Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server

Ferad Zyulkyarov[†*] Vladimir Gajinov[†*] Osman S. Unsal[†] Adrián Cristal[†]
Eduard Ayguadé[†*] Tim Harris[‡] Mateo Valero[†*]

[†]Barcelona Supercomputing Center [*]Universitat Politècnica de Catalunya [‡]Microsoft Research
[†]{ferad.zyulkyarov, vladimir.gajinov, osman.unsal, adrian.cristal, eduard.ayguade, mateo.valero}@bsc.es
[‡]tharris@microsoft.com

Abstract

Transactional Memory (TM) is being studied widely as a new technique for synchronizing concurrent accesses to shared memory data structures for use in multi-core systems. Much of the initial work on TM has been evaluated using microbenchmarks and application kernels; it is not clear whether conclusions drawn from these workloads will apply to larger systems. In this work we make the first attempt to develop a large, complex, application that uses TM for all of its synchronization. We describe how we have taken an existing parallel implementation of the Quake game server and restructured it to use transactions. In doing so we have encountered examples where transactions simplify the structure of the program. We have also encountered cases where using transactions occludes the structure of the existing code. Compared with existing TM benchmarks, our workload exhibits non-block-structured transactions within which there are I/O operations and system call invocations. There are long and short running transactions (200–1.3M cycles) with small and large read and write sets (a few bytes to 1.5MB). There are nested transactions reaching up to 9 levels at runtime. There are examples where error handling and recovery occurs inside transactions. There are also examples where data changes between being accessed transactionally and accessed non-transactionally. However, we did not see examples where the kind of access to one piece of data depended on the value of another.

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

General Terms Design, Experimentation, Measurement

1. Introduction

Many researchers are studying the use of transactional memory (TM [18]) as a mechanism for writing concurrent applications. It is frequently argued that programming abstractions based on TM simplify the design of shared-memory data structures when compared with programming abstractions based on mutual exclusion locks.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

However, as we discuss in Section 2, there are very few applications that use TM. Many research results are either based on simple microbenchmarks (e.g. skip lists and red-black trees [11, 17]), or they are based on application kernels ([7, 12, 26, 35, 36]). These are valuable workloads for tuning TM implementations. However, because they are reasonably small, and because they have often been developed by TM researchers, they do not necessarily reflect how TM would be used in larger settings, or by programmers who are not aware of how TM is implemented.

In this paper we discuss our experience in building *Atomic Quake*, a multi-player game server in which we tried to use TM for all of the synchronization between threads. By studying a large, real, application we aim to gain insights into many of the choices that researchers are considering when designing programming abstractions based on TM. For example, whether or not strong atomicity is required [1, 4, 5, 23, 29], whether TM is useful for failure atomicity as well as synchronization, how frequently open nesting [24] or transactional boosting [16] are useful, which kinds of library calls, system calls, or I/O are used in transactions [6]. Our work also provides further data on typical sizes of transactions; how long they run for, how many distinct locations they access, and so on. This is useful for determining possible capacities for bounded-size hardware implementations of TM, or for motivating the adoption of unbounded designs [9, 10, 27, 30].

Our starting point for Atomic Quake is a lock-based parallel implementation of the Quake multi-player game server [2]. We describe the structure of the original application in Section 3. In developing Atomic Quake we took the extreme approach of using TM for all of the synchronization requirements. The resulting implementation comprises 27 400 lines of C code in 56 files. In the current code we have identified a total of 61 `atomic` blocks, mostly on the critical path of the application. Inside these `atomic` blocks there are I/O operations and system calls. There are long and short running transactions (200–1.3M cycles) with small and large read and write sets (a few bytes to 1.5MB). There are nested transaction reaching up to 9 levels at runtime. There are examples where error handling and recovery occurs inside transactions. There are also examples where data changes between being accessed transactionally and accessed non-transactionally. However, we did not see examples where the kind of access to one piece of data depended on the value of another.

In Section 4 we share our experience by giving examples from the Quake source code of cases where transactions fit well, and cases where they fall short. We discuss the challenges of replacing the lock based synchronization with transactions and, in particular, the challenges when trying to preserve the program logic and the original granularity of the critical sections.

Although our main focus is on programming examples, we also discuss the performance of Atomic Quake (Section 5). We have used the publicly-available Intel C++ STM Compiler (Prototype Edition 2.0). This lets us gather statistics about the transactions – for example, the distribution of their read/write set sizes and the distribution of their execution times. However, since we are working with a prototype compiler, our overall performance results should be seen as preliminary. In particular, because of problems when compiling our examples, we have had to disable optimizations. Earlier results have shown that TM-specific static analyses can significantly affect the performance of programs using TM [3, 15, 33], and so overall execution times are unlikely to be representative of a production implementation.

Nonetheless, despite using transactions ubiquitously for synchronization, our statistics show that we get less than 26% aborts on a 8-thread server configured to use a small game map. The results show a wide range of transaction sizes; many small transactions, with a number of long-running transactions. This emphasises the importance, when designing hybrid hardware/software TM implementations, to balance the need to avoid overhead on short-running transactions (to optimize the common case) while still supporting longer running transactions.

We discuss future work and conclusions in Section 6.

2. Related Work

STMBench7 [12] is a transactional version of the OO7 benchmark [8]. STMBench7 is implemented in both Java and C++ programming languages. There are lock-based and TM-based versions of the benchmark available for both languages. The Java implementation uses annotations to identify transactions. The C++ version uses explicit function calls to access an STM library. Transactions in STMBench7 contain recursive calls. However, they do not include I/O operations, or privatization patterns. The benchmark has large data structures that are accessed in operations of varying length. However, even the short operations' transactions are reasonably long and would not fit in typical hardware caches for HTM. Consequently, STMBench7 is useful for evaluating HTM-virtualization techniques but not so suitable for evaluating bounded-size HTMs.

STAMP [7] is a suite of applications written to use TM. The independent applications in the suite are algorithm kernels with different characteristics in terms of how long they spend running inside transactions, how large those transactions are, and how likely concurrent transactions are to conflict with one another. The STAMP applications can be configured for use with HTM (in which only the start and end of each transaction is identified in the source code), or for use with STM (in which case the shared memory accesses are also made explicit). The STM configuration therefore models the behavior of a compiler that can avoid the use of STM on memory accesses to thread-local locations. The structure of the atomic sections is simple – without nested transactions, privatization patterns, system calls, I/O, and error handling. STAMP does not provide lock based implementations of the applications, although the behavior of variants using a single global lock, in place of transactions, has sometimes been studied [1].

SPLASH-2 [35] is a suite of highly parallel applications which have subsequently been adapted to use TM for synchronization [25]. In general, the SPLASH-2 applications involve short, infrequent, critical sections. These make up a small proportion of the overall execution time.

The Haskell STM Benchmark suite [26] is a collection of programs, ranging from small synthetic workloads (e.g. contended access to a shared counter), to applications written by programmers who were not STM researchers (e.g. a parallel solver for Su Doku puzzles). Although the Haskell STM API is similar to library-based

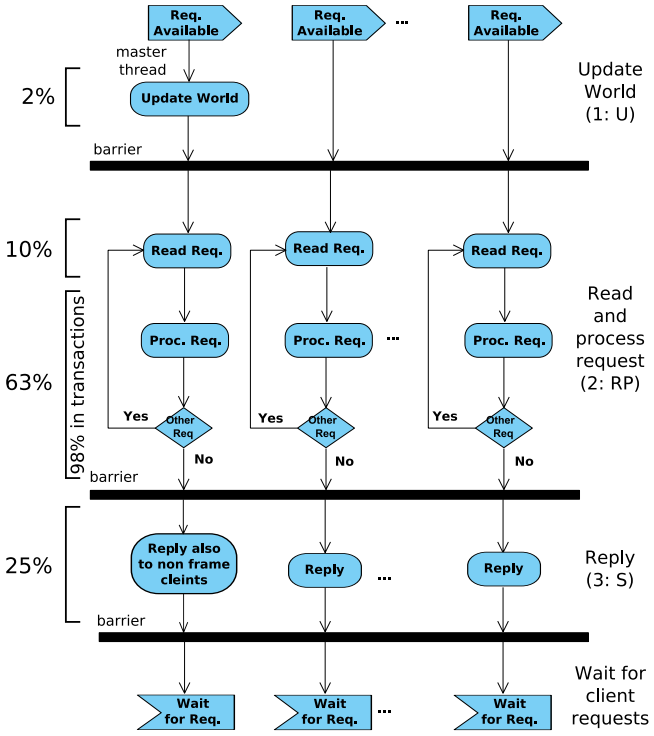

Figure 1. The game cycle in parallel Quake server with the execution time in the different phases (only the threads that have received a request are shown).

STM APIs in imperative languages, the core functional parts of these benchmarks are very different from current mainstream languages. It would be difficult to rewrite them in a language like C# or Java.

WormBench [36] is a configurable synthetic application implemented in an extension to C# that provides block-structured `atomic` sections. By controlling WormBench's input configuration, it can be made to resemble the transactional behavior of an existing application, or to create a runtime scenario that stresses a particular aspect of the underlying TM system.

TMunit [13] is an extensive framework for testing and evaluating STM libraries. It provides a domain specific language for writing unit tests. These tests can specify particular interleavings of threads in order to recreate problematic scenarios – e.g. when there is a non-transactional access to a memory location that occurs concurrently with a transactional access to the same location. TMunit can also generate test workloads by analyzing traces from the lock-based execution of a parallel program. Transactional workloads generated by TMunit run on top of an interpreter that makes direct calls to an STM library.

Although WormBench and TMunit are useful to study and debug the performance issues in different TM implementations, they cannot illustrate the practical use of transactions, or the strengths and weaknesses of different programming language constructs based on TM.

3. Quake Overview

In this section we discuss the existing parallel implementation [2] of the Quake game server [19] on which we base our work.

Multi-player Quake games use a client-server model in which the server has ultimate control over all the events that happen in the game; clients connect to the server and interact with each other

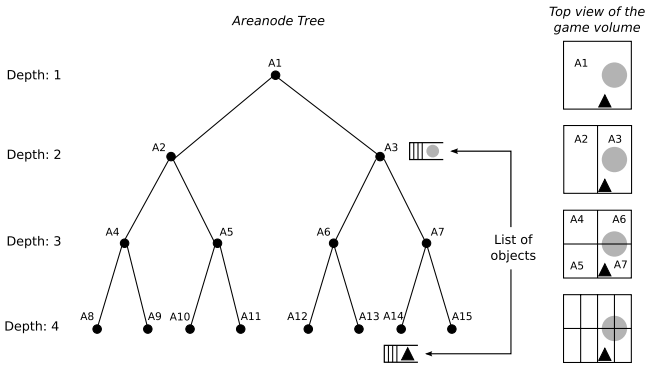

Figure 2. Mapping game volume into areanode tree (adapted from [2]). The tree shows the relationship between areanodes. The diagrams on the right show the areanodes' spatial relationship.

solely by sending requests to the server and receiving responses. There are two types of client request message: session-management operations (e.g. connect, disconnect) and interaction with the game world (e.g. move, shoot). For our research, we are primarily interested in the second type of message since processing them places the most significant load on the server.

3.1 Parallel Quake Server

The parallel Quake game server operates as a single multi-threaded process using barriers, locks, and condition variables for synchronization. The game executes in a series of rounds, known as *frames*. Each frame comprises three distinct phases: updating the physical simulation of the game world (U), reading and processing client requests (RP) and sending responses to clients (S). Server-wide barriers occur between each phase. The U phase is single threaded (executed by a designated master thread). The RP and S phases are executed in parallel by multiple threads. Figure 1 shows this structure and indicates, approximately, the relative time spent in each phase.

The server uses a configurable number of worker threads. This number is independent of how many clients connect to the server. However, in any particular game, each client is statically assigned to a specific worker thread. During the update phase, each of these worker threads waits in the main loop until it receives a request from a client. The first thread that receives a request is designated the *master thread* for that frame. The master thread executes the U phase. During the U phase, other threads may receive request from their own clients. Those threads that have received requests by the end of the master thread's work are said to *participate* in the frame. These threads proceed to the RP phase. It is possible that other threads have not received any requests while waiting. These threads remain waiting until the next U phase, even if they receive requests in the meantime.

During the RP phase, each thread proceeds to read from a buffer of incoming requests from its clients. Every client request is processed independently in the `SV_ExecuteClientMessage` function which incrementally builds the new frame as a reply message. After a thread has processed all of its client requests, it waits at a second barrier for the other threads to finish their own work. When all the threads are synchronized they enter the S phase.

During the S phase, threads send responses to the clients from which they received requests. The master thread sends responses to clients from which no request was received. At the end of S phase, threads wait at a final barrier until all of the responses have been sent.

3.2 Shared Data Structures

The threads in the parallel Quake server make concurrent accesses to three different data structures: *(i)* per-client reply-message buffers, *(ii)* a common global state buffer, and *(iii)* game objects.

The reply-message buffers are used to accumulate all of the pending messages that are to be sent to a client during the reply phase. These messages can originate from other clients (e.g. a text message between two players), as well as originating from the server (e.g. to provide information about the connection or disconnection of other players). Since multiple threads can write client's message buffer, the thread access to every message buffer is synchronized with locks, one per buffer.

The second kind of shared data structure is the global state buffer. This is updated during the U-phase and the RP-phase. This buffer contains data that needs to be sent to all clients, regardless of whether or not the server received a request from that client during the current frame. During the reply phase, each thread participating in the current frame uses this buffer to update the message buffers of its complete set of clients. The master thread performs this operation for clients belonging to threads that are not participating in the current frame. During the U-phase the global state buffer is accessed only by the master thread, and so there is no need for concurrency control. However, during the RP-phase, multiple threads may attempt to update the global state buffer at the same time. During this phase the global state buffer is guarded with a single lock.

The third kind of data structure are the game objects themselves. These represent the players and other entities on the game map. A complicated fine-grained locking scheme is used to control access to the game objects. Before executing an action, a worker thread identifies the list of the objects on the map that the player is likely to interact with and locks them to prevent concurrent access.

The data structure that the server uses to maintain information about the location of each object on the map is a balanced binary tree with depth 5 (including the root node) called the *areanode tree*. Nodes in the tree are known as *areanodes*. Each areanode represents a part from the entire 3D volume of the game. The children of an areanode represent two equally sized volumes that are obtained by dividing the volume represented by their parent with a vertical plane on either the *X* or *Y* axis. This division proceeds recursively, alternating between the two axes. All the areanodes in the same depth are a refined representation of the entire game volume. Moreover, each areanode has a list of the objects that are located in the volume that it represents. An object is associated with the highest areanode that completely contains it.

Figure 2 illustrates this structure. The root node of the tree is labeled *A1* and represents the entire 3D volume in the game. Its children, *A2* and *A3*, represent the two equally sized volumes obtained after partitioning the entire game volume with a vertical division plane on the *Y* axis. The triangle object is not crossing any division plane and it is in the list of objects associated with the *A14* leaf only. The circle object intersects with the plane that divides *A3* therefore it is put in the list of objects associated with *A3*, but not *A6* or *A7*.

A region-based locking scheme is used for concurrency control [22, 32]. To ensure exclusive access to the objects that a player interacts with, threads lock regions of the map prior to processing a request.

If an object is wholly located in a leaf of the areanode tree then a lock on that entire arenode is used (e.g. a thread requiring the triangle in Figure 2 would lock *A14*). If an object is not wholly located in a leaf then a lock on the object itself is used; this is acquired by temporarily holding a lock on the areanode holding the object, then locking the object, and finally releasing the areanode lock (e.g. a thread requiring the circle would temporarily lock *A3*).

```
1  switch(object->type) { /* Lock phase */
2      KEY: lock(key_mutex); break;
3      LIFE: lock(life_mutex); break;
4      WEAPON: lock(weapon_mutex); break;
5      ARMOR: lock(armor_mutex); break
6  };
7
8  pick_up_object(object);
9
10 switch(object->type) { /* Unlock phase */
11     KEY: unlock(key_mutex); break;
12     LIFE: unlock(life_mutex); break;
13     WEAPON: unlock(weapon_mutex); break;
14     ARMOR: unlock(armor_mutex); break
15 };
```

(a) Original implementation

```
1  atomic {
2      pick_up_object(object);
3  }
```

(b) Implementation with TM.

Listing 1. Per-type locking of objects.

4. Using Transactions in Quake

In this section we discuss the ways in which we re-structured the parallel Quake game server to use transactions. In summary, on a fully loaded server, about 98% of the request processing part in RP phase (Figure 1) executes in transactions, and the request processing part as a whole constitutes about 63% of the total execution time. There are 61 different `atomic` blocks. Almost all `atomic` blocks contain function calls. On average the static call graphs for the `atomic` blocks are 4 levels deep and contain 20-25 functions. Dynamically, transactions are nested up to 9 levels deep, with some `atomic` blocks occurring in recursive functions. All in all the process of developing Atomic Quake from the original parallel implementation took 10 man months.

We first introduce the programming abstractions provided by the compiler that we have used (Section 4.1). We then show how, using transactions, we can drastically simplify the structure of accesses to the core areanode data structure (Section 4.2). We also present cases where a basic transactional model is less effective: non-block-structured critical sections (Section 4.3), condition synchronization (Section 4.4), I/O within transactions (Section 4.5), error handling within transactions (Section 4.6), and cases where data changes between transactional and non-transactional accesses (Section 4.7).

4.1 Language Extensions for TM

We used Prototype Edition 2.0 of the Intel C++ STM compiler [20]. This extends the C and C++ language with keywords `__tm_atomic` and `__tm_abort`, and with attributes `tm_callable`, `tm_pure`, and `tm_unknown` that can be applied to function definitions.

The `__tm_atomic` keyword marks a block of code to be executed using transactions. The implementation provides exactly-once semantics in which the block is attempted until it commits successfully. For brevity, in code examples in this paper, we write `atomic` in place of the full keyword. Memory accesses within an `atomic` block are performed through an STM library and, when optimizations are enabled, the compiler tries to remove redundant accesses (e.g. when accessing provably thread-local data). The implementation provides a form of weak atomicity [5]: there is no concurrency control between concurrent transactional and non-transactional access to the same data.

```
1  /* Start locking leafs*/
2  lock(tree.root);
3  stack.push(tree.root);
4  while (!stack.is_empty()) {
5     parent = stack.pop();
6     if (parent.has_children()) {
7       for (child = parent.first_child();
8            child != NULL; child.next_sibling()) {
9             lock(child);
10            stack.push(child);
11       }
12       unlock(parent);
13    }
14 } /* End locking */
15
16  <move or shoot>
17
18 /* Start unlocking */
19 if (tree.root.has_children()) {
20    lock(tree.root);
21    stach.push(tree.lock);
22 } else {
23    unlock(tree.root);
24 }
25 while (!stack.is_empty()) {
26    parent = stack.pop();
27    for (child = parent.first_child();
28         child != NULL; child.next_sibling()) {
29         if (child.has_children()) {
30             lock(child);
31             stack.push(child);
32        }
34         else { // This is a leaf
35             unlock(child);
36         }
37     }
38     unlock(parent);
39 }
40 /* End unlocking */
```

Listing 2. Stack-assisted fine-grain locking.

The `__tm_abort` keyword is an explicit way to abort a transaction; execution continues after the `atomic` block involved.

The `tm_callable` attribute instructs the compiler to create a transactional version of the function, using the STM library as in an `atomic` block. The `tm_pure` attribute indicates that a function can be used without requiring a special TM implementation. It is used on side-effect-free functions like `sin` and `cos`. The `tm_unknown` annotation indicates that the function has unknown side effects. The implementation switches to *irrevocable mode* [34] if a `tm_unknown` function is called: only one transaction can be in irrevocable mode any one time. If a function is not annotated then it is assumed to be `tm_unknown`.

We declared math library functions to be `tm_pure` (since their results depend only on their input parameters). We manually re-implemented string manipulation routines so that they could be declared `tm_callable` (since their results depend on input data in the heap, which could have been modified by a calling transaction).

4.2 Where Transactions Fit: Complex Shared-Memory Data Structures

In our work, TM was most effective at simplifying parallel code that had previously used fine-grain locking. Listing 1(a) shows a code fragment where an object is first locked based on its type and then passed to a function that operates on it. The corresponding example is trivial with transactions (Listing 1(b)).

Other lock-based examples are more complicated, particularly when it is necessary to lock part of a collection of objects. In this case the programmer has to keep track of *(i)* the objects that might be accessed, *(ii)* the locks associated with these objects, and *(iii)* the order of acquiring locks to avoid deadlock. Again, when

```
1 for (i=0; i<sv_tot_num_players/sv_nproc; i++){
2     <statements1>
3     LOCK(cl_msg_lock[c - svs.clients]);
4     <statemnts2>
5     if (!c->send_message) {
6         <statements3>
7         UNLOCK(cl_msg_lock[c - svs.clients]);
8         <statements4>
9         continue;
10    }
11    <stamemnts5>
12    if (!sv.paused && !Netchan_CanPacket (&c->netchan)) {
13        <statmenets6>
14        UNLOCK(cl_msg_lock[c - svs.clients]);
15        <statements7>
16        continue;
17    }
18    <statements8>
19    if (c->state == cs_spawned) {
20        if (frame_threads_num > 1) LOCK(par_runcmd_lock);
21        <statements9>
22        if (frame_thread_num > 1) UNLOCK(par_runcmd_lock);
23    }
24    UNLOCK(cl_msg_lock[c - svs.clients]);
25    <statements10>
26 }
```

(a) Original implementation

```
1 bool first_if = false;
2 bool second_if = false;
3 for (i=0; i<sv_tot_num_players/sv_nproc; i++){
4   <statements1>
5   atomic {
6     <statemnts2>
7     if (!c->send_message) {
8       <statements3>
9       first_if = true;
10    } else {
11      <stamemnts5>
12      if (!sv.paused && !Netchan_CanPacket(&c->netchan)){
13        <statmenets6>
14        second_if = true;
15      } else {
16        <statements8>
17        if (c->state == cs_spawned) {
18          if (frame_threads_num > 1) {
19            atomic {
20              <statements9>
21            }
22          } else {
23            <statements9>;
24  } } } } }
25  if (first_if) {
26    <statements4>;
27    first_if = false;
28    continue;
29  }
30  if (second_if) {
31    <statements7>;
32    second_if = false;
33    continue;
34  }
35  <statements10>
36 }
```

(b) Implementation with TM.

Listing 3. Non-block-structured critical sections.

using transactions this becomes more straightforward because, for correctness, it is necessary only to identify the part of the code that has to execute atomically.

Listing 2 shows a simplified example from the parallel lock based version of Quake, implementing the region-locking technique from Section 3.2. The logic for acquiring locks uses supporting data structures such as stack. In addition to this logic, Quake uses a form of lightweight simulation whenever a player moves or shoots: the locations in the virtual world that the player/bullet will pass through are identified, and then the areanodes that represent these locations are locked. This form of fine grain synchronization is simplified when using transactions because the transaction logs implicitly track the objects involved, and the ability of the implementation to roll-back transactions replaces the use of an explicit simulation in the application.

4.3 Non-Block-Structured Critical Sections

Although TM is effective for managing the core operations on the areanode structure, we encountered many problems when attempting to use it ubiquitously in the game server.

One frequent source of problems was that, unlike existing TM benchmarks, critical sections were rarely block-structured. Consequently we could not simply replace a `LOCK` operation with "`atomic` {" and an `UNLOCK` operation with "}". Listing 3(a) shows an example with unstructured `LOCK` and `UNLOCK` operations. Such examples cannot be rewritten to use block-structured transactions without understanding the logic in the code; in more complex examples locks are acquired in one function and released in another.

In this example, the sections that complicate the use of block-structured transactions are <*statements4*> and <*statements7*>. We restructured the example as shown in Listing 3(b), introducing two additional variables, and moving these sets of statements to the end of the critical section. This approach increases the complexity in the control-flow logic in the function and, of course, it is specific to this kind of example in which the post-unlock operations are readily available in the same function.

The example would become more complicated if there were similar conditional blocks, or if new variables were declared in <*statements4*> or <*statement7*>. In such cases the declaration of those variables would have to be moved so that they are in scope when the statements are executed; the interaction with (e.g.) C++ destructors may add further complications.

This example shows a particular case where block-structured transactions do not work well. However, providing language support for non-block-structured transactions introduces fresh problems: rolling back a transaction may require stack-frames to be re-built, and a caller must be aware that a callee might commit a running transaction. (In other work, this kind of approach has been explored in conjunction with annotations for which functions may contain commit operations [21]).

Of course, if this application was written from scratch the developer might use a single `atomic` block starting at line 3 and ending at line 24 in Listing 3(a).

4.4 Condition Synchronization

In the parallel Quake server, conditional synchronization is used to implement the barriers shown in Figure 1. The STM compiler we use does not provide primitives for this kind of case, and so we have retained a lock-based barrier implementation.

```
1  pthread_mutex_lock(mutex);
2  <statements1>
3  if (!condition)
4      pthread_cond_wait(cond, mutex);
5  <statements2>
6  pthread_mutex_unlock(mutex);
```

(a) Original implementation

```
1  atomic {
2     <statements1>
3     if (!condition)
4        retry;
5     <statements2>
6  }
```

(b) Related implementation using `retry`

Listing 4. Condition-synchronization.

However, even with support for condition synchronization via `retry` [14], the implementation of barriers is another case where the lock-based code cannot be directly converted to use TM. The problem, as has been observed before [28], is that two `atomic` blocks would be needed: one to publish the arrival of a new thread at the barrier, and the second to block if necessary. The naïve translation from Listing 4(a) to Listing 4(b) would be incorrect.

4.5 I/O and Irrevocability Inside Transactions

There are two situations in which irrevocability is used in Atomic Quake. First, if a transaction calls a library function that is not marked `tm_pure` or `tm_callable` then the transaction becomes irrevocable no matter what the library function does. Second, there are examples where genuine I/O operations are attempted within transactions.

We encountered several examples of the first kind. In particular, many of the transactions in Atomic Quake use thread-private data that is accessed through the `pthread_setspecific` and `pthread_getspecific` operations. Listing 5 provides an example. Here, the application-defined `thread_id` is stored in a thread-private data area. However, when a function called inside an `atomic` block (e.g. `PR_ExecuteProgram`) makes a call to the `pthread_getspecific` API then the transaction implementing that function becomes irrevocable. This limits scalability because at most one transaction can run at any given time.

An ad-hoc workaround for examples like this is to declare functions like `pthread_getspecific` as `tm_pure` because it does not matter whether they are executed multiple times if a transaction is rolled back.

In our initial version of Atomic Quake almost all the transactions included the possibility of executing genuine I/O operations, e.g. printing debug messages when a player connects, or is killed. Using `tm_pure` is not appropriate here: the message would be re-printed if the `atomic` block were to be re-executed. Furthermore, it is not clear whether the I/O function would see tentative updates that its transactional caller had made, or whether it could be invoked by an invalid transaction. Where possible we handled such cases by hoisting the I/O out of the atomic block. An alternative would be a keyword such as `__tm_waiver` [25] to identify non-transactional code. However, as with using `tm_pure`, care would be needed over its exact semantics.

4.6 Error Handling Inside Transactions

When restructuring the source code to use transactions, there were many places where error cases are detected within transactions. In some, it was considerably easy to do so but in others the existing language extensions fell short. All the time we tried to adhere to the approach that the compiler developers provided for the C++ which is to try to commit when error happens and then handle the error.

```
1  void SV_Impact() {
2      atomic {
3          PR_ExecuteProgram();
4      }
5  }
6
7  __attribute__((tm_callable))
8  void PR_ExecuteProgram() {
9      int thread_id = pthread_getspecific(THREAD_KEY);
10     /* Continue based on the value of thread_id */
11     return;
12 }
```

Listing 5. Access to per-thread data with `pthread_getspecific`.

For example, Listing 6(a) shows a function that has to handle a critical system error. Our transactional solution is given in Figure 6(b). We took the calls to `Sys_Error` function outside the `atomic` block. Inside the transaction on the condition where error occurs, we save the type of the error in a local variable `error_no` and use break that ends the `for` loop and reaches to the end of the transaction. Then based on the value of the `error_no` we call the proper error handling function. Things get more complicated when function `Z_CheckHeap` is called inside another transaction. In this case, we will need to call `Sys_Error` function outside the outermost transaction. To be able to do so, we need a mechanism such as commit handlers to dynamically tell what to be done as a compensating action by providing relevant arguments to the handler function. For now we ignored the complicated cases like this by just letting the runtime switch to irrevocable mode.

It is worth discussing what would happen if the transaction is re-executed and during the subsequent execution of the transaction the observed error does not manifest. In this case there might be two situations: *(i)* the error was repaired or *(ii)* we lost detecting a hidden bug that would leave the program in undefined state. Based on this simple example, we reach to a conclusion that error handling in transactional code require deeper analysis and primitives helping to detect and recover from errors.

In the Quake code there are patterns of error handling that would greatly benefit from the failure atomicity. Listing 7 shows an example code fragment from part of the implementation of the request dispatcher function. In lines 18 and 25 the `PR_RunError` is called. This prints a stack trace and terminates the process. In this particular case the code from lines 2 to 26 can be wrapped in an `atomic` block and, instead of calling `PR_RunError`, it could abort the transaction. The abort would restore the original values of the global state stored in `pr_global_array` and `pr_global` and the execution would continue from line 27 at the end of function. The effect of this usage would be that client's request would not be processed as if they were lost on the network and the server will continue running. There are many similar uses like this but failure atomicity cannot be applied to all of them because it is important that the execution of the program can proceed safely.

4.7 Privatization

In TM research, the term *privatization* has been used to describe cases where data changes from being accessed using transactions, to being private to a given thread and being accessed directly [31]. Such idioms are problematic when using implementations of STM with weak atomicity: in some cases it is possible for the implementation to continue accessing locations transactionally (e.g. from a transaction that has experienced a conflict but not yet rolled back), concurrently with the direct accesses [11, 31, 33]. It is unclear whether such STM implementations should be treated as incorrect, or whether such inputs should be treated as a form of data race.

```
 1 void Z_CheckHeap (void)
 2 {
 3   memblock_t *block;
 4   LOCK;
 5   for (block=mainzone->blocklist.next;;block=block->next){
 6   if (block->next == &mainzone->blocklist)
 7   break; // all blocks have been hit
 8   if ( (byte *)block + block->size != (byte *)block->next)
 9     Sys_Error("Block size does not touch the next block");
10   if ( block->next->prev != block)
11     Sys_Error("Next block doesn't have proper back link");
12   if (!block->tag && !block->next->tag)
13     Sys_Error("Two consecutive free blocks");
14   }
15   UNLOCK;
16 }
```

(a) Original implementation

```
 1 void Z_CheckHeap (void) {
 2 memblock_t *block;
 3 int error_no = 0;
 4 atomic{
 5    for (block=mainzone->blocklist.next;;block=block->next){
 6       if (block->next == &mainzone->blocklist)
 7          break; // all blocks have been hit
 8       if ((byte *)block + block->size !=
 9          (byte *)block->next; {
10          error_no = 1;
11          break; /* makes the transactions commit */
12       }
13       if (block->next->prev != block) {
14          error_no = 2;
15          break;
16       }
17       if (!block->tag && !block->next->tag) {
18          error_no = 3;
19          break;
20       }
21    }
22 }
23 if (error_no == 1)
24   Sys_Error ("Block size does not touch the next block");
25 if (error_no == 2)
26   Sys_Error ("Next block doesn't have proper back link");
27 if (error_no == 3)
28   Sys_Error ("Two consecutive free blocks");
29 }
```

(b) Implementation with TM.

Listing 6. Error handling.

We encountered two kinds of privatization examples in Atomic Quake. However, each is a benign form of privatization that is correctly handled by typical STMs. The first example is the global state buffer that accumulates data for sending to all clients. The buffer changes between being exclusively accessible by the master thread, to being shared via transactions, to being shared in read-only mode. Each change is punctuated by a barrier, as in the examples of Spear *et al*. [31].

The second kind of example involves data being allocated inside a transaction, and then accessed directly by the thread that performed the allocation. Listing 8 shows an example. In the example, the function `Z_TagMalloc` allocates a memory block which is then initialized after the transaction.

5. Experimental Results

In this section we discuss the performance of Atomic Quake in terms of its run-time performance (Sections 5.1–5.2), and workload characteristics of the transactions is uses (Section 5.3).

```
 1 void PR_ExecuteProgram (func_t fnum, int tId){
 2    f = &pr_functions_array[tId][fnum];
 4    pr_trace_array[tId] = false;
 5    exitdepth = pr_depth_array[tId];
 6    s = PR_EnterFunction (f, tId);
 7    while (1){
 8       s++;    // next statement
 9       st = &pr_statements_array[tId][s];
10       a = (eval_t *)&pr_globals_array[tId][st->a];
11       b = (eval_t *)&pr_globals_array[tId][st->b];
12       c = (eval_t *)&pr_globals_array[tId][st->c];
13       st = &pr_statements[s];
14       a = (eval_t *)&pr_globals[st->a];
15       b = (eval_t *)&pr_globals[st->b];
16       c = (eval_t *)&pr_globals[st->c];
17       if (--runaway == 0)
18           PR_RunError ("runaway loop error");
19       pr_xfunction_array[tId]->profile++;
20       pr_xstatement_array[tId] = s;
21       if (pr_trace_array[tId])
22           PR_PrintStatement (st);
23   }
24   if (ed==(edict_t*)sv.edicts && sv.state==ss_active)
25       PR_RunError("assignment to world entity");
26   }
27 }
```

Listing 7. Using failure atomicity to recover from critical errors.

5.1 Experimental Methodology

For our experiments we modified the implementation by ensuring that *all* worker threads participate in every frame. We did this by modifying the barrier between the U and RP phases (Figure 1) to wait until all of the worker threads have received a packet from at least one of their clients. This ensures that all worker threads participate in the frame, and consequently it increases contention during the RP phase. This change lets us execute the RP phase in parallel with one client per worker thread; otherwise, we would have needed several hundred clients.

We test workloads with 1..8 clients, each running on a different commodity PC. The clients replay a pre-recorded trace of player actions, without human intervention. We use a fixed 1-room training map in order to encourage conflicts. We collect overall results from a run of 1 000 interactions with each client, starting from a point when all of the clients have connected. The statistics about transaction sizes are collected by the STM library from the whole program's execution. Our results show the mean of 4 runs.

We run the server on Dell PE6850 machine with 4*2-core x64 Intel Xeon processors with 32KB L1 I-Cache and 32KB L1 D-Cache private per core, 4MB L2 shared between the two cores on each die, 8MB L3 shared between all eight cores, and 32GB physical memory. The installed operating system was Suse 11.0. We use Prototype Edition 2.0 of the Intel C++ STM Compiler.

Our performance results must be seen as preliminary for three reasons. First, the prototype compiler was not able to generate transactional clones of some of the methods that we had marked `tm_callable`. This meant that any transactions calling these methods had to switch to irrevocable mode, limiting scalability. Second, we needed to compile the programs with optimizations disabled (`-O0`) because the compiler failed when optimizing some of the methods. Earlier work has shown that [3, 15, 33], STM-specific optimization can make a substantial difference to performance. Finally, we encountered problems at runtime when executing code fragments similar to those shown in Listing 5. The problem is that `pthread_getspecific` returns an uninitialized value which is subsequently de-referenced. The cause of the problem is not yet clear; it happens non-deterministically, and even in single-threaded runs. We have worked around the problem by passing `thread_id` values manually between functions.

```
1 void* buffer;
2 atomic {
3    buffer = Z_TagMalloc(size, 1);
4 }
5 if (!buffer)
6     Sys_Error("Runtime Error: Not enough memory.");
7 else
8    memset(buf, 0, size);
```

Listing 8. Privatization example.

5.2 Scalability and Performance Results

We evaluate three different implementations: STM, LOCK and STM_LOCK. In STM critical sections are implemented with transactions. In LOCK the critical sections are implemented with a global re-entrant lock. STM_LOCK combines both, using the global lock for concurrency control, but retaining the use of STM within critical sections. The difference between STM_LOCK and LOCK illustrates the straight-line overhead of using STM, so we can distinguish it from the scalability lost by aborted transactions. We also plot the IDEAL scalability (i.e. the scalability that would be achieved if there was no lock or STM overhead) to easily compare the different synchronization implementation across an upper bound.

With single-threaded executions we see 4x-5x overhead from LOCK to STM_LOCK. To a large extent this is likely to be due to needing to disable compiler optimizations.

We measured the throughput of the server with 1..8 threads, expressing the throughput in terms of the number of client request messages processed per second. In Figure 3 the throughput of each Quake version is normalized to itself when ran with a single thread. Because all the threads start to process the client requests at the same time, we can assume that measured results will match those when the server is 100% loaded – the requests' queue is not empty. In our diagram the LOCK version scales nicely up to 8 threads, but STM and STM_LOCK based versions scale up to 4 threads and at 8 threads are saturated. With two threads all versions scale following the ideal speedup curve as STM_LOCK even achieves super-linear speedup which we relate to two reasons: *(i)* the STM library at times performs a global initialization or cleanup which takes constant time with any number of threads or *(ii)* because when the two threads execute the same code segment in parallel the instruction cache misses of the one thread happen to be hits for the second and thus less time spent in waiting a cache line to be brought from memory to the L2. With 4 threads the LOCK and STM_LOCK scale again well, but the STM versions show poor scalability because there are retrying transactions and the time to process a request increases. With 8 threads the server is saturated and the performance of the STM version degrades and is worse than when running with 2 threads due to the many aborts. Because of the STM library instrumentations, executing the critical section in STM_LOCK takes longer which causes more threads to serialize on critical section when running with 8 threads and thus being less scalable than LOCK. Based on the last observation we notice that the scalability of the transactional version is limited by the aborts, whereas the lock based version by the serialized execution of the critical sections. The map that we used in the game session is small and represents high-conflict scenarios where the players interact with each other all the time. Unfortunately, because of the issues in the tool set that we used, we could not run experiments with larger maps. Also, it is noteworthy to say that the transactional Quake server may perform better if not fully loaded which may result in fewer conflicts because of non-interleaved execution of the critical sections.

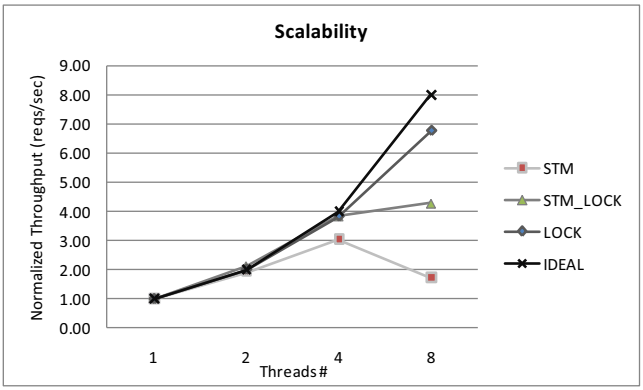

Figure 3. Scalability and speedup.

Threads	Transactions	Aborts		Irrevocable Tx
		Num	%	
1	36 667	0	0.00%	17
2	75 824	241	0.42%	31
4	166 000	2 612	1.58 %	85
8	477 519	76 771	25.50%	237

Table 1. Transactional characteristics.

5.3 Transactional Characteristics

Table 1 summarizes the characteristics of the transactions executed in Atomic Quake. Very few transactions become irrevocable, given our care to use `tm_pure` annotations. With 2 threads there are almost no aborts, but the abort rate increases exponentially with the number of threads. The high rate of aborts and consequently wasted computation time is the reason why the Atomic Quake server (STM) scales poorly. Since conflicts have such a significant impact on performance, this is also a signal that more research should be done on conflict avoidance, detection and resolution in STM libraries.

Table 2 shows a summary of the read and write sets for all the transactions. These results confirm typical assumptions about the prevalence of reads over writes inside transactions. The mean transaction sizes show that many of these transactions could be implemented in hardware that supports a few hundred transactional accesses. However, the high maximum sizes show that there are some larger transactions with read sets up to 1.5MB and write sets up to 344KB.

Table 3 shows per atomic block statistics obtained from a single-threaded execution of Atomic Quake. We omit entries for atomic blocks that were not executed in our test workload (e.g. walking in water).

A small number of atomic blocks make up the bulk of the transactional workload. These are located in functions in the critical path of the RP phase. The second group of transactions execute much less frequently; e.g. they include an example where the server sends a message to a client that has expended all their weapons. It is worth noting that the most frequently executed atomic block is a simple read-only non-nesting example which seems amenable to hardware implementation in a hybrid implementation.

6. Conclusion

In this paper we have described our experience porting the parallel Quake game server to use TM for all of its synchronization.

This is, of course, a somewhat extreme undertaking and not necessarily the way that we would recommend structuring a large

Threads	Read Set (bytes)					Write Set (bytes)				
	Min	Avg	Max	Total	Reads	Min	Avg	Max	Total	Writes
1	8	490	53 566	18 214 226	83%	0	98	11 161	3 639 952	17%
2	8	540	172 508	40 907 196	83%	0	115	47 784	8 737 623	18%
4	4	575	181 740	95 505 459	81%	0	131	52 032	21 737 915	19%
8	4	798	1 591 946	381 290 019	81%	0	183	352 640	87 837 969	19%

Table 2. Read and write set sizes.

program if developing it from scratch. As we showed in Section 4, there are some places where transactions substantially simplify the implementation of the server (e.g. the areanode tree), but there are other places where simple block-structured `atomic` sections are not a good fit (e.g. where there is complicated control-flow logic, and error handling).

Aside from the areanode structure, it is interesting to consider which of these programming problems are best tackled by introducing additional programming constructs, and which are best treated as problems that should be solved with traditional locking.

Our results from Section 5 show that, given care over a few examples like `pthread_getspecific`, it is possible to run most transactions in this workload without falling back to catch-all schemes like irrevocability. It is surprising, to us, that despite that coarse granularity of the transactions we use, we see an abort rate of less than 26% with 8 clients.

In future work we anticipate examining the performance of the transactions in Atomic Quake in more detail, once profiling tools are available to pinpoint the reasons for transactions being aborted. We also plan to compare the performance of the implementation with other alternatives and, in particular, to compare it with a separately-developed transactional implementation that we are building from the original single-threaded implementation.

The Atomic Quake implementation and benchmark driver scripts will be made available at `http://www.bscmsrc.eu`.

Acknowledgments

We would like to thank Intel for making the prototype C++ STM Compiler publicly available. We would also like to thank the anonymous reviewers for their useful comments. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center – National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852). Ferad Zyulkyarov is also supported by a scholarship from the Government of Catalonia.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proc. 14th ACM SIGPLAN symposium on principles and practice of parallel programming*, Feb. 2009.

[2] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *IPDPS '04: Proc. 18th international parallel and distributed processing symposium*, pages 72–81, Apr. 2004.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proc. 2006 ACM SIGPLAN conference on programming language design and implementation*, pages 26–37, June 2006.

[4] L. Baugh, N. Neelakantam, and C. 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, June 2008.

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

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, Department of Computer and Information Science, May 2006.

[7] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. 11th IEEE International Symposium on Workload Characterization*, September 2008.

[8] M. J. Carey, D. J. DeWitt, C. Kant, and J. F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *OOPSLA '94: Proc. 9th annual conference on object-oriented programming systems, language, and applications*, pages 414–426, Oct. 1994.

[9] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. 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, Oct. 2006.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS '06: Proc. 12th international conference on architectural support for programming languages and operating systems*, pages 336 – 346, Oct. 2006.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proc. 20th international symposium on distributed computing*, pages 194–208, Sept. 2006.

[12] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys '07: Proc. 2nd European systems conference*, Mar. 2007.

[13] D. Harmanci, P. Felber, M. Sukraut, and C. Fetzer. TMunit: A transactional memory unit testing and workload generation tool. Technical Report RR-I-08-08.1, Université de Neuchâtel, Institut d'Informatique, Aug. 2008.

[14] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 48 – 60, Feb 2005.

[15] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN conference on programming language design and implementation*, pages 14–25, June 2006.

[16] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proc. 13th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 207–216, Feb. 2008.

[17] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd Annual ACM SIGACT-SIGOPS symposium on principles of distributed computing*, pages 92–101, July 2003.

ID	Tx#	Nest	Dynamic Length (CPU Cycles)				Read Set (bytes)				Write Set (bytes)			
			Total	Min	Max	Avg	Total	Min	Max	Avg	Total	Min	Max	Avg
56	26 962	0	172 872 572	288	112 832	6 412	1 328 536	20	104	49	0	0	0	0
60	5 931	0	5 810 152	224	41 552	980	76 212	12	640	13	928	0	116	0.16
61	1 095	0	20 573 540	4 560	49 984	19 208	723 474	88	776	661	90	84	84	84
59	1 042	0	3 117 844	1 520	39 344	2 999	29 176	5	28	28	16 672	16	16	16
57	1 038	5	401 502 152	288 704	522 528	387 552	10 963 719	7 614	15 490	10 562	2 592 367	1 680	3 656	2 497
58	1 002	1	134 949 344	87 056	1 341 504	134 949	5 054 282	3 028	53 566	5 044	931 445	548	11 161	930
15	3	0	67 660	720	48 176	1 735	96	32	32	32	18	6	6	6
5	2	0	99 988	592	36 384	1 923	64	32	32	32	10	5	5	5
22	2	1	43 632	12 176	35 504	21 816	72	36	36	36	128	64	64	64
36	2	0	40 476	6 800	44 880	20 238	249	108	141	125	55	22	33	28
38	2	0	71 368	2 144	31 504	4 461	90	44	46	45	26	12	14	13

Table 3. Statistics for each atomic block from a single-threaded execution. Transactions that are not executed are not shown.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[19] ID Software. Quake. `http://www.idsoftware.com/games/quake/quake`.

[20] Intel Corporation. *Intel C++ STM Compiler Prototype Edition 2.0 Language Extensions and User's Guide*, Mar. 2008. `http://softwarecommunity.intel.com/isn/Downloads/whatif/stm/Intel-C-STM-Language-Extensions-Users-Guide-V2_0.pdf`.

[21] M. Isard and A. Birrell. Automatic mutual exclusion. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[22] S.-Y. Lee and R.-L. Liou. A multi-granularity locking model for concurrency control in object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, pages 144–156, Feb. 1996.

[23] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabi, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proc. 20th annual symposium on parallelism in algorithms and architectures*, pages 314–325, June 2008.

[24] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hoskin, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proc. 12th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 68–78, Mar. 2007.

[25] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA '08: Proc. 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 195–212, Oct. 2008.

[26] C. Perfumo, N. Sonmez, S. Stipic, A. Cristal, O. Unsal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *CF '08: Proc. ACM international conference on computing frontiers*, pages 67–78, May 2008.

[27] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proc. 32nd annual international symposium on computer architecture*, pages 494–505, June 2005.

[28] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM.

[29] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabi, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proc. ACM SIGPLAN conference on programming language design and implementation*, pages 78–88, June 2007.

[30] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 139–150, June 2008.

[31] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proc. 26th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 338–339. Aug. 2007.

[32] A. S. Tanenbaum. *Distributed Operating Systems*. 1994.

[33] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proc. 2007 international symposium on code generation and optimization*, pages 34–48, Mar. 2007.

[34] A. Welc, , B. Saha, and A.-R. Adl-Tabatabi. Irrevocable transactions and their applications. In *SPAA '08: Proc. 20th annual symposium on parallelism in algorithms and architectures*, pages 285–296, June 2008.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA '95: Proc. 22nd annual international symposium on computer architecture*, pages 24–38, June 1995.

[36] F. Zyulkyarov, S. Cvijic, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. WormBench: A configurable workload for evaluating transactional memory systems. In *MEDEA '08: Proc. 2008 workshop on memory performance: dealing with applications, systems and architecture*, Oct. 2008.