Debugging Programs that use Atomic Blocks and Transactional Memory

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

[†]BSC-Microsoft Research Centre [*]Universitat Politècnica de Catalunya [‡]Microsoft Research

[†]{ferad.zyulkyarov, osman.unsal, adrian.cristal, mateo.valero}@bsc.es

[‡]tharris@microsoft.com

Abstract

With the emergence of research prototypes, programming using `atomic` blocks and transactional memory (TM) is becoming more attractive. This paper describes our experience building and using a debugger for programs written with these abstractions. We introduce three approaches: (*i*) debugging at the level of `atomic` blocks, where the programmer is shielded from implementation details (such as exactly what kind of TM is used, or indeed whether lock inference is used instead), (*ii*) debugging at the level of transactions, where conflict rates, read sets, write sets, and other TM internals are visible, and (*iii*) debug-time transactions, which let the programmer manipulate synchronization from within the debugger—e.g., enlarging the scope of an `atomic` block to try to identify a bug.

In this paper we explain the rationale behind the new debugging approaches that we propose. We describe the design and implementation of an extension to the WinDbg debugger, enabling support for C# programs using atomic blocks and TM. We also demonstrate the design of a "conflict point discovery" technique for identifying program statements that introduce contention between transactions. We illustrate how these techniques can be used by optimizing a C# version of the Genome application from STAMP TM benchmark suite.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids; D.3.4 [*Programming Languages*]: Processors—Debuggers

General Terms Languages, Reliability

Keywords Transactional Memory, Debugging

1. Introduction

Atomic blocks are programming language constructs for controlling concurrency in multi-threaded applications. Many researchers, including ourselves, have developed research prototypes for `atomic` blocks, some based on static analysis for automatic lock inference, and others based on various kinds of transactional memory (TM), either implemented in software (STM) or hardware (HTM) [13].

However, based on our experience developing complex transactional applications such as Atomic Quake [26], QuakeTM [7], RMS-TM [12], WormBench [25] and Haskell-STM [9] we found it frustrating to use current debuggers when writing programs using atomic blocks and TM. This experience has motivated us to study how to extend debuggers to better support transactional applications.

In this paper we present the new principles and approaches that we have developed. In particular, we introduce the idea of distinguishing between debugging at the level of `atomic` blocks, and debugging at the level of transactional memory. When working at the level of `atomic` blocks, the programmer should only be aware that the blocks run atomically and in isolation: the programmer should not see implementation details such as exactly how `atomic` blocks are built over TM, or the internal algorithms used by a given TM implementation. Thus, when a breakpoint fires in an `atomic` block, the interrupted thread will be the only one in any `atomic` block. If the programmer single steps through the block, they will not see conflicts, transaction reexecutions, and so on. A rule of thumb is that, at this level, the experience using the debugger should be the same, whether `atomic` blocks are built over TM, or whether they are built over a static analysis for lock inference. We discuss our overall design in Section 2, and `atomic`-level debugging in Section 3.

Conversely, when debugging at the lower level of transactions (Section 4), the programmer is presented with a view of the implementation of their program. This view is intended for debugging performance errors—for instance, identifying the instructions that are responsible for conflicts between transactions. Transactions represent the runtime execution of `atomic` blocks and have various attributes such as the number of aborts, status, priority, nesting level, and read and write sets. This information is helpful in debugging pathological cases such as forms of starvation [2]. In addition, besides finding errors, the debugger must be extended to handle basic information about the transactions, such as the read and write sets, in order to present the user with a correct view of memory. For example, in lazy versioning STMs that buffer the updates until commit, the user might be confused if the values of the variables in a watch list do not change while stepping inside an `atomic` block. Moreover, the user might be interested in debugging inside a particular `atomic` block only when a specific change in its state happens such as a transition from valid to invalid. To help in these situations, the user can additionally use the debugger to monitor for various events associated with the change of the transaction status and when for example a conflict is detected, the debugger will break automatically and display relevant information such as conflicting threads, statements and memory addresses.

We provide a *conflict point discovery* mechanism. At the end of the program's execution, conflict point discovery identifies the

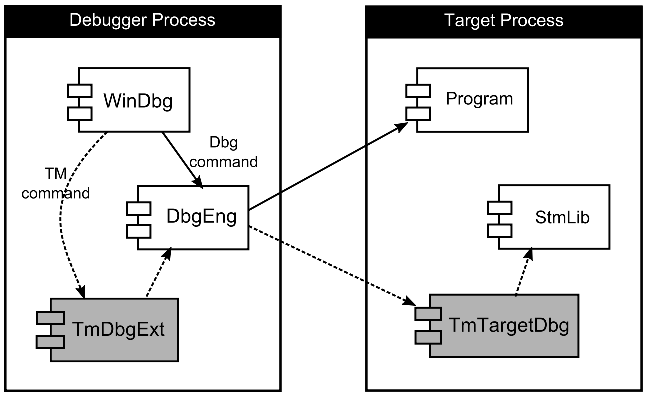


Figure 1. Decoupled design approach for the debugger extension. The components in gray represent our extension and the dashed lines represent TM operations. The implementation of TmTargetDbg is specific for our STM library and the implementation of TmDbgExt is specific for WinDbg family of debuggers.

statements within the source code where the transactions have conflicted, together with additional contextual information. This feature is useful in optimizing applications by reducing the abort rate of their transactions. To demonstrate its use in practice, we iteratively optimized a C# version of Genome application from STAMP TM benchmark suite [3] by applying conflict point discovery and examining the state of the transactions. Initial version of Genome did not scale at all, but after the optimization we achieved consistent performance with the original C version of Genome.

In Section 5, we discuss debugger features that the user can use to dynamically control transactions and their state. This provides mechanisms to create and to remove *debug-time transactions* under the control of the debugger without changing and recompiling the source code. These features are useful when investigating errors such as data races, atomicity violations and order violations—much as existing debuggers provide abstractions for modifying the contents of data in memory when investigating errors.

We discuss related work in Section 6 and conclude in Section 7.

2. Design and Implementation

We prototyped our ideas in an extension module for the publicly-available WinDbg debugger [18]. Concretely, we target transactional C# applications compiled with Bartok [10] compiler. However, our design decisions are motivated by maintaining applicability of our approaches to other debuggers, other TMs, and to non-TM implementations of `atomic` blocks.

WinDbg is a multi-purpose debugger for Win32 applications. Its functionality can be extended by using the Microsoft Debug Engine Extension APIs [17]. WinDbg extensions are Dynamic Link Libraries (DLL) that implement and export a number of callback functions. Some of these callbacks are required by the debugger for the extension's integration, and other callbacks implement the additional user commands that extend the debugger functionality or let it visualize specific data structures.

Bartok is an ahead-of-time C# compiler with language level support for `atomic` blocks. The runtime execution of the `atomic` blocks in applications compiled with Bartok is handled by an STM library which from now on we will refer to as Bartok-STM. Bartok-STM updates memory locations in-place by logging the old value for rollback in case a conflict happens. It detects conflicts at an object granularity, eagerly for write operations and lazily for read operations.

Operation	*Description*
GetTxStatus	Get the status of the transaction.
SetTxStatus	Set the status of the transaction.
GetPriority	Get the priority of the transaction.
SetPriority	Set the priority of the transaction.
GetReadSet	Get the read set of a transaction.
GetWirteSet	Get the write set of a transaction.
AddToReadSet	Add entry to the read set.
RemoveFromReadSet	Remove an entry from the read set.
AddToWriteSet	Add entry to the write set.
RemoveFromWriteSet	Remove an write from the read set.
GetNestingLevel	Get the nesting level of a transaction.
GetOriginalValue	Get value before a speculative update.
GetSpeculativeValue	Get value after speculative update.
IsTxIrrevocable	Check if a transaction is irrevocable
SwitchToIrrevocable	Switches transaction to irrevocable mode.
StartIrrevocableTx	Starts a transaction in irrevocable mode.
CommitIrrevocableTx	Commits an irrevocable transaction.
SplitTx	Splits a transaction

Figure 2. The API of TmTargetDbg component.

In the following sections we introduce our design and implementation of the debugger extension and then from Section 3 return to the high-level debugging approaches.

2.1 Design Approach

We have chosen a decoupled design for extending WinDbg. Our design consists of two components: a debugger extension library (TmDbgExt) and an STM-library debug helper (TmTargetDbg). Figure 1 shows the structure of the system. TmDbgExt implements the end-user debugger commands for use with `atomic` blocks and transactions. It is dynamically loaded by WinDbg and runs as part of the debugger process and uses the debugger engine (DbgEng) to access the target. TmDbgExt is specific to a particular debugger, but independent of the TM in use. Conversely, TmTargetDbg runs in the address space of the program being debugged. TmTargetDbg is specific to the TM, but independent of the debugger.

We were inspired by the approach described in Lev's presentation [14]. Comparing with Herlihy and Lev's subsequent paper [11], we have only one component at the debugger side (TmDbgExt), whereas Lev's design uses two (*tm_db* and a *Remote Debugging Module*, RDM). tm_db defines an common interface for implementing extensions to debug transactional applications. It can be used with all debuggers providing the `proc_service` interface and is independent of the TM implementation. RDM provides tm_db with functionality for debugging a particular TM. Within the target process, the TM runtime system provides a support layer (RTDB). We chose to avoid placing any TM-specific components on the debugger side—the developer of our TmTargetDbg will not need to know about the debugger and vice versa. Ultimately, we might be tempted to define a common interface and communication mechanisms between TmDbgExt and TmTargetDbg—but this seems premature at the moment.

We also experimented with an alternative approach which implements all the functionality in the debugger extension (TmDbgExt), without the helper component in the target process. In this approach the debugger extension is coupled with the STM library implementation and depends on the layout of the data structures, size of buffers, alignment, and so on. For instance, suppose that we want to check if a specific memory address is in the read set of a transaction. The debugger-side module would need to be coupled to the layout of the data structure representing the read set entry and the field where the address is stored. Also, the module has to know any possible alignment restrictions that the compiler might apply. Modifying the read set entry data structure by adding a new field or compiling for different architecture (e.g., 64-bit) would require changing and re-testing the debugger extension. We felt that this model was not a good fit with rapidly-evolving transactional memory systems.

We believe our decoupled design approach can readily be applied to implementations of `atomic` blocks over other TMs; the details of TmTargetDbg will vary, depending on the exact data structures used, but the approach will remain the same.

2.2 Interaction Between TmDbgExt and TmTargetDbg

The interaction between the debugger and the STM library has two levels of indirection. First, TmDbgExt accesses TmTargetDbg over the debugger engine API and then TmTargetDbg accesses the STM internals (see Figure 1). TmTargetDbg acts as a wrapper for the STM library and exports a set of functions listed in Figure 2. TmDbgExt may query or modify the STM state by setting a call to one or more of these functions. To safely execute a function in the target process, TmDbgExt saves the process context prior the call and restores it after the call. For simplicity, we have designed the prototypes of the TmTargetDbg functions in a way that if the return value is larger than a register (e.g., an array or a data structure) the value is stored in a temporary location and the address to this location is returned.

2.3 Internal Breakpoints

We use breakpoints to implement many of our new debugger features. For instance, when debugging at the level of `atomic` blocks and a normal breakpoint fires inside an `atomic` block, we must check that the current transaction is valid, and then "clean" the visible state of other threads (e.g., by rolling back transactions that other threads are in). This provides the impression of isolation. In many examples like this we either need to cause the target process to execute STM-helper functions, or we need to roll forward application code in the target process.

Both of these operations involve adding temporary breakpoints in addition to those set by the user (e.g., we must regain control after rolling forward). We refer to these as "internal" breakpoints. As described later in the paper, we used internal breakpoints to override the step command to interpret `atomic` blocks as a single statement (Section 3.1), to implement watchpoints (Section 4.1), to implement debug-time transactions (Section 5.1) and to split `atomic` blocks (Section 5.2).

We use a breakpoint-time callback to distinguish ordinary user-breakpoints from internal breakpoints. The callback overrides the default debugger behavior of suspending the target program when an internal breakpoint is hit and, if necessary, it executes complementary actions associated with the internal breakpoint.

The diagram in Figure 3 shows how the callback works. When a breakpoint is hit, the callback checks whether it is a normal breakpoint, or an internal breakpoint. If it is a normal breakpoint and the event thread is executing a transaction, the callback executes a complementary action to switch the transaction to irrevocable mode [22, 23] and breaks to the debugger prompt (Section 3). If the breakpoint is internal, the callback executes a complementary action based on its type (purpose). Also, depending on the type of the internal breakpoint the debugger may either break or continue execution as if the breakpoint is not hit.

3. Debugging at the Level of Atomic Blocks

In this section we discuss our approach for debugging transactional applications at the level of `atomic` blocks. We extend the debugger to model the semantics of `atomic` blocks, presenting the user with the impression that they run with atomicity and isolation (even when the underlying implementation uses TM).

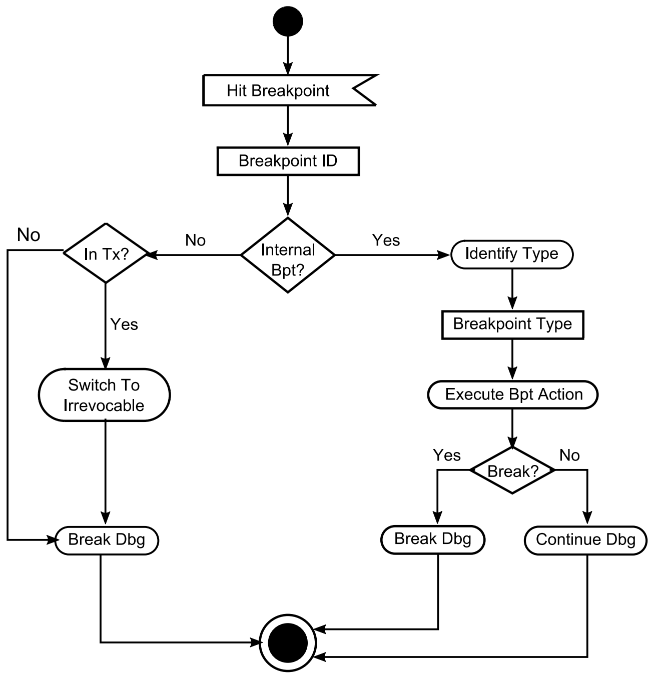


Figure 3. Using a breakpoint callback to distinguish between normal user breakpoints and the internal breakpoints. Also, the breakpoints fired during transaction execution may require to do complementary actions such as switching the transaction to irrevocable mode.

Consequently, when debugging a program using `atomic` blocks, we provide facilities to single-step over entire blocks so that they appear as indivisible operations (Section 3.1), and to step into a block while preserving the appearance that it is executing in isolation (Section 3.2).

By analogy, a debugger for a language implemented with garbage collection (GC) will abstract away the details of how the heap is structured—e.g., when single-stepping, it would not step into the GC implementation if it runs, and it would clear and re-set data watchpoints if the underlying objects are relocated.

3.1 Stepping Over Atomic Blocks

The atomicity property of `atomic` blocks guarantees that the statements comprising the `atomic` block execute either all or none. When debugging higher-level concurrency errors in transactional applications, the user may therefore have the expectation that the debugger will execute the `atomic` block in its entirety without being interested in what is going on inside—much as the user may step over a complete function call.

Earlier work that studied the construction of parallel programs with `atomic` blocks and TM [19, 21] and our experience of developing such applications [7, 12, 20, 25, 26] suggests that programmers organize transactional synchronization between threads in a different, more abstract, way by relying on the atomicity of complete transactions but not identifying the individual shared data structures to protect them with locks. In this approach, the concurrency errors in transactional applications are coarser and manifest on the level of `atomic` blocks and not on the level of individual statements inside the `atomic` block.

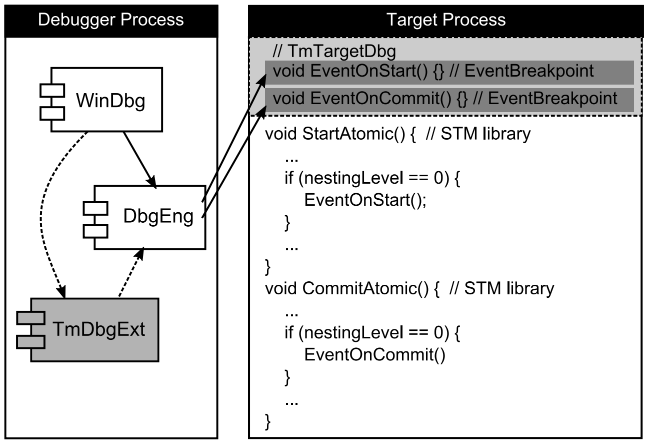
```
 1 atomic {
 2    ...// Initialize the bounding box
 3    if (ent->v.modelindex)
 4       SV_FindTouchedLeafs (ent, sv.worldmodel->nodes);
 5    ent->num_leafs = 0;
 6    if (ent->v.modelindex)
 7       SV_FindTouchedLeafs (ent, sv.worldmodel->nodes);
 8    if (ent->v.solid != SOLID_NOT) {
 9       tm_block_flag = true;
10       i=1;
11       node = sv_areanodes; // Areanode tree
12       while (1) {
13          if (node->axis == -1)
14             break;
15         if (ent->v.absmin[node->axis] > node->dist) {
16             node = node->children[0];
17             i *= 2;
18          }
19          else if (ent->v.absmax[node->axis] < node->dist) {
20             node = node->children[1];
21             i = i*2 + 1;
22          }
23          else
24             break;
25       }
26       if (ent->v.solid == SOLID_TRIGGER)
27          InsertLinkBefore (&ent->area, &node->trigger_edicts);
28       else
29          InsertLinkBefore (&ent->area, &node->solid_edicts);
30    }
31 } // end atomic
```

Figure 4. The body of the `atomic` block in function `SV_LinkEdict` from Atomic Quake which is responsible for changing the location of an object such as a player from its old to the new position in the map (areanode tree).

Existing debuggers are not aware of `atomic` block boundaries and so they do not provide the illusion of atomicity. In such a case, instead of helping to identify the concurrency problem, the debugger may cause additional confusion, especially if the `atomic` block contains sophisticated logic and function calls. For example, Figure 4 shows the body of the `atomic` block in function `SV_LinkEdict` taken from the Atomic Quake code [26]. This function is responsible for changing the location of a game object (e.g., a player) from one to another location in the game map. Suppose that we are searching for an error and want to see the state of the map data structure (i.e. `sv_areanodes` line 11) before and after executing the `atomic` block. When we advance in the debugger, we would normally proceed by stepping into each of the statements inside the `atomic` block. This will show the intermediate changes, rather than the overall effect of the block. Furthermore, if the transaction implementing the `atomic` block aborts part-way through, the user may find execution back at the start of the first statement.

Without debugger support for TM, a workaround for this problem is to put a breakpoint at the end of the `atomic` block (i.e. line 31) and to continue execution up to that point. This has the effect of executing the `atomic` block as a single statement.

To support execution of complete `atomic` blocks, we provide a distinct `tmstep` operation. This steps over the whole `atomic` block in a single operation. To implement this, TmDbgExt puts internal breakpoints at the functions exported by TmTargetDbg that are called at the start and end of an outermost transaction (Figure 5). These breakpoints are enabled by default and, when the first one is hit upon starting a transaction, the debugger continues to execute until it reaches the matching commit function. When committing the outermost function the breakpoint on the commit function is hit and this time the debugger switches back to normal stepping mode.



```
// TmTargetDbg
void EventOnStart() {} // EventBreakpoint
void EventOnCommit() {} // EventBreakpoint
void StartAtomic() {  // STM library
   ...
   if (nestingLevel == 0) {
      EventOnStart();
   }
   ...
}
void CommitAtomic() {  // STM library
   ...
   if (nestingLevel == 0) {
      EventOnCommit()
   }
   ...
}
```

Figure 5. The illusion of atomicity in TmDbgExt is implemented by putting internal breakpoints at the functions `EventOnStart` and `EventOnCommit` called by the STM library when outermost transactions start and commit respectively. When the breakpoint on function `EventOnStart` is hit, TmDbgExt continues execution in `go` mode, and when the breakpoint on `EventOnCommit` is hit TmDbgExt restores the execution to `step` mode.

3.2 Stepping Inside Atomic Blocks

The isolation property of `atomic` blocks guarantees that threads will not see the intermediate updates made by a thread which executes an `atomic` block. Consequently, we provide a mechanism to preserve isolation when stepping into `atomic` blocks.

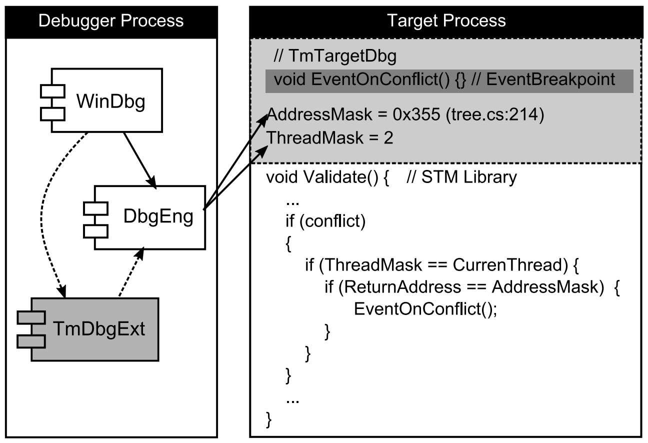
This is intended for debugging errors within a single `atomic` block—for instance, if our code in function `InsertLinkBefore` (Figure 4 line 26–29) is wrong, and its internal logic needs to be examined. Debugging within an `atomic` block is activated automatically when a breakpoint is hit while executing a transaction. For example, if the user puts a breakpoint at line 27 in Figure 5 then the user will be able to advance inside the `atomic` block by stepping over each statement.

To preserve the appearance of isolation, we must take care to prevent interference between transactions—e.g., consuming speculative updates from concurrent transactions, operating on an inconsistent view of memory, or being aborted and reexecuted. For instance, in the code example from Figure 4 the root of the areanode tree is assigned to a local variable (line 11), and if a second transaction commits a change to the root, then `InsertLinkEdict` might operate on invalid data. Debugging logic inside an `atomic` block based on invalid values but not yet detected conflict, camouflages the actual problem and violates isolation.

We preserve isolation by switching the transaction being debugged into irrevocable mode [22, 23] (i.e., a transaction that is guaranteed to commit). Our implementation of irrevocable transactions is simplistic: before switching to irrevocable mode the TM library validates all transactions and makes sure that the only transaction being executed is the irrevocable one (rolling back any others). Thus, while stepping through an `atomic` block, the user will see only actual values and never see transactional aborts.

If a conditional breakpoint is reached while executing a transaction, we first validate the transaction, and if the validation passes successfully we break into the debugger. If validation fails, then the transaction is aborted and reexecuted, without breaking into the debugger.

This is necessary to prevent invalid transactions from falsely suspending the execution, and reflects our intended semantics for



```
// TmTargetDbg
void EventOnConflict() {} // EventBreakpoint

AddressMask = 0x355 (tree.cs:214)
ThreadMask = 2
```

```
void Validate() {   // STM Library
    ...
    if (conflict)
    {
        if (ThreadMask == CurrenThread) {
            if (ReturnAddress == AddressMask)  {
                EventOnConflict();
            }
        }
    }
    ...
}
```

Figure 6. Filtering uninteresting events. The debugger extension sets filter mask for thread id 2 and instruction address to monitor for conflicts. When conflict happens the STM checks the masks and if they are true calls the function `EventOnConflict` which is set a breakpoint.

`atomic` blocks which are designed to abstract the details of particular TM implementations.

4. Debugging at the Level of Transactions

When debugging at the level of transactions, the debugger extension deliberately exposes a TM-based implementation of `atomic` blocks. The aim is to provide the user with means to discover and reason about pathological situations, such as those described by Jayaram *et al.* [2]. Such examples can harm overall performance or prevent progress.

When debugging at the level of transactions, the user can step into the statements inside an `atomic` block without changing the transaction into irrevocable mode like we did in Section 3. In such a case, when advancing line-by-line over the source code, the execution of two or more `atomic` blocks may be interleaved, and the user may observe the effect of this interleaving on the TM system. At any time, the user can see the state of any active transaction, and inspect the following attributes:

- The status of the transaction such as valid, invalid, blocked.
- The priority of the transaction.
- How many times the transaction aborted and reexecuted.
- The transaction's read and write set.
- Whether the transaction is irrevocable.
- The ID of the thread executing the transaction.
- The original and the speculative value of a variable.

The debugger must distinguish between original and speculative values in order to support some of its existing features. For example, a user might have a variable in a watch list that is speculatively updated in a transaction. The underlying value of this variable will not change in TM systems with lazy versioning (i.e., which buffer updates until commit). In such cases, the debugger must monitor transactional writes to check if this variable is updated by a transaction and display its most current value. Herlihy and Lev [11] make a more detailed analysis of this problem and discusses the changes for the current debuggers in order to support it.

By combining these primitive queries, we have also implemented richer operations to intersect the read or write sets of two or more transactions. This is intended to help the programmer understand the common data sets between the transactions, and to discover pathological cases that prevent transactions to progress and hurt the overall application performance.

However, the user would usually prefer not to dive in the world of this complicated debugging which requires knowledge about the workings of the underlying TM implementation until a specific event happens such as a transition from a valid to an invalid state due to a conflict. We discuss transaction events in more detail in the next section. Then, in Section 4.2, we describe how we implement conflict point discovery to help users to find the locations where transactions conflict. Later, in Section 4.3, we iteratively optimize a C# ported version of Genome application from the STAMP benchmark suite by studying conflict points and examining the state of the transactions.

4.1 Transaction Events

Our debugger extension can monitor transaction events that relate to changes in the status of the transaction and its read and write sets. The events that users can monitor are:

- Transaction start.
- Transaction commit.
- Transaction abort.
- New read or write set entry.

A user can set a watchpoint on any of these events. When the watchpoint is triggered, the debugger breaks and provides contextual information such as the event thread, the conflicting transactions, the conflict addresses, or the entry being added into the read or write set. To avoid interrupting the target process at uninteresting places, the user can also introduce filters for these events so that the event is triggered—for example, only if the conflict happens on a specific `atomic` block(s).

To be able to catch the transaction events as they happen, we define stub functions in TmTargetDbg for each of these events. The stubs are called by the STM library when the event happens. To break on an event, TmDbgExt places a internal breakpoint on the entry to the relevant stub. Also, to filter out irrelevant events, TmDbgExt can modify a *filter mask* variable defined in TmTargetDbg (see Figure 6). Depending on the filter criteria the STM library decides whether or not to call the corresponding event function. We enable these tests only when compiling in debug mode.

4.2 Conflict Point Discovery

To help aid performance debugging, we have developed a conflict point discovery mechanism. Conflict point discovery is a debugger feature that provides the exact source-code statements where memory accesses are involved in a conflict. Along with the line numbers, it includes contextual information such as how many times a specific statement was involved in a conflict, whether due to read or write access, and the `atomic` blocks where the conflicts occurred. Figure 7 shows an example output from the C# version of Genome application.

In recent empirical studies of developing transactional applications, Rossbach *et al.* [21] and Pankratius [19] report that the very first version of transactional applications often suffer from poor performance due to unanticipated overheads of the underlying STM system. Therefore, providing profiling information that developers can use to quickly optimize `atomic` blocks is important for the adoption of TM systems. Many researchers [3, 7, 20, 24, 26] have observed that one of the primary overheads in TM workloads is due to the aborts. Conflict point discovery provides information for reducing the abort rate and thereby improving overall performance.

```
File:Line         #Conf.  Method     Line
------------------------------------------------------------------
Hashtable.cs:51   152     Add        if (_container[hashCode] ...
Hashtable.cs:48   62      Add        uint hashCode = HashSdbm ...
Hashtable.cs:53   5       Add        _container[hashCode] = n ...
Hashtable.cs:83   5       Add        while (entry != null)
ArrayList.cs:79   3       Contains   for (int i = 0; i < cont ...
ArrayList.cs:52   1       Add        if (count == capacity - 1)
```

Figure 7. Example output generated by conflict point discovery for the C# version of Genome application.

We support conflict point discovery by using further "stub" functions to provide abstraction over the underlying STM library. These stubs are called when the STM library does book-keeping work. In effect, this automates the *reach point* technique we used in earlier work [7], by removing the need for manual instrumentation of code. We experimented with an alternative implementation that operates entirely on the debugger side, but the overhead of additional internal breakpoints was prohibitively high.

In Bartok-STM, conflicts can be detected in the write barriers, intermediate validations of the read set and the commit method (which also validates the read set). In the stubs we add to read and write barriers, we log the return address of the STM operation, along with the address of the memory location being accessed (see Figure 8). The return address of these functions is the place in the user code where the actual access to the memory is done. If the STM library detects a conflict while handling any of these methods we record the return address together with the origin of the conflict—whether caused by read or write. If the address is already recorded, then we increment a conflict counter associated with it.

Of course, as with many debugging techniques, this approach adds a probe effect because of the extra logging. We have measured the probe effect over the Red Black Tree micro benchmark which reflects even the minimal overheads in an amplified scale. Figure 9 shows relative difference between a binary compiled without any additional logging and binary compiled with the logging required for conflict point detection. Column *Execution Time* shows the relative difference between execution times and column *Aborts* shows the relative difference between abort rates. This experiment suggests that while the probe effect may change the fine-grain behavior of the program it does not introduce or remove high level contention. Qualitatively, when reducing contention on hot spots identified by conflict point discovery, contention in the underlying program is reduced. Similarly, programs with low contention under normal execution have low contention under conflict point discovery.

We do not currently try to identify the actual data structures that are involved in conflict. The reason is that we use a managed environment with a copying garbage collector which relocates objects. There is no stable way to identify an object over the course of its lifetime. Identifying a "hot" shared counter is trivial, but dynamically allocated data structures do not have symbolic names for internal nodes. In future work we would like to examine stable naming schemes for these cases.

4.3 The Debugger in Action

In this section we describe how we iteratively optimized a C# version of the Genome application from the STAMP TM application suite [3]. We use conflict point discovery to examine how transactions progress.

The STAMP STM version of Genome is a gene sequencing application implemented in C using TL2 STM library [6]. We initially ported this application from C to C# in a direct manner by annotating the `atomic` blocks using the available language constructs that the Bartok compiler implements. In the original version of Genome, the memory accesses inside `atomic` blocks are made through explicit calls to the STM library, whereas in the C# port the STM library calls are automatically generated by the compiler. Our observations optimizing the C# version therefore do not necessarily reflect aspects of the manually-instrumented C program.

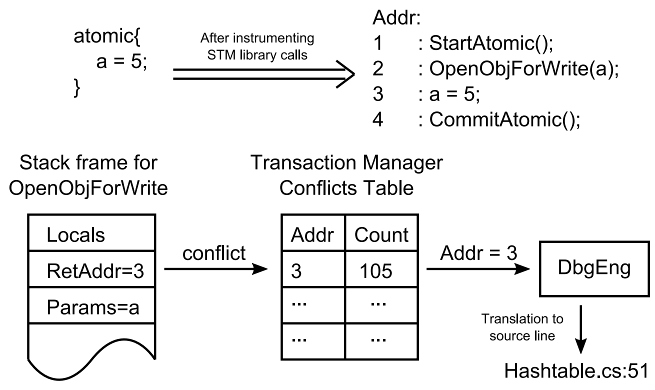


Figure 8. This figure shows how we identify the locations in the source code where conflicts happen. We modified the read and write barriers in the STM library to log their return address in the user code. When conflict is detected, we record the return address associated with the conflicting memory access in *Conflicts Table* and increment the conflict counter. At the end of the execution, using the debugger engine (DbgEng) we translate the addresses into source lines.

Threads	Execution Time	Aborts
1	0.0%	n/a
2	0.4%	1.2%
4	6.0%	4.5%
8	10.6%	1.6%
16	5.6%	10.0%

Figure 9. The probe effect of additional logging to support conflict point detection. In this experiment we used the Red Black Tree microbenchmark.

We performed our experiments on a 4*2-core CPU with 2 hardware threads per core. We show the effect of the different improvements on the normalized performance and on the reduction in the abort rate in Figure 10 and Figure 11 respectively. For comparison, we also show variants where we used a global lock in place of the `atomic` blocks (prefixed with L). We developed four variants using `atomic` blocks:

Unoptimized Genome (Unopt). Our first version of the C# Genome application had poor performance and did not scale. The reason for this was a very high abort rate. Using conflict point discovery, we saw that most of the conflicts happened in the first phase of the Genome application when duplicate gene segments are filtered by adding them to a hashtable. The highest contention was in two conflict points: (*i*) the test in a loop that checks whether a bucket already contains the entry to be added, and (*ii*) when incrementing a shared counter that indicates the number of elements in the hashtable. After a careful look at the implementation of our hashtable we realized that it is a variation of an open addressing hashtable where entries are stored in the bucket array and the array is probed for empty slots on collisions.

Using chaining hashtable (Opt). The open addressing hashtable performs poorly in our implementation because Bartok-STM uses

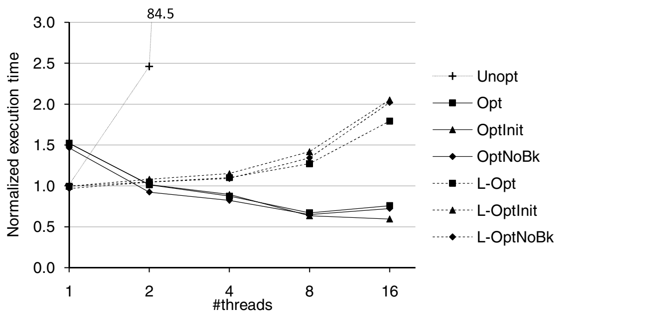


Figure 10. The execution time of Genome, normalized to L-Opt.

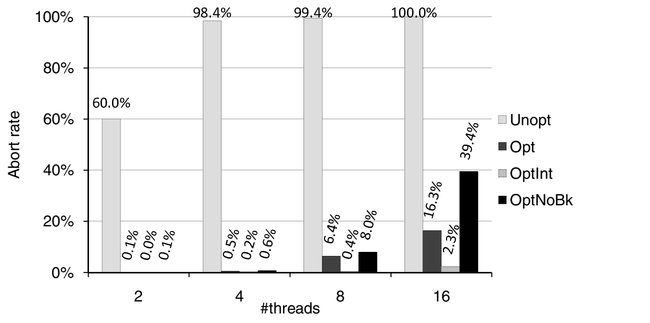


Figure 11. The effect of the optimizations on the abort rate.

object level conflict detection: all array elements are considered as one object with respect to the conflict detection. We changed the implementation of the hashtable to a chaining version and also removed the shared counter, much like the hashtable from the STAMP suite. After these changes Genome's conflict rate was very low and scaled as in the original C version (see Figure 10 Opt).

Friendly fire pathology when rehashing. A second observation was that, when running with 4 or more threads, sometimes the execution was unusually long. Then looking at the number of reexecutions of the individual `atomic` blocks, we observed the *friendly fire* pathology [2]: transactions were aborting one another without any being able to commit. Linking this information with the conflict points we found the underlying reason: one transaction was trying to rehash and at the same time another thread was starting the execution of the same `atomic` block. Then the two transactions were continuously aborting each other. When running with 2 threads it is less likely that the execution of the same `atomic` block will overlap, but with 4 or more threads this probability becomes much higher. Although a better solution could be found, our quick approach was to initialize the hashtable with a larger bucket array.

Initializing the buckets (OptInit). At this point, we examined the conflict data of the application more carefully and noticed that the number of conflicts when adding an element to a hashtable was approximately the same as the number of entries in the hashtable. Almost every addition of a new entry to the hashtable was causing a conflict. The reason for this was that we were initializing the elements in the bucket array at the time of adding the first entry in the bucket and again due to the object granularity conflict detection this was causing other transaction working on the array to abort. Our solution for this problem was to initialize the bucket array with default bucket objects during the initialization phase. This significantly reduced the abort rate (see Figure 11 OptInit) and made the application scale up to 16 threads.

Removing the buckets (OptNoBk). We have also developed a slightly different version of the chaining hashtable which does not have buckets and stores the linked list directly into the buckets array. This approach is slightly faster because it saves one indirection when performing a hashtable operation but has the same even higher contention than Opt. Figure 12 visualizes the implementation differences between the chaining hashtables that we used to optimize Genome.

We can see from Figure 11 that OptInit has smallest abort rate and scales up to 16 threads whereas Opt and OptNoBk scale up to 8 threads and are saturated at 16 threads. OptNoBk is faster because of saving one extra indirection due to the direct pointer in the array and not initializing all the buckets. In Figure 10 we can see that the single threaded execution of Unopt has the best performance but simply the implementation of this hashtable is not TM friendly.

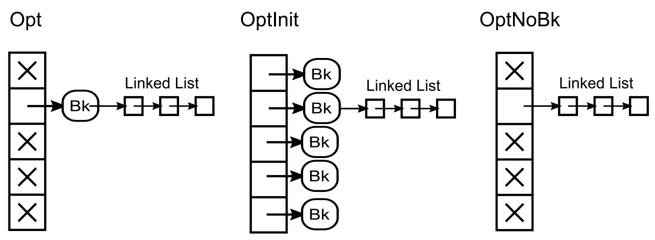


Figure 12. The different variants of the chaining hashtable we used in Genome. *Opt* uses bucket objects and does not initialize the bucket array. *OptInt* is the same as Opt but the bucket array is initialized. *OptNoBk* is a version of Opt that stores linked lists directly on the bucket array.

5. Debug-Time Transaction Management

Our final set of debugger extension features allow the user to manage the transactions under the control of the debugger. At the level of `atomic` blocks, the user can create *debug-time transactions* or *split* `atomic` blocks. These features are intended for investigating errors in the source code, and trying to patch the errors without modifying and recompiling the source code (e.g., when testing out a hypothesis for what is causing a race condition).

Although it might be error prone, drawing analogy from current debuggers' functionality that allow users to modify the program aspects by changing the values of variables in memory or processor registers, we were motivated to implement operations that the user might use to change the state of the transactions such as by adding or removing entries into the transaction's read and write set when debugging at the level of transactions.

5.1 Debug-Time Transactions

A *debug-time transaction* is a new debugger abstraction that helps for the correctness debugging of transactional applications. While debugging, a user may notice that `atomic` blocks are missing in certain places or that `atomic` blocks could be reduced in size. Figure 13 has a contrived example (line 26) where a data race occurs because an `atomic` block is too small. Figure 14 has an example where, instead of defining one large `atomic` block, the program uses two smaller blocks. In such cases, the user can create a debug-time transaction or enlarge the scope of an existing `atomic` block by marking the boundaries of the new `atomic` block on the source code. Thereafter, the debugger ensures that the debug-time transactions are executed atomically, as if regular `atomic` blocks, but without exiting the debug process to change and recompile the source code.

In Figure 15 we show a difficult to find atomicity violation example that we discovered in the QuakeTM [8] source code after a careful inspection. The error manifested in disconnecting the

```
 1  static public void Main(string[] args) {
 2     Thread t1 = new Thread(ThreadEntryIncrement);
 3     Thread t2 = new Thread(ThreadEntryDecrement);
 4
 5     t1.Start();
 6     t2.Start();
 7  }
 8
 9  static void ThreadEntryIncrement() {
10     int temp = 0;
11
12     atomic {
13         temp = counter;
14         temp++;
15         counter = temp;
16     }
17  }
18
19  static void ThreadEntryDecrement() {
20     int temp = 0;
21
22     atomic {
23         temp = counter;
24         temp--;
25     }
26     counter = temp;
27  }
```

Figure 13. An example where the `atomic` block in lines 22-25 is shorter and line 26 must be included in the `atomic` block.

```
initially a = b = 0;

Thread 1                      Thread 2

atomic{
   a++;
}                             atomic {
                                  a++;
atomic{                       }
   b--;
   assert(a + b == 0);
}
```

Figure 14. An example of incorrectly splitting a critical section in two smaller `atomic` blocks. The shown interleaving between thread 1 and thread 2 will result in violating the invariant that a+b=0.

clients from the game session due to bad formatted messages. We checked the functions such as `WriteMulticast` which build these client messages and their definitions were all correctly synchronized. To see how the execution changes, we randomly created and removed temporary `atomic` blocks or coarsened existing ones. Due to the nondeterministic nature of the error, it took us quite long time to constrain the problematic location to the code that interprets Quake extension functions implemented in Quake C and compiled to intermediate representation. If we were able to create, remove and resize `atomic` blocks while debugging, we would find the problematic location easier. In this case we would save a lot of time from changing and recompiling the source code and trying to reproduce the error by re-establishing the client-server game session.

Later, by reverse engineering the Quake extension functions interpreted inside this problematic code, we noticed that there is one function (`FireAxe`) which calls the function `WriteMulticast` several times to build the individual parts of a multicast message. This pattern of use is similar to calling `printf` to print multiple lines on the console. In a serial execution, these functions would execute one after the other and build a correct message. But in multi-threaded execution, although each `WriteMulticast` function is correctly synchronized a possible interleaving with another

```
// Correctly synchronized function
void
WriteMulticast(message) {
  atomic {
    <update message buffer>;
  }
}

    Thread 1                       Thread 2
void FireAxe() {
  WriteMulticast(msg_part1);
                                 WriteCoordinate(coord);
  WriteMulticast(msg_part2);
}
```

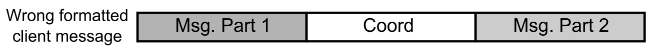


Figure 15. A difficult-to-discover atomicity violation from QuakeTM code. In a serial execution, the two calls to `WriteMulticast` function would be executed one after other and the two parts of the multicast message would be next to each other. To properly synchronize this is necessary to call `FireAxe` method inside an `atomic` block.

thread, like the one shown in the Figure 15, would result in a malformed packet.

The implementation of debug-time transactions, relies on the availability of irrevocable transactions in the STM library. When the user marks the start and the end of the transaction TmDbgExt gets the addresses of the statements using the debugger engine and puts internal breakpoints (see Section 2.3) at these places—one denoting the start and other denoting the end of the transaction. And when the start or end breakpoint is hit, TmDbgExt calls respectively the `StartIrrevocableTransaction` or `CommitIrrevocableTransaction` function from TmTargetDbg by following the method described in Section 2.2.

There is one more subtlety of calling function `StartIrrevocableTransaction`. This method manipulates locks within the STM library and must synchronize with other threads (e.g., if they are also trying to start irrevocable transactions). Consequently, if we call this method by resuming only one thread in the target process and keep the other threads blocked may cause deadlock. Therefore, in this case we resume all target-process threads until the call to `StartIrrevocableTransaction` is complete.

5.2 Splitting Atomic Blocks

To split a large `atomic` block into two smaller ones, we provide the user with two alternatives. In the first alternative, while stepping inside an `atomic` block, the user can split the transaction for one time only at the place where the next statement is to be executed. In the second alternative, the user marks at which statement to split the transaction (see Figure 16). In the former case following the method for calling functions in the target process, described in Section 2.2, we call the `SplitTransaction` function from TmTargetDbg. In the latter case, TmDbgExt creates a internal breakpoint on the location where the transaction is to be split. Whenever any breakpoint is hit, TmDbgExt checks if it is used to split an `atomic` block and if so, the debugger transparently calls the `SplitTransaction` function and continues the execution without breaking into the debugger. In effect, function `SplitTransaction` commits the current transaction and then immediately initiates a new transaction.

One subtlety inherent to our STM implementation that should be considered implementing this feature is where the split point is introduced. The user should be disallowed to split atomic blocks in functions that are not defining the outermost transaction. In this situation, the second part of the transaction (e.g., lines 4-6) may

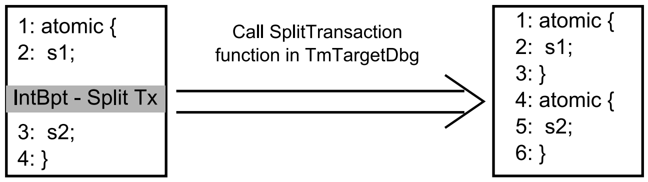


Figure 16. Splitting a transaction. TmDbgExt puts a internal breakpoint (IntBpt) denoting the place where the transaction is to be split. When the breakpoint is hit the debugger transparently calls a function `SplitTransaction` in the target process, creating the effect of committing a transaction and initiating a new one.

not be able to roll back to an interior point (the place where the transaction was split) because the function stack is torn down.

We believe that users who want to optimize their transactional applications by decreasing the size of the coarse grain `atomic` blocks would greatly benefit from this feature. For example, at debug-time users can split the large `atomic` blocks and see how this affects the correctness and the runtime performance.

5.3 Modifying Transactional State

TmDbgExt implements user commands to directly modify the state of the transaction by changing any of its attributes and also adding or removing an entry into the transaction's read and write set while debugging at the level of transactions (see Section 4). All these operations may cause an incorrect execution of the application and it is the user's responsibility to use them reasonably. Adding an entry into the read or write set of a transaction may cause the transaction to become invalid and abort. The debugger extension detects such cases and warns the user by requesting to confirm the action. These operations are implemented by calling the respective functions from Figure 2 which modify the STM state.

6. Related Work

In a parallel work to ours, Herlihy and Lev have developed an infrastructure for debugging transactional applications—tm_db [11]. From a user's perspective, compared to our work, when debugging a transactional application with the abstractions that Herlihy and Lev introduce, it will look like debugging at the level of transactions (Section 4). Their approach has the objective to properly integrate the debugger with the TM implementation. The primary focus of tm_db is to consistently expose the TM state through the debugger without changing the existing debugging conventions. In addition to transaction-level debugging we introduce the notion of debugging at the level of `atomic` blocks, attempting to abstract over whether or not these are implemented with TM. We also propose and implement mechanisms to create debug time transactions, split `atomic` blocks and modify the state of transactions under the control of the debugger. In tm_db Herlihy and Lev introduce important concepts such as logical value, scopes, distinction between transactional reads, writes and their respective conflict coverages. These new concepts abstract the internal organization of different STM systems. Logical values are necessary for preserving the isolation property of transactions when debugging at the level of transactions. Abstracting the reads and writes with their respective coverages hides the internal mechanism to manage the read and write sets and also help in identifying false conflicts. Incorporating these new abstractions into our extension would provide users an uniform view to the TM state when debugging at the level of transactions.

Using their debugging infrastructure, Herlihy and Lev provided support for 8 different TM implementations [15]. To do so, they implemented separate Remote Debugging Module (RDM) systems, one for each library variation, and they extended the STM libraries with support for debugging.

In earlier work, before tm_db, Lev and Moir discussed how the debugger and the TM implantation should by integrated [16]. They surveyed features that a debugger could provide by leveraging the underlying TM system. From their work, we were inspired that seeing the read set and write set of transactions can help to understand the reason for aborts. We extended this idea, and implemented conflict point discovery (Section 4.2) which identifies the program statements that caused transactions to conflict.

Chafi *et al.* have developed a micro architectural extension TAPE [4] for the Transactional Coherence and Consistency [5] system that has HTM support. They used TAPE to profile and optimize transactional applications by studying the locations where transactions conflict much like we do in conflict point discovery but in STM. These two approaches can be combined in a hybrid transactional memory system.

Recent work carried by Gupta *et al.* leveraged the existing infrastructure in a hardware transactional memory system RaceTM [8] to detect data races in multi-threaded applications. The combination of this functionality and debug-time transactions would be a complete tool to find and fix data races in multi-threaded applications at debug time.

7. Conclusion and Future Work

In this paper we have presented three different debugging approaches for transactional applications. Debugging at the level of `atomic` blocks provides users the same experience across different underlying implementation mechanism. The debugger is extended to reflect the atomicity and isolation properties of `atomic` blocks and this makes it easier to debug synchronization problems across different `atomic` blocks and incorrect code within `atomic` blocks. Debugging at the level of transactions assumes that the underlying implementation of `atomic` blocks is TM and exposes their typical attributes such as read and write set. Debugging by following this approach is useful to discover pathological cases that have negative impact on the overall runtime performance. To facilitate the profiling and optimization of `atomic` blocks we have implemented conflict point discovery which tells the exact statements where conflicts happen along with additional contextual information. By using conflict point discovery and examining the state of transaction we iteratively optimized a C# port of the Genome application from the STAMP TM application suite. We introduced mechanisms for adding and removing `atomic` blocks under the control of the debugger which would make debugging synchronization problems such as atomicity violations easier. In our implementation of these features, we followed a general decoupled approach that can be applied to any debugger and TM system.

In future work, we hope to implement asymmetric data race detection, a mechanism that will report the user when a variable is accessed at the same time inside and outside a transaction. We plan to enable this by leveraging the strong atomicity implementation in Bartok-STM [1].

Acknowledgments

We would like to thank Emery Berger for the discussion about concurrency errors in multi-threaded applications, Burton Smith for his comments on the conditional breakpoints, Yossi Lev for his comments on the tm_db and all the reviewers for their useful feedback.

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*, pages 185–196, Feb. 2009.

[2] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proc. 34th international symposium on computer architecture*, pages 81–91, June 2007.

[3] 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*, pages 35–46, September 2008.

[4] H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: A transactional application profiling environment. In *ICS '05: Proc. 19th international conference on supercomputing*, pages 199–208, June 2005.

[5] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA '07: Proc. 13th IEEE international symposium on high performance computer architecture*, pages 97–108, Feb. 2007.

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

[7] V. Gajinov, F. Zyulkyarov, A. Cristal, O. S. Unsal, E. Ayguadé, T. Harris, and M. Valero. QuakeTM: Parallelizing a complex serial application using transactional memory. In *ICS '09: Proc. 23rd international conference on supercomputing*, pages 126–135, June 2009.

[8] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler. Using hardware transactional memory for data race detection. In *IPDPS '09: Proc. 23rd IEEE international parallel and distributed processing symposium*, pages 1–11, may 2009.

[9] T. Harris and K. 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, Oct. 2003.

[10] 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.

[11] M. Herlihy and Y. Lev. tm_db: A generic debugging library for transactional programs. In *PACT '09: Proc. 18th international conference on parallel architectures and compilation techniques*, pages 136–145, Sep 2009.

[12] G. Kestor, S. Stipic, O. S. Unsal, A. Cristal, and M. Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT '09: 4th workshop on transactional computing*, Feb. 2009.

[13] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. 1st edition, Jan. 2007.

[14] Y. Lev. Making debuggers transaction-ready. Transactional Memory: From Implementation to Application, Seminar 2008241, Dagstuhl, Germany, June 2008.

[15] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09: 4th workshop on transactional computing*, Feb. 2009.

[16] Y. Lev and M. Moir. Debugging with transactional memory. In *TRANSACT '06: 1st workshop on transactional computing*, June 2006.

[17] Microsoft Corporation – MSDN. Debugger engine and extension APIs. `http://msdn.microsoft.com/en-us/library/cc267863.aspx`.

[18] Microsoft Corporation – MSDN. Debugging tools for windows. `http://msdn.microsoft.com/en-us/library/cc266321.aspx`.

[19] V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? Results from an empirical study. Technical Report 2009-12, University of Karlsruhe, Sept. 2009.

[20] C. Perfumo, N. Sonmez, S. Stipic, A. Cristal, O. S. 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. 5th international conference on computing frontiers*, pages 67–78, May 2008.

[21] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *PPoPP '10: Proc. 15th ACM SIGPLAN symposium on principles and practice of parallel programming*, Jan. 2010.

[22] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and exploiting inevitability in software transactional memory. In *ICPP '08: Proc. 37th IEEE international conference on parallel processing*, pages 59–66, Oct. 2008.

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

[24] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *SPAA '08: Proc. 20th ACM symposium on parallelism in algorithms and architectures*, pages 265–274, June 2008.

[25] F. Zyulkyarov, S. Cvijic, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. WormBench: A configurable workload for evaluating transactional memory systems. In *MEDEA '08: Proc. 9th workshop on memory performance*, pages 61–68, Oct. 2008.

[26] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic Quake: Using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proc. 14th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 25–34, Feb. 2009.