UNIVERSITAT POLITÈCNICA DE CATALUNYA

Programming, Debugging, Profiling and Optimizing Transactional Memory Applications

PhD Thesis Proposal

PhD Student : Ferad Zyulkyarov
Supervisors : Osman Unsal, Adrian Cristal
Director : Eduard Ayguade

04.06.2010, Barcelona

Abstract

The shift from developing powerful monolithic CPUs to a less powerful but multi-core CPUs made developers to rethink their approach of writing programs. Programmers cannot anymore expect that their programs will execute faster on the next generation CPUs unless their programs are parallel. For many years, researchers have been seeking for various solutions to make parallel programming for shared memory architectures easier and also efficient. Transactional Memory (TM) is one such potential solution.

In TM synchronizing access to shared data is simpler than locks. The programmer defines the critical sections using atomic blocks and the underlying TM implementation automatically executes the enclosed instructions atomically and in isolation. In contrast, when using locks, the programmer manually implements the atomicity and isolation for the shared data. In addition, when conflicts are rare, the speculative execution of atomic blocks promises to deliver performance which is comparable to efficient lock-based implementations.

To answer the questions “Is programming applications using atomic blocks easier than locks?” and “Is the performance of TM competitive with locks?” we have developed a real TM application – Atomic Quake. To implement Atomic Quake, as a base we used a parallel lock-based Quake game server and replaced all lock-based critical sections with atomic blocks. We have found out that developing applications with atomic blocks would be easier than locks but the performance of STMs should be improved. In addition, our experience of developing Atomic Quake revealed unsought problems which showed that TM is not ready for use in production quality software. First, migration from locks to TM would not be trivial because of the non-block structured critical section defined by locks. Second, the TM did not provide mechanisms to handle and recover from errors that can arise inside transactions and even worse TM did not have defined semantics for such situations. Third, the use of non-revocable operations such as I/O inside atomic blocks was difficult to eliminate and sometimes impossible. Forth, it was difficult to debug errors and almost impossible to profile the TM relevant bottlenecks.

Among the different aspects of TM, debugging and profiling were not being studied in depth. Therefore our research focused on investigating how to extend current debuggers to debug TM applications and finding profiling techniques that would reveal the bottlenecks in the TM applications.

We have introduced three new approaches to debug TM applications. First, the user can debug at the level of atomic blocks. In this approach, an atomic block is treated as a single instruction and the implementation details of the atomic blocks, weather TM or lock inference, are hidden to the user. Second, the user can debug at the level of transactions. In this approach, the implementation of atomic blocks is assumed to be TM and the user can step inside atomic blocks and examine the TM state. Third, the user can manage the TM state at debug time which is analogues to the mechanisms how one can change the CPU state. Also, we have introduced new abstractions such as debug time atomic blocks and TM watch points. Debug time atomic blocks let the user create and remove atomic blocks at debug time. We have implemented our ideas in an extension for WinDbg debugger and the ahead-of-time C# to x86 Bartok compiler.

To profile TM applications we have introduced new techniques that provide in-depth and comprehensive information about the wasted work caused by aborting transactions. We have explored three directions: (i) techniques to identify multiple conflicts from a single program run, (ii) techniques to describe the data structures involved in conflicts by using a symbolic path through the heap, rather than a machine address, and (iii) visualization techniques to summarize which transactions conflict most. To demonstrate the effectiveness of these techniques we have built a standalone profiling tool and a profiling framework for the Bartok compiler. Then using our tool we have profiled and optimized several applications from the STAMP benchmark suite.

Contents

1. Objectives

The objectives of this research are:

1. Answer the question “Is programming multi-threaded programs using atomic blocks and TM easier than locks?”
2. Answer the question “Is the performance of TM competitive to locks?”
3. Answer the question “Is TM mature enough to develop production quality software? If no, what is missing?”
4. Study how to extend current debuggers to support atomic blocks and TM. We have found that current debuggers are not aware of atomic blocks and TM. Therefore debugging TM programs is very difficult and a frustrating task.
5. Find proper profiling techniques that will show the program bottlenecks caused by the TM programming model (but not TM implementation). We and other researchers have found that TM applications have bottlenecks specific to the way of using atomic blocks (TM programming model). Finding these bottlenecks is difficult because there are no profilers to analyze the TM aspects of the program and the implementation of the TM can be closed.
6. Study feedback directed compilation techniques for transactional memory applications.

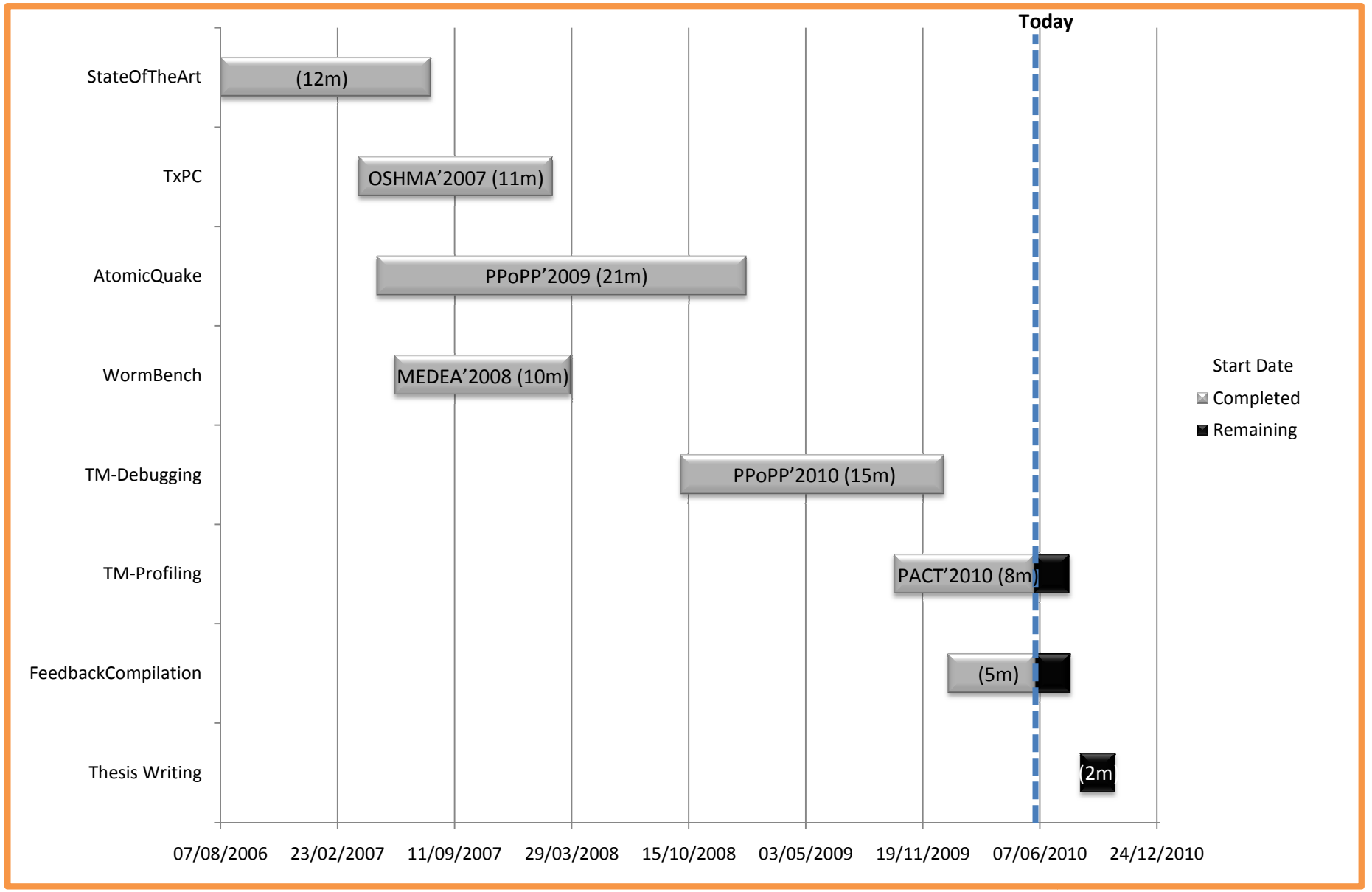
2. Current State and Work Plan

The work described in this document has already been completed.

1. To accomplish the objectives (1), (2), (3) we have developed a real complex transactional memory application – AtomicQuke. The work was published as a full paper at PPoPP'2009.
2. To accomplish the objective (4) we have developed a debugger extension for WinDbg debugger. The work was published as a full paper at PPoPP'2010.
3. To accomplish the objective (5) we have developed a standalone profiling tool and low overhead profiling framework. The work will be published as a full paper at PACT'2010.

Things to be completed:

1. I need to prepare a camera ready version for the PACT'2010 paper.
2. I have to write a paper about feedback directed compilation for the transactional memory applications and submit it to ASPLOS'2011.
3. After the ASPLOS submission I will start writing my PhD thesis.


Today
StateOfTheArt
(12m)
TxPC
OSHMA'2007 (11m)
AtomicQuake
PPoPP'2009 (21m)
WormBench
MEDEA'2008 (10m)
TM-Debugging
PPoPP'2010 (15m)
TM-Profiling
PACT'2010 (8m)
FeedbackCompilation
(5m)
Thesis Writing
(2m)
Start Date
Completed
Remaining
07/08/2006
23/02/2007
11/09/2007
29/03/2008
15/10/2008
03/05/2009
19/11/2009
07/06/2010
24/12/2010

3. Publications

1. **Ferad Zyulkyarov**, Milos Milovanovic, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, Mateo Valero, Memory Management for Transaction Processing Core in Heterogeneous Chip-Multiprocessors, OSHMA '09: Workshop on Operating System support for Heterogeneous Multicore Architectures, September 2007
2. **Ferad Zyulkyarov**, Sanja Cvijic,Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, Mateo Valero, WormBench - A Configurable Workload for Evaluating Transactional Memory Systems, MEDEA '09: Workshop on MEmory performance: DEaling with Applications, systems and architecture, October 2008
3. **Ferad Zyulkyarov**, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, Mateo Valero, Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server , PPoPP'09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009
4. **Ferad Zyulkyarov**, Tim Harris, Osman Unsal, Adrian Cristal, Mateo Valero, Debugging Programs that use Atomic Blocks and Transactional Memory, PPoPP'10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, January 2010
5. **Ferad Zyulkyarov**, Srdan Stipic, Tim Harris, Osman Unsal, Adrian Cristal, Mateo Valero, Ibrahim Hur, Discovering and Understanding Performance Bottlenecks in Transactional Applications, to appear at PACT'10: Proc. 19th International Conference on Parallel Architectures and Compilation Techniques, September 2010

4. Background – Transactional Memory

Transactional memory is an optimistic concurrency control mechanism. Typically, the language level support of transactional memory is expressed via the "atomic block" statement (see Figure 1). The programmer uses atomic blocks to identify the operations that should execute atomically and in isolation with respect to other atomic blocks.

```
atomic {
  statement1;
  statement2;
 ..
}
```

Figure 1. Example atomic blocks section.

Compared to locks, transactional memory is promising to be easier to develop parallel programs and still deliver performance comparable to efficient lock-based implementations when the conflict rate is low. The programmer is not required to know the shared data structures because the underlying implementation of the TM system automatically implements atomicity over them. On the other side, when using locks, the programmer should identify the shared data structures, associate locks with them and then manually implement atomicity. In addition, TM is deadlock free and using atomic blocks is a composable way of implementing concurrency control.

Because locks are pessimistic approach for implementing concurrency control, they may limit the parallelism in the program execution. Consider for example the code segment from Figure 2.

```
acquire(lock) {                          atomic {
    hashtable.Insert(obj);                  hashtable.Insert(obj);
}                                         }
        (a)                                        (b)
```

Figure 2. Synchronizing access to a hashtable using a) lock, b) atomic block.

Typically, insertions to the same bucket in a hashtable are rare and most of the insert operations can proceed in parallel. However, if we synchronize the insert operation with a lock as shown in Figure 2 the execution will serialize at this point and block on the lock. On the other side, using atomic blocks and transactional memory would allow the parallel execution of the atomic blocks. If threads try to insert to the same bucket, the TM will detect conflict and re-execute.

Of course, the advantages of transactional memory do not come for free. The cost is the overhead that TM implementations have. The overhead that TMs have is due to the implicit memory versioning which is necessary to detect conflict. The overhead varies between different implementations but typically, software transactional memories (STM) are slower but flexible whereas hardware transactional memories (HTM) are faster but limited. Hybrid (HyTM) approaches are proposed to either speedup STMs or make HTMs unbounded (1).

5. Atomic Quake

Atomic Quake (2) is a real transactional memory application. It is derived from a parallel lock-based version of the Quake game server by replacing all lock-based synchronization with atomic blocks. The motivation behind this work is:

1. Answer the question “Is programming parallel applications using TM easier than locks?”
2. Answer the question “Is the performance of TM comparable to efficient lock implementations?”
3. Answer the question “Is TM mature enough to be used for production software? If not, what is necessary?”
4. Deliver a real transactional memory application to drive the research in TM.

Indeed, we could accomplish the goals that we have set.

5.1. Is Programming with TM Easy?

From our experience we found that programming with TM is easier than locks. The most important advantage of TM over locks is that the programmer is not required to find all shared data structures, associate locks with them and care about avoiding deadlocks.

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

Figure 3. Synchronizing multiple objects using locks and atomic blocks.

For example Figure 3 shows a code fragment from AtomicQuake which implements the logic for picking an object. In this code there are two distinctive parts that are not relevant to the logic at all but implement the concurrency – “lock phase” and “unlock phase”. During the lock phase we first check the type of the object and then acquire the lock associated with this code. The unlock phase is opposite. When the number of shared data structures as in AtomicQuake is large, implementing such synchronization might be very difficult task and require the application of special locking conventions to avoid deadlocks. In fact, the actual problem appears when it comes time to maintain such code. Because the association between the locks and shared data is not in the program but in

the head of the programmer who wrote the code maintaining such code might be difficult accompanied with the writing of heavy documentations.

A more complicated example from AtomicQuake is region based locking. When player moves from location X to location Y, it interacts with various objects (e.g. other players). These objects are synchronized with fine grain locking by locking only specific regions on the map. To identify exactly which locks to acquire, the game server simulates the action of the player and it finds the regions that it might interact with. In TM, because of the implicit versioning, such simulation wouldn't be necessary and only the accessed memory references will be tested for conflicts.

5.2. Is TM Competitive to Lock?

We have compiled AtomicQuake with a prototype version of the Intel C/C++ compiler which have language level support for STM. Our results showed that for the moment the performance of STM is not competitive to locks. AtomicQuake had x4-x5 single thread overhead and its scalability saturated at 4 threads (see Figure).

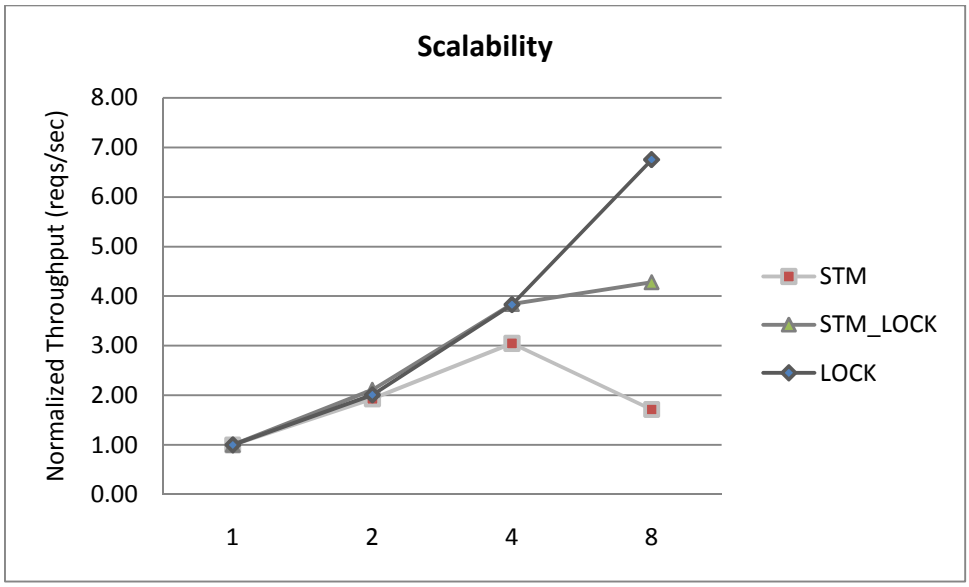


Figure 4. Scalability of AtomicQuake. In this graph each experiment (STM, STM_LOCK and LOCK) is normalized to itself. STM critical sections are atomic blocks. STM_LOCK critical sections are atomic blocks, but before entering the atomic blocks a global lock is acquired. LOCK – each critical section is protected with a global lock.

We have performed three different experiments:

1. STM – all critical sections are defined by atomic blocks. This experiment is used to measure the performance of the underlying STM implementation.
2. STM_LOCK – all critical sections are defined by atomic blocks and also a global lock. Before entering an atomic block a global lock is acquired. This experiment is used to see the overhead of the STM excluding aborts.
3. LOCK – each critical section is defined by a global lock. This experiment is used to check if our base implementation scales.

The results from STM and STM_LOCK experiments show that the STM instrumentation overhead causes the application to not scale well beyond 4 threads. From here we conclude that the STM

instrumentation affects also scalability. In addition we have found out that the number of aborts drastically increases from 4 to 8 threads causing the performance of STM to degrade (see

Threads	Transactions	Aborts		Irrevocable
		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

Figure 5). We guess that one reason for this to happen is that we were restricted to use a small game map. In such case, players interact most of the time close to each other and cause conflicts. Probably when using larger maps the conflict rate would be lower. Lupei et al. (3) have independently developed their version of the transactional memory Quake application. They stripped the network communication component from the server and applied various optimizations based on the usage of the data structures. In their experiments STM performs comparably to the lock-based implementation.

Threads	Transactions	Aborts		Irrevocable
		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

Figure 5. General transactional characteristics.

Because of the lack of proper profiling tools we could not analyze and understand the reason for the poor performance. However, this experience was motivation for our next work of studying profiling techniques for transactional memory applications.

5.3. Is TM Mature?

After our experience of developing real transactional memory applications we have realized that TM is not ready for use in production software. The reasons we think so are:

1. Richer language primitives for TM are required.
2. The TM semantics are to be defined.
3. Software development tools such as debuggers and profilers should be aware of atomic blocks and TM programming model.

Although replacing lock-based synchronization with atomic blocks seems to be straightforward we found that it is not. Large applications such as AtomicQuake have many shared data structures which are synchronized with complex locking schemes (e.g. region based locking). It took us significant effort to understand the shared data and how it is synchronized. Also, the non-block structured use of lock and unlock operations (see Figure 6) required us to carefully examine the code and restructure it so that we could use atomic blocks. This suggests that using automatic tools for conversion from locks to atomic blocks may not be viable. However, having a mechanism to explicitly commit a transaction just like to explicitly aborting it would make transactifying lock-based applications easier.

Other challenges that we faced are I/O and handing and recovering from errors inside transactions. In some cases we could not eliminate I/O and we relied on fall back mechanisms such as irrevocable transactions. Besides, our TM system did not support any mechanisms for handling and recovering from errors inside transactions. Even more, the TM implementation didn't have defined semantics for the cases when error happens in a transaction. For the purpose of running the application, we had to define our default semantics i.e. to commit the transaction and deal with the error in a non-transactional code. The code fragment in Figure 7 shows a case where an error that happens inside a critical section is handled differently based on its type. To implement such logic using TM is necessary that the TM semantics precisely defines how the execution should proceed and also relevant language extensions to expose these semantics to the programmer.

```
 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 }
```

Figure 6. Non-block structured use of lock and unlock operations.

```
 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 }
```

Figure 7. Example error handling inside a critical section.

5.4. Atomic Quake – TM Workload

AtomicQuake is a real transactional memory application. It has various transactional memory characteristics and rich TM programming idioms (see Figure 8 and Figure 9Figure 7). In particular AtomicQuake has:

1. I/O, error handling, library calls, system calls inside transactions
2. nested transaction
3. has privatization and publication idioms
4. has use of failure atomicity
5. long and short running transactions
6. atomic blocks that execute frequently and infrequently
7. atomic blocks that abort a lot and abort less
8. read-only transactions
9. transactions with different read and write set sizes.

These features make AtomicQuake a suitable workload for testing all aspects of TM system including language level support, semantics, runtime, OS, and TM implementations.

Threads	Read Set (bytes)					Write Set (bytes)				
	Min	Avg	Max	Total	%	Min	Avg	Max	Total	%
1	8	478	53,566	17,515,419	83%	0	95	11,161	3,492,370	17%
2	4	540	80,404	40,032,868	82%	0	116	16,861	8,570,992	18%
4	4	575	181,740	95,505,459	81%	0	131	52,032	21,737,915	19%
8	4	799	1,591,946	381,290,019	81%	0	184	352,640	87,837,969	19%

Figure 8. Read and write set characteristics.

ID	TX#	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	172,872,572	288	112,832	6,412	1,328,536	20	104	49	0	0	0	0
60	5,931	5,810,152	224	41,552	980	76,212	12	640	13	928	0	116	0
61	1,095	20,573,540	4,560	49,984	19,208	723,474	88	776	661	90	84	84	84
59	1,042	3,117,844	1,520	39,344	2,999	29,176	5	28	28	16,672	16	16	16
57	1,038	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	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	67,660	720	48,176	1,735	96	32	32	32	18	6	6	6
5	2	99,988	592	36,384	1,923	64	32	32	32	10	5	5	5
22	2	43,632	12,176	35,504	21,816	72	36	36	36	128	64	64	64
36	2	40,476	6,800	44,880	20,238	249	108	141	125	55	22	33	28
38	2	71,368	2,144	31,504	4,461	90	44	46	45	26	12	14	13

Figure 9. Statistics for each atomic block from a single-threaded execution. Transactions that are not executed are not shown..

5.5. Related Work

There are several transactional memory applications used to evaluate the performance of TM implementation – STAMP (4), StmBench7 (5), WormBench (6), RMS-TM (7).

STAMP is benchmark which consists of 8 kernels from different domains. STAMP applications are implemented by manually embedding calls to the STM library. STAMP applications have various transactional characteristics and are good for evaluating the performance of TM implementations. However, stamp applications are small and do not have nested transactions, I/O, and error handling.

StmBench7 is a synthetic benchmark. It has long transactions and as its name implies it is designated to evaluate and stress the limits of STM systems.

WormBench is developed by us and is also a synthetic workload like StmBench7. However, WormBench is highly configurable and can model events that occur rarely to reproduce bugs or model real applications.

RMS-TM is a set of 4 applications from the recognition, mining and synthesis domain. Like AtomicQuake, RMS-TM is a transactified version of existing parallel lock-based applications. These applications have very small critical sections and 3 of them spend very little time executing the critical sections. Applications scale well and show the opposite of AtomicQuake, that the performance of TM can be comparable to locks. The reason for it might be that critical sections in AtomicQuake are quite large.

Lupei et al. (3) have independently developed another transactional memory Quake application. In their version they have removed the client component and used a prerecorded packet stream to load the server. In addition, they have optimized the code based on the way how data structures are accessed. Their experiments show that STM has performed comparably to the lock-based implementation.

Pankratius et al. (8) and Rossbach et al. (9) have conveyed independent studies to understand the effort of developing TM applications. The results of these studies are aligned with our findings that developing applications with TM is easier than locks. In addition, Pankratius et al. observed that debugging and maintaining the TM applications is easier than locks.

6. Debugging Support for TM

When developing AtomicQuake we had a lot of problems trying to debug errors in the code. The reasons were that debuggers are not aware of atomic blocks and transactional memory. This motivated us to study how to extend current debuggers with support for atomic blocks and transactional memory.

Until know we have used atomic blocks and transactional memory interchangeably. However from now on, when we talk about debugging we will distinguish between them. Atomic blocks are language constructs that have the semantics of atomicity and isolation. The atomicity semantics guarantee that all statements within the atomic blocks execute all or none (see Figure 1). The isolation semantics guarantee that other threads cannot observe the speculative changes during the execution of an atomic block. Transactional memory is one (optimistic) way to implement atomic blocks. Atomic blocks can also be implemented using lock inference or even a global lock. For example, when entering an atomic block, the underlying implementation would acquire a global lock and when exiting the atomic block the underlying implementation will release the lock. We refer to a transaction as an instance (particular execution) of an atomic block.

Based on our distinction between atomic blocks and transactional memory, we introduced three approaches for debugging applications implemented with atomic blocks and transactional memory:

1. debugging at the level of atomic blocks;
2. debugging at the level of transactions;
3. managing transactions at debug time.

When *debugging at the level of atomic blocks*, the debugger does not step inside the atomic block but instead treats the atomic block as if it is a single instruction. This is very useful when debugging synchronization errors. This approach abstracts the underlying implementation of atomic blocks whether transactional memory or a global lock and extends the debugger with the atomicity semantics of atomic blocks.

But one may ask the question “What if my code inside the atomic block is wrong and I want to debug it?” Debugging a code inside an atomic block is also possible. The user has to just put a breakpoint at a statement inside the atomic block. When the breakpoint fires, the debugger breaks and the user can step through the statements inside the atomic block. However, if atomic blocks are implemented with TM, it might be possible that the transaction is invalid but it is not aborted. While a transaction is in invalid state the user can observe incorrect memory values and be confused that the code is wrong. Until the transaction aborts and restarts, the user can be debugging code that actually does not make sense. Moreover, sudden jumps because of aborts, may further frustrate the user. Therefore, to debug wrong code inside atomic blocks, debuggers should be extended with the isolation semantics of atomic blocks. We implement this by switching the transaction into irrevocable mode.

When *debugging at the level of transactions*, we assume that atomic blocks are implemented with transactional memory. When debugging in this mode, the user can observe the transactional memory state – such as read/write set, transaction status, nesting, etc. To debug pathological cases we have introduced new debugging abstractions such as TM breakpoints and also event filtering. To suspend program execution when a specific TM event such as start, conflict or commit, the user can

create TM breakpoints. Because such events can happen too many times and only very few of them might be of user's interest, the user can specify filters for these events (e.g. break when specific address is involved in a conflict).

Analogous to changing the CPU state while debugging, we let the user to also change the transactional memory state. For example, the user can add or remove entries from the read and write set of the transactions. Also, to allow changes in higher level at the level of atomic blocks, we have introduced a new debugging abstraction – debug time atomic blocks. Using debug time atomic blocks the user can create atomic blocks at debug time without exiting the debug process, changing the code and recompiling. This abstraction can be very useful when trying to find synchronization errors such as atomicity violations and data races and patching them. For example, such feature would save us quite long time when trying to find a violation error in QuakeTM (10).

We have implemented our ideas in a debugger extension for WinDbg and binaries compiled with the ahead of time C# to x86 Bartok compiler. We believe that our design (see Figure 10) is general enough and can be applied for other debuggers and STM and HTM implementations.

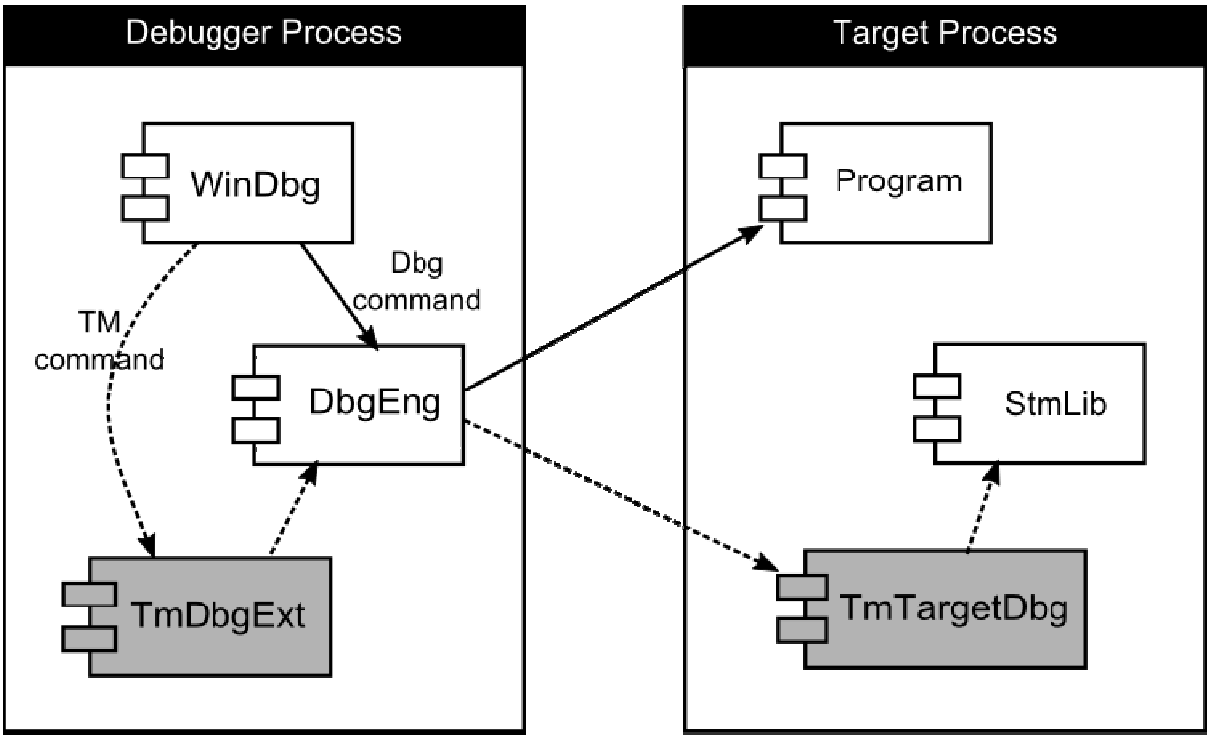


Figure 10. The design of the debugger and debugger extension to support debugging applications that use atomic blocks and transactional memory. The parts in gray are our extensions.

6.1. Related Work

Lev et al. (11) have proposed various ideas that can be useful for debugging transactional memory applications. All these ideas do not distinguish between atomic blocks and transactional memory and assume that atomic blocks are implemented with transactional memory. On a later research, concurrently with us, Maurice and Lev (12) have developed a debugging framework for transactional memory application. The debugging framework provided functionality similar to the one we described when the user debugs at the level of transactions.

Gupta et al. (13) have developed RaceTM. RaceTM has extensions on conventional HTMs and provides race detection in parallel programs for low overhead.

7. Profiling Techniques for TM Applications

After we have developed AtomicQuake we had many questions about its performance (see Figure 4 and Figure 5). However, we could not answer neither of these question because there were not available profiling tools that give in-depth comprehensive information about the bottlenecks that stem from the TM programming model. Moreover, the STM that we used was closed system and we could not understand if the poor performance is because of the STM implementation or our implementation of AtomicQuake. This motivated us to study new profiling techniques that will help the programmer to understand the various program bottlenecks caused by the TM programming model.

Initially we have developed conflict point discovery (14). Conflict point discovery shows the statements in the source code that are involved in a conflict. After, using our implementation of conflict point discovery to profile real applications we have realized that it has several limitations. In a subsequent work, we have extended our profiling idea and introduced a set of profiling techniques for TM applications. We have explored three directions: (i) techniques to identify multiple conflicts from a single program run, (ii) techniques to describe the data structures involved in conflicts by using a symbolic path through the heap, rather than a machine address, and (iii) visualization techniques to summarize which transactions conflict most.

We implemented our profiling techniques in a standalone application and a profiling framework for Bartok-STM. Our design principle for the profiling framework was to have minimal probe effect and execution overhead. Therefore, in the profiling framework only sampled runtime data which was processed offline by the profiling tool and whenever possible we combined the sampling with garbage collections.

The focus of our work is to abstract the overheads that are inherent to the TM implementation and report the overheads at the program level that are relevant to the TM programming model.

7.1. Conflict Point Discovery

Conflict point discovery is a profiling technique that identifies the statements from the source code which have been involved in a conflict. Initially, our basic implementation had a counter associated with each conflicting statement (14). The higher value of the counter indicated higher severity. Also, our first implementation accounted for only the first statement which was involved in a conflict.

After we used this technique to profile the applications from the STAMP benchmark suite we have realized several limitations. First, higher number of aborts did not necessarily mean more wasted work. For example, consider an atomic block T1 which updates a shared counter and aborts 10 times, and atomic block T2 which performs many complicated operations and aborts 9 times. In this case, T1 will be wrongly shown that is more important than T2. Second, it is possible that a transaction might have more entries in its read or write set that are invalid. However, they will not be reported and probably show-up during the subsequent profiling steps (see Figure 11). Third, without contextual information it is difficult or impossible to find the critical path in the program. For example consider Figure 12. In this example code two threads call functions which increment a shared counter with different probability. Basic conflict point discovery will only report that all conflicts happen in function `increment`. However without knowing which function calls most, the

user cannot find and optimize the critical path. In this example the critical path would be `probability80 – increment`.

```
    Thread 1                          Thread 2
1: atomic {                          atomic {
2:    obj1.x = t1;                      ...
3:    obj2.x = t2;                      ...
4:    obj3.x = t3;                      ...
5:    ...                               obj1.x = t1;
6:    ...                               obj2.x = t2;
7:    ...                               obj3.x = t3;
8: }                                 }
```

Figure 11. Basic conflict point discovery would only display the first statements where conflicts happen. On the given examples these statements are line 2 for Thread 1 and line 5 for Thread 2. However, the other statements are also conflicting.

```
increment() {
   counter++;
}
100;

probability80() {
   ...
   probability = random() % 100;

   if (probability < 80)
      atomic {
         increment();
      }
   ...
}

probability20() {
   ...
   probability = random() %

   if (probability >= 80) {
      atomic {
         increment();
      }
   ...
}
/
Thread 1
--------
for (int i < 0; i < 100; i++) {
    probability80();
    probability20();
}

Thread 2
--------
for (int i < 0; i < 100; i++) {
    probability80();
    probability20();
```

Figure 12. Identifying critical path.

To address these limitations, we have extended our work by

1. reporting the importance of the aborts in terms of wasted work (time) but not absolute number of aborts;
2. detecting all conflicting statements but not just the first conflicting statement;
3. associate a wasted work to the function in the call stack at the moment when conflict is detected.

In our tool this information is displayed in a top-down and bottom-up trees (see Figure 13).

To find which statements (memory accesses) are involved in conflict we have made an extension in our STM library to log the return address of the STM function. The implementation of our extension is shown in Figure 14.

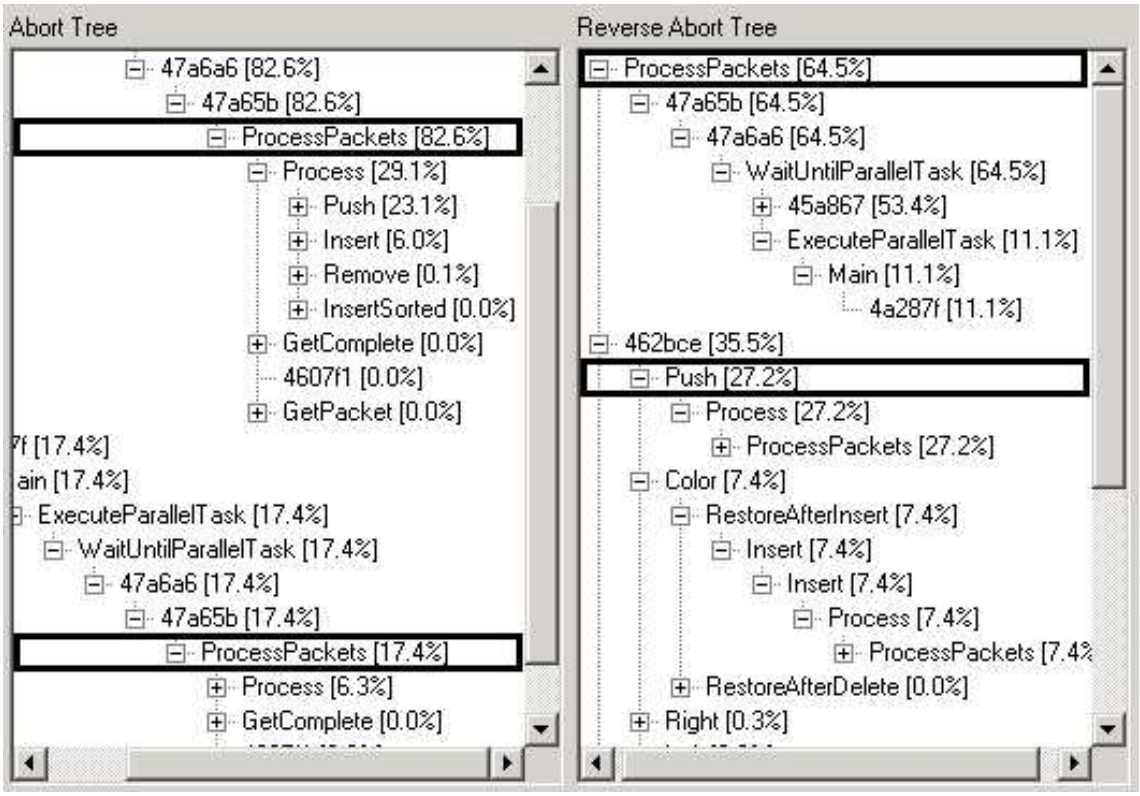


Figure 13. Top-down and bottom-up aborts tree.

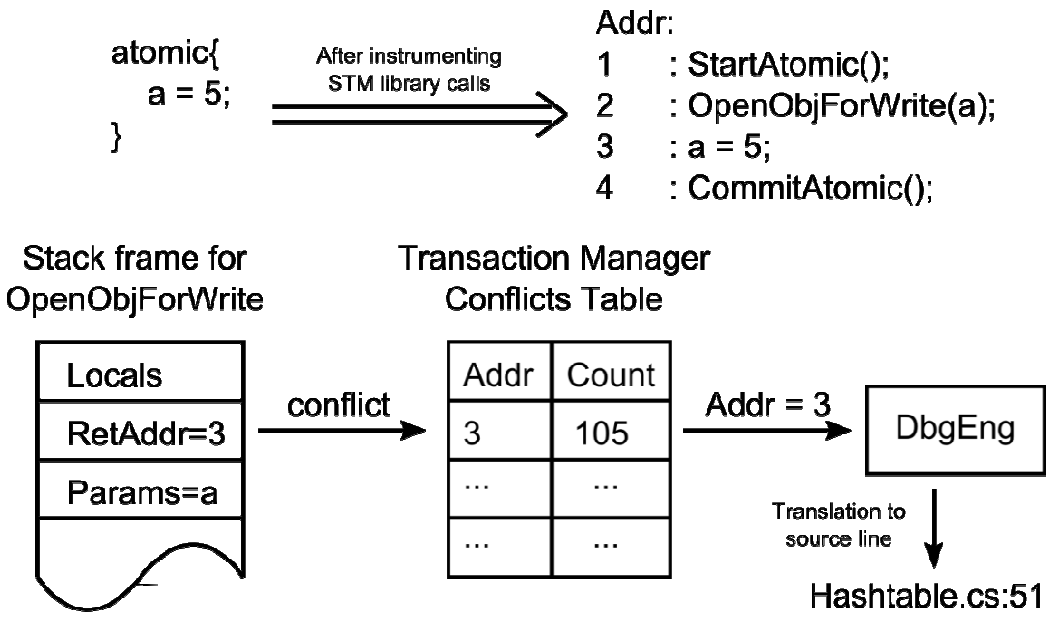


Figure 14. Logging the return address of the STM function for conflict point discovery.

7.2. Identifying Conflicting Objects

Besides reporting the conflicting statements within the code, we also implement a mechanism for reporting the objects that are involved in a conflict. Identifying static variables is trivial however identifying objects in the heap which lifetime is dynamic is challenging. In our approach we reuse the already existing infrastructure of the Bartok memory allocator and garbage collector to map a heap object to a static variable (see Figure 15). The results for the conflicting objects are reported in a tree based window shown in Figure 16.

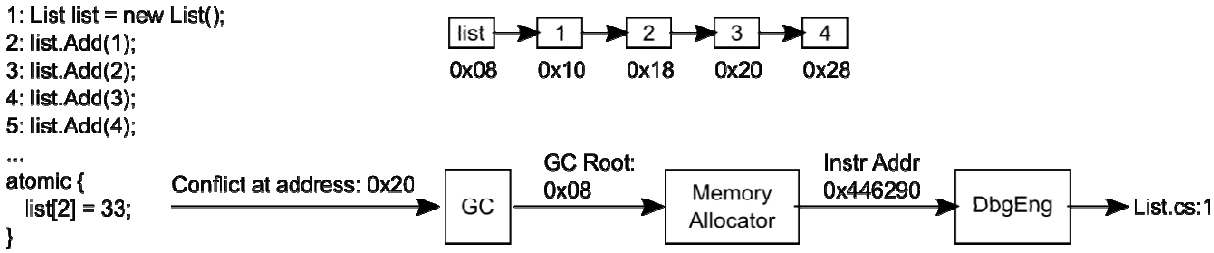


Figure 15. Identifying conflicting objects using the Bartok memory manager and garbage collector.

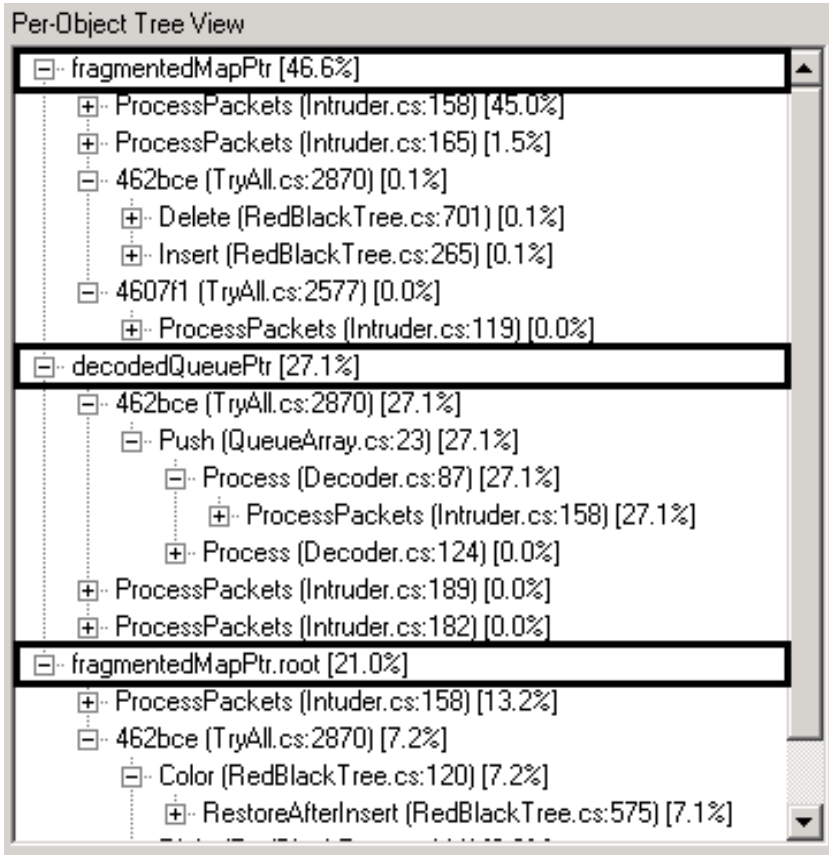


Figure 16. Per-object aborts tree.

7.3. Aborts Graph

Aborts graph shows the abort relationship between different atomic blocks. For example, consider the Bayes application from STAMP, which have 15 atomic blocks and only one aborts most and causes 63% of wasted work. To understand why this atomic block aborts you would need to find the other atomic blocks that cause it to abort. For this purpose we build a conflict graph which shows the abort relationship between the atomic blocks (see Figure 17).

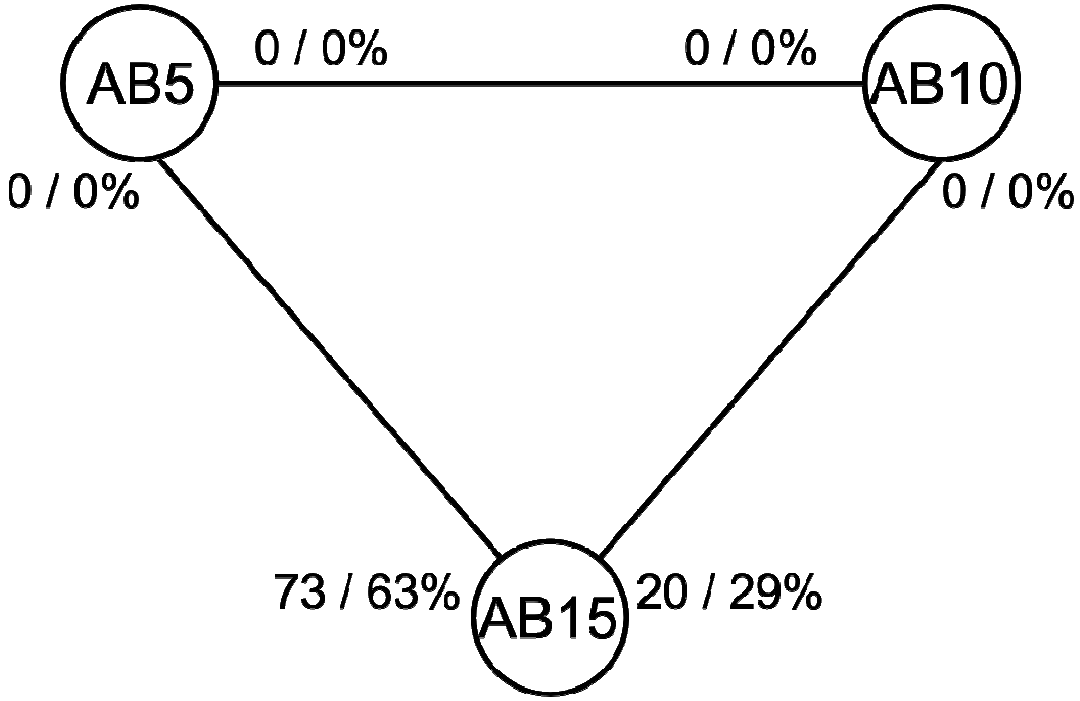


Figure 17. Aborts graph.

Figure 17 shows a part of the aborts graph drawn from the 4 threaded execution of non-optimized Bayes application. In this graph, AB5 is the atomic block which executes Insert operation, AB10 is the atomic which inserts the Reverse operation and AB15 is the atomic block which executes function `FindBestTaskArg`. 73% of AB15's aborts were caused by AB5 amounting to 63% of wasted work and 20% of the AB15's aborts were caused by AB10 amounting to 29% of wasted work. AB5 and AB10 did not abort.

7.4. Visualizing Transactions

We have built a tool that plots the execution of the transaction on a timeline (see Figure 18). In the view pane the transactions start from the left and progress to the right. Successfully committed transactions are colored in black and aborted transactions are colored in gray. The places where a color is missing means that no transaction has been running. The view in Figure 18 plots the execution of the Genome application from STAMP. From this view we can easily identify the phases where transactions abort. Most aborts occur during the first phase of the application when repeated gene segments are filtered by inserting them in a hashtable and during the last phase when building the gene sequence.

The transaction visualizer provides a high level view of the performance. It is particularly useful at the first stage of the performance analysis when the user identifies the hypothetical bottlenecks and then analyzes each hypothesis thoroughly. Another important application of the transaction visualizer is to identify different phases of the program execution (e.g. regions with heavily aborting transactions).

To obtain information at a finer or coarser granularity, the user can respectively zoom in or zoom out. Clicking at a particular point on the black or gray line displays relevant information about the specific transaction that is under the cursor. The information includes: read set size, write set size, atomic block id, and if the transaction is gray (i.e. aborted) it displays information about the abort. By selecting a specific region within the view pane, the tool automatically generates and displays summarized statistics only for the selected region.

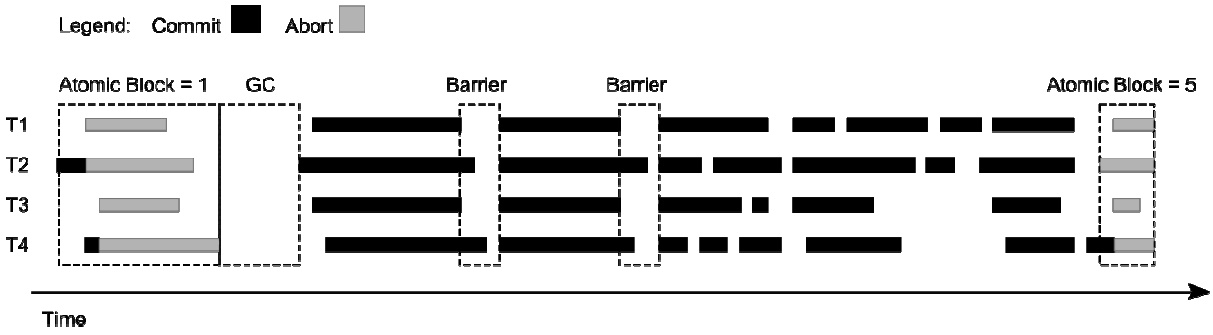


Figure 18. The transaction visualizer plots the execution of Genome with 4 threads.

7.5. Profiling Framework

Trd#	Bayes+	Bayes-	Gen+	Gen-	Intrd+	Intrdr-	Labr+	Labr-	Vac+	Vac-	WB+	WB-
2	4.39	4.69	0.09	0.10	3.69	3.51	0.19	0.15	0.80	0.80	0.00	0.00
4	16.29	27.31	0.29	0.50	14.90	13.65	0.35	0.36	2.30	2.45	0.00	0.00
8	53.74	66.08	0.50	0.82	39.64	37.41	0.40	0.47	4.91	5.30	0.02	0.02

Figure 19. Abort rate with profiling enabled and disabled.

Trd#	Bayes+	Bayes-	Gen+	Gen-	Intrd+	Intrdr-	Labr+	Labr-	Vac+	Vac-	WB+	WB-
1	1.59	1.00	1.28	1.00	1.29	1.00	1.07	1.00	1.26	1.00	0.71	1.00
2	1.00	0.56	0.92	0.65	0.97	0.58	0.64	0.61	0.83	0.59	0.60	0.55
4	0.23	0.23	0.91	0.50	0.91	0.36	0.45	0.46	0.58	0.40	0.41	0.33
8	0.21	0.20	0.72	0.50	1.57	0.38	0.72	0.56	0.53	0.34	0.33	0.22

Figure 20. Overhead of the profiling framework.

We have implemented our profiling framework using Bartok-STM (15). Bartok-STM updates memory locations in-place by logging the original value for rollback in case a conflict occurs. It detects conflicts at object granularity, eagerly for write operations and lazily for read operations. Although our particular choice, the runtime data that we collect is typical for most TMs and can be obtained from other STMs and HTMs.

The main design principle that we followed when building our profiling framework was to keep the overheads and probe effect at minimum. We sample runtime data only when a transaction starts, commits or aborts. For every transaction we log the CPU timestamp counter and the read and write set sizes. For aborted transactions we also log the address of the conflicting objects, the instructions

where these objects were accessed, the call stack of aborting thread and the atomic block id of the transactions that win the conflict. We process the sampled data offline or during garbage collection.

We have evaluated the probe effect and the overhead of our profiling framework on several applications from STAMP and WormBench (Figure 19 and Figure 20). Results suggest that, in applications where transaction execution is comparably longer than the data collection and the aborts rare like in STAMP and WormBench, the overheads and probe effect is marginal. Because we use thread private storage to collect the runtime data, our profiling framework is not a bottleneck for the applications' scalability. On the other hand, in applications which have very short running transactions, overheads can be higher as they are not amortized by the regular program execution. Also, the additional logging on abort has the effect of contention regulator because it prevents transactions from restarting immediately. As a side effect, this may reduce the abort rate in applications with high contention.

7.6. Use Cases

To demonstrate the effectiveness of our profiling techniques, we have profiled several applications from the STAMP benchmark suite.

In Bayes, we have seen that there are many aborts due to the object granularity of conflict detection. After small changes in the code which removed an encapsulating object we could make the application scale as reported by the authors of STAMP.

In Intruder, we have seen that many conflicts happen in red black tree data structure which is used as a base for a map data structure. We have changed the red black tree with a chaining hashtable. The hashtable allows more parallelism and the operations applied on it involve fewer objects that might potentially conflict. This drastically improved the performance.

In Labyrinth we have seen that all aborts happen when copying the routing matrix to a local data structure. We have used a well known technique based on early release to exclude significant part of the read set from conflict detection. This improved the application’s scalability to match the one reported by its authors.

When porting Genome from C to C#, we have unnoticeably used an open addressing hashtable. Because of the object granularity conflict detection, all insertions to a map data structure were causing conflicts. We have changed the implementation of the hashtable to be chaining. This reduced the conflict rate and made genome scale.

Vacation and WormBench did not have performance problems. However in Vacation we saw that the most aborting atomic block encloses a while loop. We were tempted to move the atomic block inside the loop as in Figure 1 but that would change the specification of the application that the user can specify the number of the tasks to be executed atomically. Moving the atomic block inside the loop would always execute one task and therefore reduce the conflict rate but the user will no longer be able to specify the number of the tasks that should execute atomically. Also, similar changes may not always preserve the correctness of the program because they may introduce atomicity violation errors.

7.7. Related Work

Chafi it et al. (16) have developed Transactional Application Profiling Environment (TAPE) which is a profiling framework for HTMs. The raw results that TAPE produces can be used as input for the profiling techniques that we have proposed. This would enable profiling transactional applications that execute on HTMs or HyTMs.

In like manner, the Rock processor provides a status register to understand why transactions abort (17). Unlike our approach, the reported results through the status register are specific for the TM unit of the processor. For example, a transaction aborted due to a buffer overflow, or transaction aborted due to a cache line eviction. Profiling applications in this way is complementary to our work which will allow users to further optimize their code for certain TM system implementations.

Independently from us, Lourenco it et al. (18) have developed tool for visualizing transactions similar to the transaction. They also summarize the common transactional characteristics that are reported in the existing literature such as abort rate, read and write set, etc. over the whole program execution. Our work complements theirs by reporting results in source language such as variable names instead of machine addresses. Also, we provide local summary which is helpful for examining the performance of specific part of the program execution.

Sonmez et al. (19) have used per- atomic block statistics to profile Haskell-STM applications. In our work we also provide per-atomic block statistics. Unlike Sonmez, our statistics include contextual information comprising the function call stack which is displayed via the top-down and bottom-up views. The contextual information is crucial for finding the critical path in large applications where atomic blocks can be executed from various functions and where atomic blocks include library calls. In addition, we generate abort graphs which relate the winner atomic blocks with the victim atomic blocks.

In our work, we also generate the common statistical data used in the research literature to describe the transactional characteristics of the TM applications such as time spent in transactions, read set, write set, abort rate, etc. In addition we generate a histogram about how much of the transactions' execution interleave. This information is particularly useful to see the amount of parallelism in the program and find cases when a program does not abort but also does not scale.

8. Feedback Directed Compilation

Feedback directed compilation is a work in progress. It focuses on to experimentally evaluate whether the performance of transactional memory applications can be improved by providing feedback to the compiler from earlier executions. In particular, the goal is to reduce the amount of wasted work caused by the aborting transaction. However, reducing the wasted work does not necessarily imply that the overall performance will improve. For example, it might be possible that we obtain poor performance because of serializing the execution of atomic blocks.

We have identified four techniques that can be used for feedback directed compilation that would reduce the wasted speculative execution in the transactional memory applications. These techniques are:

1. Static scheduling
2. Hoisting the log operations for the writes (i.e. early OpenForWrite)
3. Transaction checkpointing
4. Logging read operations as if they are write operation (using OpenForRead instead of OpenForWrite).

The scope of this work is:

1. Profile the available to us STAMP applications (Bayes, Intruder, Genome, Labyrinth, Vacation) and find how the above techniques can be used to reduce the wasted speculative execution in these applications.
2. Evaluate how the above techniques affect the performance of the STAMP applications by manually instrumenting their source code with the respective optimization technique that would have positive effect.
3. Write a paper about the findings and submit to ASPLOS'2011.

8.1. Static Scheduling

In STM and HTM implementations that support transaction scheduling, based on the conflict relationship obtained from the aborts graph (see Section 7.3) the compiler may decide to schedule two or more transactions to not execute concurrently because they would most likely conflict. For example, in Figure 17 AB10 and AB5 abort AB15. In this case, the compiler may decide to statically schedule AB15 to not execute concurrently with AB5 and AB10. However, this may serialize the program execution and have undesired effect causing poorer performance.

8.2. Hoist OpenFowWrite

To detect conflicts between two transactions, the underlying TM implementation needs to keep track of the memory locations that are accessed transitionally. We will arbitrarily call `OpenForWrite` the operation used to tell the TM system that an object is transactional written. Figure 21 shows how a user code is instrumented with calls to an STM system with in-place updates.

We have found that TM systems which detect write-write conflicts eagerly can benefit from hoisting the `OpenForWrite` operation. For example, suppose that atomic blocks in Figure 22 always aborts at the call `OpenForWrite(x)` at line 6. This would cause a lot of wasted speculative execution – something that we should try to reduce. If we move the call to `OpenForWrite(x)` upper as in the "hoist" example then the conflict will be detected early and the atomic block will

abort without accumulating wasted speculative execution. However, this might have side effects too resulting in pathological cases such as *SerializedCommit* or *RestartConvoy* (20).

```
User code                        Code with TM instrumentation
---------                        ----------------------------
1: atomic {                      StartAtomic();
2:    x = 1;                     OpenForWrite(x);
3: }                             x = 1;
4:                               CommitAtomic();
```

Figure 21. STM code instrumentation.

```
Default                                Hoist
-------                                -----
 1: StartAtomic()                     1: StartAtomic()
 2:    <statement1>                   2:    OpenForWrite(x)
 3:    <statement2>                   3:    <statement1>
 4:    <statement3>                   4:    <statement2>
 5:    ...                            5:    <statement3>
 6:    OpenForWrite(x);               6:    ...
 7:    x = 1;                         7:    x = 1;
 8: CommitAtomic();                   8: CommitAtomic();
```

Figure 22. Hoist OpenForWrite.

8.3. Transaction Chekpointing

Transaction checkpointing can be useful for long transactions that abort most of the times in a specific location and the code executed before that location is valid. For example, consider again the Default code from Figure 22. If we checkpoint the transaction just before `OpenForWrite(x)` and the code before the checkpoint is valid, then this transaction will not restart from the beginning but it will only re-execute the last operation.

In TM system we can implement the effect of checkpointing using non-flattening nested atomic blocks (see Figure 23). In the example with the nested transaction, if conflict happens inside the nested atomic block and the code outside the nested atomic block is valid, only the code inside the nested atomic block will re-execute. This approach of re-using nested atomic blocks for code optimization seems even more interesting.

```
Default
-------
 1: atomic(){
 2:    <statement1>
 3:    <statement2>
 4:    <statement3>
 5:    ...
 6:    atomic {
 7:       x = 1;
 8:    }
 8: }
```

Figure 23. Using nested transactions instead of checkpoints.

8.4. Using OpenForWrite for Read Operations

As in Figure 21, the compiler with STM support will generate calls to `OpenForRead()` function for the read operations. In TMs with eager versioning such as Bartok-STM, if a transaction is always aborted because of a value read a certain statement is invalid then we can replace `OpenForRead()` operation at that statement with an `OpenForWrite()`. Opening the value for read instead of write will make other transactions to abort when they attempt the same operation. In this case, the transaction will make a forward progress if it is not aborted because of other conflicts. However, this may have side effect of making the other transaction to abort and may not improve the performance.

8.5. Limitations

Although these ideas seem to be very interesting and useful, we may fail to show that they are useful when applied for the STAMP applications. The reason is that, based on our prior profiling, the applications Genome, Labyrinth and Vacation has very low wasted work and only Bayes and Intruder have a bit higher wasted work. Consequently, the changes that we make in the code may introduce overheads or interfere with other parts in the code and mask the performance gains or even make the application run slower.

Bibliography

1. **Laurus, James and Rajwar, Ravi.** *Transactional Memory (Synthesis Lectures on Computer Architecture).* 2007.

2. *Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server.* **Zyulkyarov, Ferad, et al.** 2009. PPoPP '09: Proc. 14th ACM SIGPLAN symposium on principles and practice of parallel programming. pp. 25-34.

3. *Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games.* **Lupei, Daniel, et al.** s.l. : EuroSys'10: Proc.5th European conference on Computer systems, 2010.

4. *STAMP: Stanford Transactional Applications for Multi-Processing.* **Chi, Cao Minh, et al.** 2008. IISWC '08: Proc. 11th IEEE international symposium on workload characterization. pp. 35-46.

5. *STMBench7: A Benchmark for Software Transactional Memory.* **Guerraoui, Rachid, Kapalka, Michal and Vitek, Jan.** 2007. EuroSys '07: Proc. 2nd European systems conference. pp. 315-324.

6. *WormBench: A Configurable Workload for Evaluating Transactional Memory Systems.* **Zyulkyarov, Ferad, et al.** 2008. MEDEA '08: Proc. 9th workshop on memory performance. pp. 61-68.

7. *RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications.* **Kestor, Gokcen, et al.** 2009. TRANSACT' 09: 4th workshop on transactional computing.

8. **Pankratius, Victor, Adl-Tabatabai, Ali-Reza and Otto, Frank.** *Does Transactional Memory Keep Its Promises? Results from an Empirical Study.* s.l. : University of Karlsruhe, 2009.

9. *Is Transactional Programming Actually Easier?* **Rossbach, Christopher J., Hofmann, Owen S. and Witchel, Emmett.** s.l. : PPoPP '10: Proc. 15th ACM SIGPLAN symposium on principles and practice of parallel programming, 2010.

10. *QuakeTM: Parallelizing a Complex Serial Application Using Transactional Memory.* **Gajinov, Vladimir, et al.** 2009. ICS '09: Proc. 23rd international conference on supercomputing. pp. 126-135.

11. *Debugging with Transactional Memory.* **Lev, Yossi and Moir, Mark.** 2007. TRANSACT' 07: 2nd workshop on transactional computing.

12. *tm_db: A Generic Debugging Library for Transactional Programs.* **Herlihy, Maurice and Lev, Yossi.** 2009. PACT '09: Proc. 18th international conference on parallel architectures and compilation techniques. pp. 136-145.

13. *Using Hardware Transactional Memory for Data Race Detection.* **Gupta, Shantanu, et al.** 2009. IPDPS '09: Proc. 23rd IEEE international parallel and distributed processing symposium. pp. 1-11.

14. *Debugging Programs that use Atomic Blocks and Transactional Memory.* **Zyulkyarov, Ferad, et al.** s.l. : PPoPP'10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010.

15. *Optimizing Memory Transactions.* **Harris, Tim, et al.** s.l. : PLDI '06: Proc. 2006 ACM SIGPLAN conference on programming language design and implementation, 2006.

16. *TAPE: A Transactional Application Profiling Environment.* **Chafi, Hassan, et al.** 2005. ICS '05: Proc. 19th international conference on supercomputing. pp. 199-208.

17. *Early experience with a commercial hardware transactional memory implementation.* **Dice, Dave, et al.** s.l. : ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, 2009.

18. *Understanding the behavior of transactional memory applications.* **Lourenco, Joao, et al.** s.l. : PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems, 2009.

19. *Profiling Transactional Memory applications on an atomic block basis: A Haskell case study.* **Sonmez, Nehir, et al.** s.l. : MULTIPROG '09: Proc. 2dn workshop on Programmability issues for multi-core computers , 2009.

20. *Performance pathologies in hardware transactional memory.* **Bobba, Jayaram, et al.** 2007 : ISCA '07: Proc. 34th international symposium on computer architecture.

21. *Inferring locks for atomic sections.* **Cherem, Sigmund, Chilimbi, Trishul and Gulwani, Sumit.** 2008. PLDI '08: Proc. 2008 ACM SIGPLAN conference on programming language design and implementation. pp. 304-315.