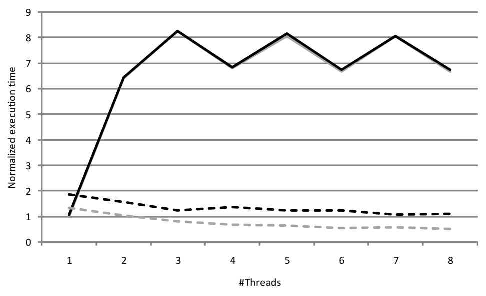
STM implementations have typically not implemented strong atomicity. However, there are a few exceptions. Hindman and Grossman's STM implementation ensures that all code is compiled to acquire an object's lock before accessing it [13].

Shpeisman *et al.* show how to implement strong atomicity by expanding normal accesses into optimized forms of short transaction [25]. They describe static whole-program analyses to reduce the number of accesses that need to be expanded.

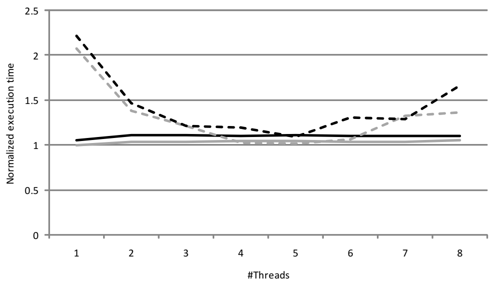
Concurrently with our work, Schneider *et al.* have extended Shpeisman *et al.*'s design to work dynamically during JIT compilation [22]. Only code produced by the JIT compiler is executed, so it suffices to ensure that there are no conflicts between transactional and normal execution in this body of code (rather than in the entire source of the program and libraries). Normal accesses are patched at run time if the JIT subsequently compiles a method that may conflict with code that has already been compiled. In one sense their approach is more conservative than ours because it statically detects possible conflicts against the body of already compiled code, rather than dynamically detecting actual conflicts. In another sense



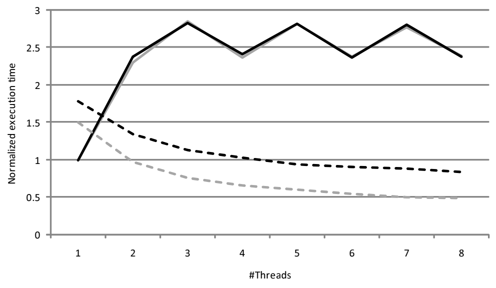
(a) Delaunay



(b) Genome



(c) Labyrinth



(d) Vacation

Figure 8. Scaling on 1..8 cores on a machine with 2*4-core CPUs. Each line is normalized against the program's 1-thread performance with a single global lock. Solid lines show lock-based implementations. Dashed lines show STM-based implementations. Gray lines show the original system, with support for weak atomicity. Black lines show the new system, with support for strong atomicity.

is it less conservative because it considers field accesses and not page-level conflicts. Our patching mechanism is simpler because it is not necessary to synchronize with other threads (if a thread does not see a patch then it may take a further AV), whereas Schneider *et al.*'s approach requires stop-the-world synchronization to guarantee that all threads immediately see any patches.

Matveev *et al.* propose a "virtual memory STM" (VMSTM) in which conflicts between transactions are managed at a page level [15]. A hardware "virtual memory filter" (VMF) enhances this by providing fine-grained conflict detection between transactions accessing the same page. Without this filter, Matveev *et al.* show that the basic VMSTM performs poorly because of the vast number of page-table updates and AVs. We use an existing STM to detect conflicts between transactions, and use page-level techniques only to detect conflicts between transactions and normal accesses. We make page protection changes lazily and dynamically update code to avoid repeated AVs.

Some distributed shared memory systems use the idea of mapping the same physical page multiple times [19]. Different objects on the page are accessed through different mappings. This provides, in effect, sub-page protection settings. We make multiple mappings to distinguish different kinds of access (transactional/normal) rather than different pieces of data. We could, in principle, use additional mappings for sub-page access control.

Ratanaworabhan *et al.* have studied the problem of detecting and tolerating asymmetric data races, where a variable is accessed directly by one thread while it is locked by another [21]. Within a critical section, memory accesses are redirected to a shadow version which includes a copy of the original contents of the location. These original values are compared against the "real" memory to identify and recover from data races. Concurrently with our own work, Rajamani *et al.* have used page-based memory protection to detect asymmetric data races [20]. Data is duplicated (both in the virtual and physical spaces), with one copy being used inside critical sections, while the other is made inaccessible.

9. Conclusion

In this paper we show how to use off-the-shelf page-level memory protection hardware to provide strong atomicity. We also develop a simple heuristic of patching instructions that produce possible conflicts; this approach is effective in benchmarks.

The sequential overhead of implementing strong atomicity appears similar to that of providing weaker guarantees such as Single Global Lock Atomicity (SGLA) [16]. Our implementation incurs extra costs in maintaining per-page meta-data. However, using page-based protection allows us to use an STM with in-place updates, rather than being restricted to ones that make deferred updates. Recent work has illustrated the high costs that implementations with deferred updates can incur [4].

It is interesting to consider whether or not strong atomicity is a desirable property for `atomic` blocks. It provides consistent behavior with typical HTM implementations, and semantics that appear easy to explain to programmers. However, even with strong atomicity, programmers must still understand transformations that may re-order memory accesses [8]. Furthermore, our implementation of strong atomicity is much more complex than implementations of other models like SGLA or dynamic separation [1].

A possible extension of our work would be to build a form of race detector, using our implementation to identify concurrent accesses to the same location from transactional and non-transactional code. The underlying use of page protection would be the same, but the AV handler would need to disambiguate genuine conflicts (in which case an error would be reported), from false conflicts (in which there is no error).

We believe that our approach is applicable to a wide range of STMs. Our implementation tightly couples the object-level and page-level logging functions, but these are logically separate and our informal correctness argument does not refer to low-level STM implementation details. As in Baugh *et al.*'s work, our approach could be extended to hybrid TMs [2].

Acknowledgments

We would like to thank the anonymous reviewers for their feedback on the paper, and our colleagues Darko Kirovski, Venkatesh-Prasad Ranganath, Kapil Vaswani, and Ben Zorn for discussions of the ToleRace and Isolator techniques for handling asymmetric data races, Úlfar Erlingsson for his assistance in using x86 segmentation, Alexandro Baldassin and Lingli Zhang for porting the Genome and Delaunay benchmarks to C#, and Nitin Kumar Goel and Richard Shupak for help with the `msdis` module on which our x86 emulation is based. We would also like to thank Dave Detlefs, Steven Hand, Mark Hill, Jon Howell, Michael Isard, Eric Koskinen, Yossi Levanoni, and Viktor Vafeiadis for comments on earlier drafts of this paper.

References

[1] Martín Abadi, Tim Harris, and Katherine F. Moore. A model of dynamic separation for transactional memory. In *CONCUR '08: Proc. 19th international conference on concurrency theory*, pages 6–20, 2008.

[2] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA '08: Proc. 35th international symposium on computer architecture*, pages 115–126, 2008.

[3] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *WDDD '05: Proc. 4th workshop on duplicating, deconstructing and debunking*, pages 48–55, 2005.

[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.

[5] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ASPLOS '06: Proc. 12th international conference on architectural support for programming languages and operating systems*, pages 347–358, 2006.

[6] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proc. 20th international symposium on distributed computing*, pages 194–208, 2006.

[7] Dave Dice and Nir Shavit. What really makes transactions faster? In *TRANSACT '06: Proc. 1st ACM SIGPLAN workshop on languages, compilers, and hardware support for transactional computing*, 2006.

[8] Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *MSPC '06: Proc. 2006 workshop on memory system performance and correctness*, pages 62–69, 2006.

[9] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 388–402, 2003.

[10] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN conference on programming language design and implementation*, pages 14–25, 2006.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd ACM symposium on principles of distributed computing*, pages 92–101, 2003.

[12] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proc. 20th international symposium on computer architecture*, pages 289–301, 1993.

[13] Benjamin Hindman and Dan Grossman. Strong atomicity for Java without virtual-machine support. Technical Report UW-CSE Technical Report 2006-05-01, 2006.

[14] Jim Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[15] Alex Matveev, Ori Shalev, and Nir Shavit. Dynamic identification of transactional memory locations. Unpublished Manuscript, Tel-Aviv University, 2007.

[16] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proc. 20th symposium on parallelism in algorithms and architectures*, pages 314–325, 2008.

[17] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. IEEE international symposium on workload characterization*, pages 35–46, 2008.

[18] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *HPCA-12: Proc. 12th international symposium on high-performance computer architecture*, pages 254–265. 2006.

[19] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proc. 9th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 179–190, 2003.

[20] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Isolator: Dynamically ensuring isolation in concurrent programs. Technical Report MSR-TR-2008-91.

[21] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Rahul Nagpal, Karthik Pattabiraman, and Benjamin Zorn. Detecting and tolerating asymmetric races. In *PPoPP '09: Proc. 14th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2009.

[22] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proc. 23rd ACM SIGPLAN conference on object oriented programming systems languages and applications*, pages 181–194, 2008.

[23] Michael L. Scott, Mike F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC '07: Proc. IEEE international symposium on workload characterization*, pages 107–113, 2007.

[24] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proc. 14th ACM symposium on principles of distributed computing*, pages 204–213, August 1995.

[25] 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. In *PLDI '07: Proc. 2007 ACM SIGPLAN conference on programming language design and implementation*, pages 78–88, 2007.

[26] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester, 2007.

[27] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proc. 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 32–41, 1996.

[28] 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: Proc. 2007 international symposium on code generation and optimization*, pages 34–48, 2007.

[29] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA-13: Proc. 13th international symposium on high performance computer architecture*, pages 261–272, 2007.