QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory

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

[†]Barcelona Supercomputing Center [*]Universitat Politecnica de Catalunya [‡]Microsoft Research Cambridge

vladimir.gajinov@bsc.es ferad.zyulkyarov@bsc.es osman.unsal@bsc.es adrian.cristal@bsc.es
eduard.ayguade@bsc.es tharris@microsoft.com mateo.valero@bsc.es

ABSTRACT

"Is transactional memory useful?" is the question that cannot be answered until we provide substantial applications that can evaluate its capabilities. While existing TM applications can partially answer the above question, and are useful in the sense that they provide a first-order TM experimentation framework, they serve only as a proof of concept and fail to make a conclusive case for wide adoption by the general computing community.

This paper presents QuakeTM, a multiplayer game server; a complex real life TM application that was parallelized from the sequential version with TM-specific considerations in mind. QuakeTM consists of 27,600 lines of code spread across 49 files and exhibits irregular parallelism for which a task parallel model fits well. We provide a coarse-grained TM implementation characterized with eight large transactional blocks as well as a fine-grained implementation which consists of 58 different critical sections and compare these two approaches. In spite of the fact that QuakeTM scales, we show that more effort is needed to decrease the overhead and the abort rate of current software transactional memory systems to achieve a good performance. We give insights into development challenges, suggest techniques to solve them and provide extensive analysis of the transactional behavior of QuakeTM, with an emphasis and discussion of the TM promise of making parallel programming easier.

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

General Terms: Design, Experimentation, Performance.

Keywords: Game Server, Transactional Memory

1. INTRODUCTION

Recently, processor manufacturers have done a right-hand turn away from increasing single core frequency and complexity. Low returns from instruction level parallelism (ILP) and problems with power/heat density have led to the appearance of multi-core processors that leverage thread level parallelism (TLP). In this new era of multi-core architectures, the coordination of the work done by the multiple threads that cooperate in the parallel execution is one of the challenging issues both in terms of programming productivity and execution performance. Transactional memory (TM) is a technology that may help here, by aiming to provide the performance of fine-grained locking with the ease-of-programming of coarse-grained critical sections. In this paper we assess the extent to which this is true of current TM implementations, based on code descriptions and examples as well as through performance evaluation.

As a case study we started from a sequential version of Quake, a complex multi-player game. Using OpenMP and software transactional memory (STM) we built QuakeTM, a parallel version which consists of 27,600 lines of code spread across 49 files. Developed in 10 man-months, QuakeTM exhibits irregular parallelism and long transactions contained within eight different atomic blocks with large read and write sets.

Our intention was not to pursue performance *per se*, but to examine whether or not it is possible to achieve good results with a coarse-grained parallelization approach. This decision was driven by one of the hopes for TM, to make parallel programming easier by abstracting away the complexities of using fine-grained locking, while still achieving good scalability. When parallelizing an application from scratch using TM, this kind of coarse-grained approach is likely to be popular with programmers. Consequently, this approach needs to be tested on a highly complex application in order to see how well it works in practice.

This paper makes following contributions:

- We describe how we developed QuakeTM and discuss the challenges we encountered.
- We show that our implementation scales reasonably well, despite the use of coarse-grained transactions. However, we show that this scalability is unable to compensate for the high overhead and abort rate of the software transactional memory system.
- Further on, we have adapted the fine-grained TM implementation described in our previous work on Atomic

Quake [20] in order to compare these two different parallelization approaches.

- In a pleasant side-effect during our performance optimization effort, we developed a simple mechanism, which we call ReachPoints, that could be useful to discover and isolate TM-related performance problems.

The remainder of this paper is organized as follows: In Section 2, we comment on related work. In Section 3, we describe the structure of the sequential Quake application. In Section 4, we describe the parallelization process and the development of QuakeTM. Section 5, details the evaluation environment and Section 6 follows with the results. In Section 7 we discuss future work and we conclude in Section 8.

2. RELATED WORK

Early TM research used micro-benchmarks to demonstrate the potential of the new programming paradigm. Subsequently, sets of kernel applications have been developed, such as Lee-TM [5], Delaunay mesh refinement and/or agglomerative clustering [9][11][17] and STMBench7 [8]. The total code size of these kernels ranges from 800 lines for Lee-TM to 5000 lines for STMBench7, but the common fact is that the total size of critical sections doesn't exceed a couple of hundred lines of code.

There are several benchmark suites of programs using TM. The Haskell STM benchmark suite [15], consists of nine Haskell applications of different sizes, which target different aspects of an underlying TM system. While it is good for its domain, the Haskell STM benchmark suite is not directly applicable to other languages. The STAMP benchmark suite [13], consists of eight applications that cover a variety of domains and exhibit different characteristics in terms of transaction lengths, read and write set sizes and amounts of contention. The downside of these applications, when used with STM, is the fact that they were manually optimized, with an application level knowledge beforehand, which enabled authors to manually implement the optimal number of read/write barriers in the code. This doesn't help the effort to prove the primary goal of TM which is to make multithread programming easier. If programmers are required to manually instrument the code in order to achieve basic performance then TM is not the solution. As it was pointed out by Dalessandro et al. [6] library interfaces can remain a useful tool for systems researchers, but application programmers are going to need language and compiler support.

In general, most previous TM applications and benchmarks were either derived automatically from lock-based parallel versions (this is, replacing lock-based critical sections with transactions), or if they were developed from sequential versions then the resulting code was somewhat limited in size and complexity (which limits the benefits of detailing the development challenges and the programmer effort). Therefore, there is a clear need to develop a substantial TM application from the ground up while extensively detailing the parallelization process and the challenges involved. This is one of the main contributions of this paper.

In recent work [20], we developed a transactional version of Quake from an existing lock-based version [2][3]. We encountered a different set of challenges when parallelizing the sequential version of Quake with TM. Comparing that approach with the one in this paper gives us a new perspective on how the use of "atomic" blocks in new parallel code might compare with their use as a replacement for lock-based critical sections. We discuss these two issues further in Section 4.

3. QUAKE DESCRIPTION

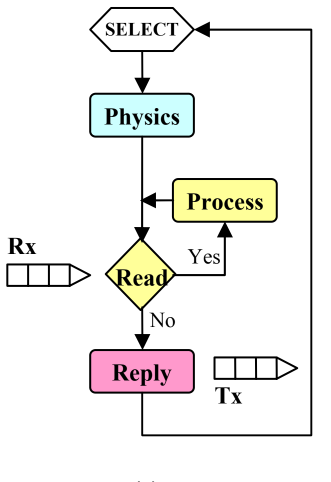
Quakeworld is the multi-player mode of Quake I, the first person shooter game released under the GNU general public license by ID Software. It is a sequential application, built as a client-server architecture, where the server maintains the game world and handles coordination between clients, while the clients perform graphics update and implement user-interface operations.

The server executes an infinite loop, where each iteration performs the calculation of a single frame. The frame execution algorithm is presented in Figure 1(a). The server blocks on the `select` system call waiting for client requests. If requests are present on the receiving port, it starts the execution of the new frame. It is possible to distinguish three stages of frame execution: world physics update (P), request receiving and processing (R) and reply stage (S). Upon the end of execution of all three stages the server frame ends and the process is repeated. Generally, the server will send replies only to clients which were active in the current frame, namely those who have sent a request. All replies are sent after all requests have been processed. This clear separation of the frame stages simplifies the parallelization, as we present later in the paper.

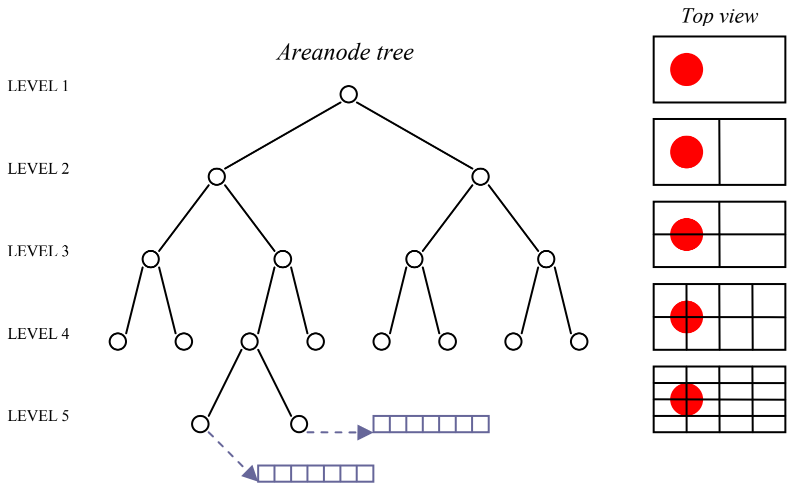
The Quake game world is a polygonal representation of the 3D virtual space in which all objects, including players, are referred to as entities. Each entity has its own specific characteristics and actions it can perform. During the update, the server will send information only for those entities which are of interest to the client. Nevertheless, the server has to simulate and model, not only the players' actions, but also the effects induced by these actions. Thus, server processing is a complex, compute intensive task that increases superlinearly with the number of players [3].

3.1 Map Description

A map of the Quake world is represented with a file which holds the binary space partition (BSP) implementation of the 3D world with all the details relevant to draw and position the objects in the world [7][12]. The level of details contained within theBSP tree is large; therefore BSP trees are hard to maintain for dynamic scenes. If the server wants to generate a quick list of the objects that an entity may interact with, traversing the BSP tree is inefficient, and since this is a common operation involved in each move command, the server constructs and maintains a secondary binary-tree structure, called an areanode tree. This is a 2D representation of the BSP tree, constructed during server initialization by dividing the 3D volume in the x-y plane. Figure 1(b) demonstrates the building process. Each areanode has an associated list of objects contained within the space defined by that areanode. When an object is moved, it is necessary to update the areanode tree to reflect the new position of the object. This is done by removing the object from the original list, and inserting it into the list of the areanode that corresponds to the destination of the object.



(a)



(b)

Figure 1: (a) Frame execution algorithm. (b) Constructing areanode tree from the BSP map volume. Adapted from [3]

3.2 Move Execution

Clients influence the gameplay by sending the move command, which specifies various parameters related to the player's state, actions and intentions. Using the data extracted from the command (motion indicators, origin of the player and time to run the command), the server constructs the bounding box of the player's motion, thus defining the region of the world it can affect. Then it traverses the areanode tree to find all the objects contained within this bounding box and associate them with the move command. It then simulates the move, and upon completion, removes the player's object from the old position in the game world and links it to the new one.

4. PARALLELIZATION

Parallelization of the Quake server has already been done using POSIX threads (pthreads) in [2]. It is the fine-grained lock-based implementation which, to the best of our knowledge, took about 15 man-months of development. That version is the base for our previous work on Atomic Quake [20], whose main objective was to evaluate the effort of replacing locks with transactions. In that paper [20] we report that the process was not straightforward and that we encountered specific challenges which considerably increased the development effort. First, the lock parallelization is not block structured which required code reorganization to adapt to the TM model. The second problem was to avoid I/O operations which is not an issue in a lock based system. Finally, a big fraction of the development time was spent in understanding how locks are associated with the variables and to get a grip with the locking strategy.

This paper doesn't build on top of the pthreads version, but implements a different parallelization strategy. Our intention was to start from the sequential application and parallelize it using OpenMP and transactional memory. Further on, we deliberately decided to implement a coarse-grained parallelization approach driven by the desire to test one of the promises of transactional memory, which is to make multithreaded programming easier by providing good performance scaling coupled with a coarse-grained parallelization effort.

In order to give a complete overview of the parallelization of the Quake game server using transactional memory, we also adapted the fine grained implementation described in Atomic Quake. Later on, in the results section, we give a comparison between these two implementations and show that both of them suffer from the same problems associated with high transactional memory instrumentation overhead, though the extent in which the fine-grained version is affected is lower.

Semantically comparing these two solutions, it is clear that QuakeTM, a coarse-grained implementation, is characterized with atomic blocks and read and write sets that are an order of magnitude bigger. Overall, there are only eight transactional blocks in QuakeTM compared to 58 in the fine-grained implementation of Atomic Quake. These facts lead to a noticeable difference in the abort rate which is 35.3% in the QuakeTM compared to only 4.1% in the Atomic Quake case. Nevertheless, the TM instrumentation overhead is still high in both implementations, but it does seem to be proportional to the read set size of transactions.

4.1 QuakeTM

One of the main goals of this work was to test the primary objective of transactional memory [10], to make multithreaded programming easier. We had no prior knowledge of the application itself, so we took some time to understand the code. We have identified the parts of the application suitable for parallelization, using profile information and by studying the program structure. The process of adding OpenMP parallelization pragmas and transactional boundaries was then straightforward and simple, if we disregard occasional problems with the compiler. The real challenge was to identify which of the global data structures and variables must be shared and which can be re-structured as thread-local data. From a performance perspective this is crucial, since a lot of sharing leads to bad performance. The development of QuakeTM, our coarse grained version, was done in ten man-months. Additional two months were spent on adapting the fine-grained solution described in our work on Atomic Quake [20]. In this section we describe our QuakeTM parallelization approach.

```
while (NET_GetPacket ()) {
   // Filter packets
   if (connection related packet) {
      SV_ConnectionlessPacket ();
      continue;
   }

   // game play packets
   for (i=0 ; i<MAX_CLIENTS ; i++) {
      // Do some checking here
      SV_ExecuteClientMessage ();
   }
}
```

(a)

```
while (NET_GetPacket ()) {
   // Filter packets
   if (connection related packet){
      SV_ConnectionlessPacket ();
      continue;
   }
   AddPacketToList();
   CopyBuffer();
}

#pragma omp parallel shared(packetlist, ...){
   #pragma omp single
   while (packetlist != NULL) {
      #pragma omp task firstprivate(packetlist){
         NET_Message_Init(..);
         // Do some checking here
         for (i=0 ; i<MAX_CLIENTS ; i++){
            // Do some checking here
            SV_ExecuteClientMessage ();
         }
      }
      packetlist = packetlist->next;
   }
}
```

(b)

Figure 2: Pseudocode for the request processing stage: (a) sequential and (b) parallel implementation

4.1.1 Shared Data

During the gameplay, there are three types of shared data structures: message buffers, the areanode tree and game objects (entities). Among the buffers we further distinguish the global state buffer and per-player reply buffers. The global state buffer, updated in the physics update stage and the request stage, is used to hold the updates that reflect the actions of all players involved in the game session.

Accesses to the areanode tree are in the form of linked list operations on the object lists associated with each areanode. The access pattern for the request stage has already been covered in the explanation of the move command. A similar pattern is observed in the physics update stage since physical influences, which may affect an object, can change its position and hence its areanode container.

Game objects are updated in the physics update stage and the request processing stage. During the move execution, each object that is touched is updated in a global, shared part of the memory which is divided into a number of regions (strings, functions, statements, field definitions, global definitions, globals and entities). The fact is that only the entity part of the program memory has to be shared, while other parts should be thread-private to achieve better concurrency.

4.1.2 Parallelization Strategy

An execution breakdown of the sequential Quake server is given in Table 1. It is clear that the majority of time is spent in the request processing phase, and even though physics update exhibits similar shared data access patterns, its operations seem to be significantly less involved. Therefore, we concentrate on the parallelization of the request processing stage. The algorithm for this stage is presented in Figure 2(a). Each iteration of the loop in Figure 2(a) performs the execution of a single client request. A tasking model is the most suitable for parallelization of this loop for two reasons: (i) given the diversity of the client requests, load balancing is an important factor to consider and (ii) we are not aware of the number of requests that are pending on the receiving port. The profiling information tells us that the time needed to receive all packets is negligible compared to the processing time. Therefore, to enable application of the tasking model, we separate the receiving phase from the processing phase, and receive all packets first, storing them into a temporary list. Afterwards, the list is traversed and a processing task created for each packet in the list. Figure 2(b) illustrates our approach.

The QuakeTM coarse-grained approach consists of eight large atomic blocks, but here we describe only the four which are involved in synchronization of the client move command processing, the most common action performed by the Quake server. The client message is extracted from the packet and parsed into one or more commands. Each message can hold only one move command which is by far the most common type of commands. The execution of the move command is illustrated by the diagram in Figure 3 together with the transactional block markers. The execution begins with the client's physics update (`ClientPhysics`) followed by the so called "think" function (`ClientThink`). This is a special feature of Quake to register an action that needs to be carried out, in regard to the client, in the future. This is not specific to the client entities, but overall this is the way to implement actions that exceed the duration of a single frame. Along with the pmove (player move) structure initialization (`PmoveInit`), these actions form the preparation for the actual move execution, and can be contained inside a single transactional block.

The next phase in the execution – `AddLinksToPmove`, determines which entities could be affected by the current move command. Starting from the origin of the player the maximum affected area is determined. Next, the areanode tree is traversed and the corresponding areanode entity lists checked to discover those objects whose position falls into this area. Links to all affected objects are added to the pmove structure entity list.

Further processing is carried on with the execution of the `PlayerMove` function. First, a model box is assigned to the player's entity and each entity from the pmove entity list. Then, using the parameters from the move command extracted from the received message, a trajectory is followed from the player's original position to its potential destination. If the player's model box clips a model box of the other entity moving along the trajectory line, there is a collision between them. Depending on the various parameters of the collided objects and the environment that surrounds them (air, water, solid area, etc.) the player's final position is calculated.

The last phase of the move execution is `LinkEntity` function which re-links the player's entity to the new position in the areanode tree. In the end, the player's influence during its movement is applied to each entity that was "touched" along the trajectory. This action is denoted as `PlayerTouch` in Figure 3.

4.2 Parallelization Issues and ReachPoints

Even though we dedicated significant time to manually identify global data that could be thread-private, for unmanaged code written in a sequential programming style, where a vast amount of data is global, as in the Quake case, it is not enough. During our attempt to boost performance we came up with a solution, which we call ReachPoints, that helped us identify the rest of the global variables that could be thread private and discover the problems that arose from TM cache-line granularity conflict detection implementation.

The ReachPoints solution, shown in Figure 4, consists of allocating an array of counters for each thread, taking cache line granularity into account (x*16 integers for each thread where x=1,2,... and cache line size of 64 bytes). At the end of execution, when we print the state of counters, the difference between two counters pinpoints the region of the code where transactions abort. Analyzing that region, it is possible to discover causes for the aborts. Simple as it may be, we found ReachPoints very valuable and useful. As already stated, it even helped us discover sources of false conflicts referred to as false sharing [21], which occur mostly within structured data and are the consequence of the fact that two different variables or structure fields reside in the same cache line. Assuming that only one of them is written, say variable X, under the cache line granularity conflict detection system, a read-only variable Y is also causing conflicts, since the writes to X are treated as writes to Y. When we applied cache padding for such cases, we noticed a significant performance improvement due to a decrease in the number of aborts. This solution is specific for eager conflict detection, since in TM systems with lazy conflict detection transactions can be aborted anytime, regardless of the read or write issued in the moment of the abort. It should be said that one could think of better mechanisms to detect which data access is causing conflicts, and one of them is certainly debugger support.

Table 1: Execution breakdown of the sequential server

Stage	Time [%]
Request Processing	87.8
Reply	3.1
Physics Update	2.1
Measuring and Info	5.3
Other	1.7

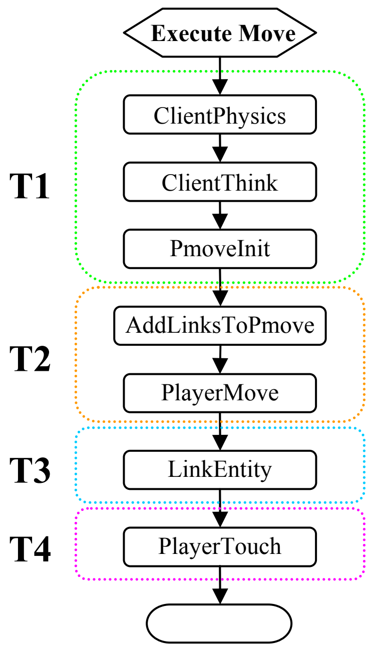


Figure 3: Diagram of the move command execution with transactional block markers.

```
int reachpoints[NumThreads][x*16]

TM_PURE
void PointReached(int check) {
   reachpoints[ThreadId][check]++;
}

int main () {
   . . .
   TRANSACTION
      PointReached (1);
      statement_1;
      PointReached (2);
   TRANSACTION_END
   . . .
}
```

Figure 4: ReachPoints: Simple solution for discovering conflicting regionsof the transactional code.

5. EVALUATION

For testing purposes, we used the existing Quake client code to build an automatic trace client called TraceBot, whose behavior is controlled by a finite state machine. We also had to implement certain changes on the server side to be able to synchronize the client's actions with the server response. Essentially, TraceBot is simply sending messages at the server frame rate until it dies, as a result of actions of the other connected players, or until the end of the trace, when it commits suicide. After TraceBot dies, it sends another special string command which function is to respawn the client into the game world and the process is repeated. The traces are recorded using VideoClient, which is similar to TraceBot with the addition of graphics. To record traces we use the original, sequential Quake server, and connect VideoClient to play the game, producing traces that represent recorded human actions.

We run the server on one machine and the clients on another, to simulate the real game environment, given that network latency and bandwidth are not critical [2]. The server and client frame rates are synchronized and set to 100ms which is enough time for the worst case transactional frame length. Both machines are PowerEdge 6850, with four dual-core 64-bit Intel® Xeon™ processors running at 3.2 GHz, with 16MB L3 cache memory per processor unit, running SUSE LINUX 10.1.

In this work we are using the prototype version 3.0 of the Intel STM C/C++ compiler [4][14][19] with level O3 optimizations enabled. The underlying STM implementation is an extended version of the McRT-STM system [16]. The compiler implements both optimistic and pessimistic concurrency control, and provides single lock atomicity semantics and weak atomicity guarantees. Serial execution mode is provided to support system calls and I/O operations inside transactions. To optimize function calls within transactions the compiler introduces function annotations: `tm_callable`, `tm_pure` and `tm_unknown`. Nesting is supported in a closed nesting fashion via flattening; a data conflict rolls back to the outermost level and re-executes the transaction. It uses cache-line granularity conflict detection and implements strict two-phase locking for writes. Writes update values in place and generate undo log entries. Transactions validate the read set at commit time, and if necessary during the read operation, which means that transaction can abort any time during the execution when it encounters a conflict.

6. RESULTS

To test the performance of QuakeTM, we compare the results with the sequential and global lock versions. For the parallel setups we vary the number of threads from one to eight. We also vary the number of clients from one to sixteen and take the mean of five runs. Each run executes for 2000 frames (about 200s of real time). The results are collected for the last 1000 frames in order to avoid effects from server initialization and client connection times. It should be noted that in our evaluation system it is not necessary to stress the server by running a large number of clients; if the server is able to service given number of clients faster, it is able to service more clients in a desired frame length.

We use the `rdtsc` instruction to measure the number of cycles between two events and then translate that value into milliseconds. Even though only the request processing stage is parallelized, we present results and for the entire frame execution.

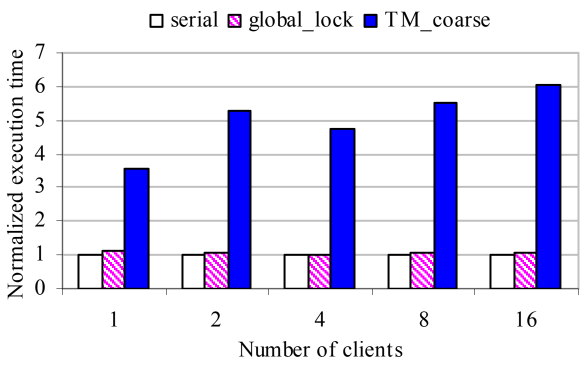


Figure 5: Normalized average frame execution times of the QuakeTM parallel implementations for the single-thread case. The baseline is the sequential execution.

Figure 5 presents normalized average frame execution times of the parallel implementations for the single-thread case. The baseline is always the average frame execution time of the sequential server for a given number of clients. Since there is no contention, the lock version introduces almost no overhead. On the other hand, the overhead for the transactional version goes from 3.5 times for a single client to 6 times for sixteen clients. These results exceed the findings of Wang et al. [18] for non optimized version of STM. For microbenchmarks the authors report an overhead of non-optimized STM code from 2.4 to 4.5 times over a fine-grained locking implementation. For the SPLASH-2 benchmarks the reported overhead doesn't exceed 20%, but it is a measure across the entire execution, which hides the fact that little time is spent in critical sections. In our case more than 85% of the time is spent in critical sections.

Figure 6 shows the comparative performance of the QuakeTM and the global lock version for different numbers of connected clients. As expected, the global lock version doesn't scale, while the transactional version starts to scale only when the workload becomes sufficient, which happens with eight connected clients. We can see that the transaction overhead remains approximately 4x-6x. When we run the application with sixteen clients, then we start to notice a considerable speedup. Figure 7(a) gives a better view of this case. The values are normalized to a single thread execution time. The speedup for eight threads is 1.62 which is a good initial result, considering that this is the first real application to test TM capabilities, but it is still not enough to cover the costs of running transactions. Figure 7(b) shows the scalability of the transactional Quake server running with sixteen clients. It is obvious that the TM version scales, but it still performs worse than the global lock version, even though it was able to compensate for more than 50% of the transactional overhead.

To discover the reasons why the transactional version doesn't perform better, it is necessary to look at the statistical data which is provided by the Intel compiler. Table 2 presents these statistics for the TM configuration running with eight threads. All statistical values increase when we increase the number of clients connected simultaneously, but the most important, from a performance perspective, is the transaction abort rate. In the case of sixteen connected clients 35.3% of transactions abort, causing a high amount of wasted work. There are examples when a transaction aborted 136 times before it eventually committed.

This leads to a significant waste of processor cycles to re-execute the transactional code. Table 2 also shows that although the mean value of the data read is about 5.1 KB there are cases when it grew up to 1.7 MB. This is an important factor which could stress the design of any hardware transactional memory system.

Table 3 presents the execution breakdown of the TM server running with eight threads and sixteen clients. Out of eight atomic blocks implemented in QuakeTM only the five presented in the table contribute considerably to the overall performance. It can be seen that the abort overhead is not significant even for the atomic blocks which have a high abort rate. Since we are not in the position to gain an insight from the profiling due to the fact that Intel compiler is not open source, we can only speculate about the reasons for such results. A possible explanation could be that transactions are aborted early during the execution or that the contention handler and the abort mechanism are efficient. On the other hand, the instrumentation time may be high as a result of the STM runtime operations associated with locks.

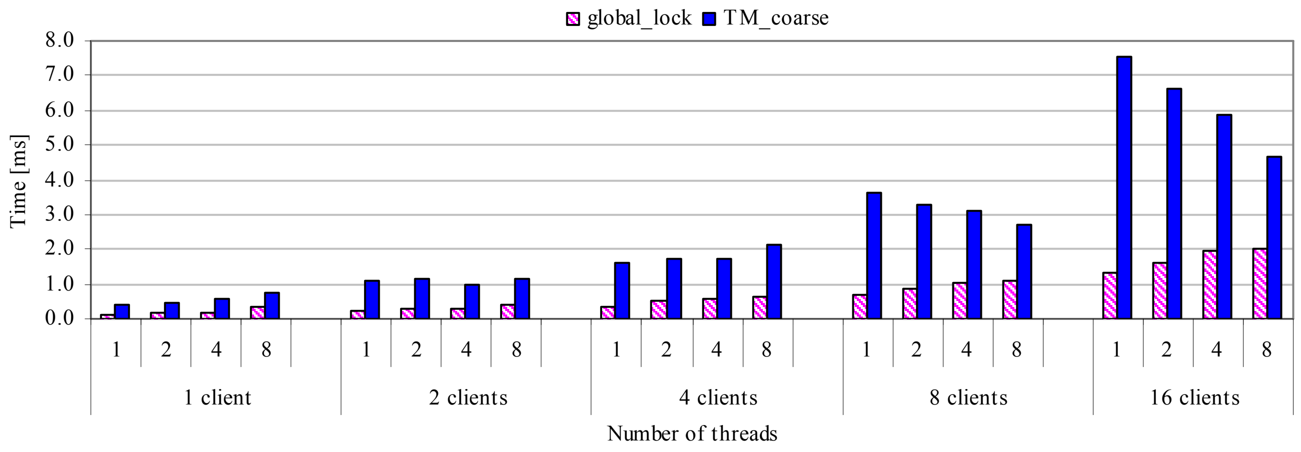
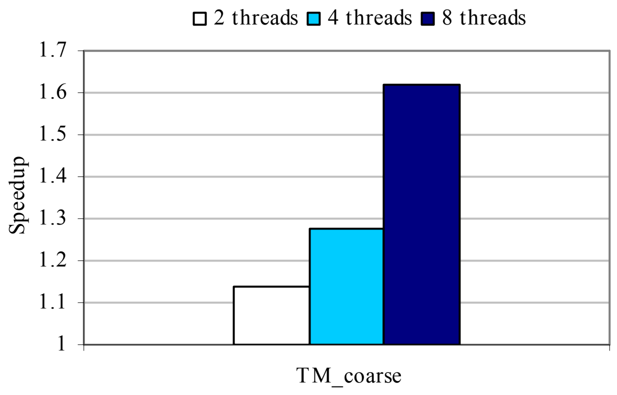
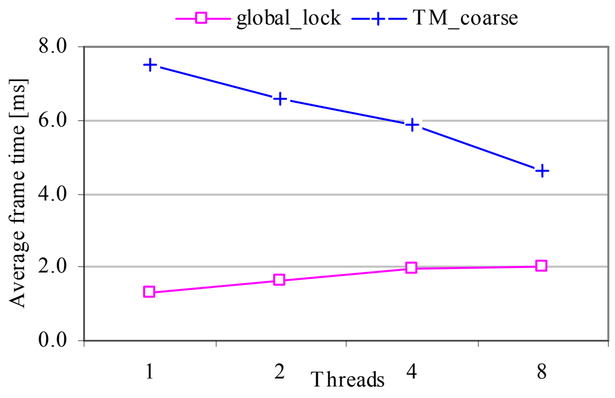


Figure 6: Comparative performance of parallel configurations





(a) **(b)**

Figure 7: Parallel servers running with 16 clients: (a) Speedup, (b) Scalability.

Table 2: Transactional statistic of the QuakeTM server running with 8 threads.

Clients	Transactions	Aborts	Abort rate [%]		Mean [KB]	Max [KB]	Total [MB]
1	34754	0	0.0	Reads	3.0	104	105
				Writes	0.6	17	20
2	95980	1970	2.1	Reads	2.8	863	263
				Writes	0.6	164	55
4	179241	10820	6.0	Reads	3.4	1413	570
				Writes	0.6	269	108
8	364305	76560	21.0	Reads	4.2	1478	1207
				Writes	0.8	251	216
16	524561	184992	35.3	Reads	5.1	1704	1725
				Writes	0.9	262	296

Table 3: QuakeTM transactional execution breakdown.

TM block	Multithread execution - 8 threads, 16 clients					
	Total [10^9 cycles]	Instrumentation overhead		Abort overhead		Abort rate [%]
		10^9 cycles	%	10^9 cycles	%	
1	13.5	10.3	75.8	3.3	24.2	19.5
2	9.5	9.0	94.1	0.6	5.9	18.0
3	17.2	15.1	87.9	2.1	12.1	52.7
4	11.6	10.9	94.3	0.7	5.7	22.4
5	5.9	3.2	53.7	2.8	46.3	61.1
overall	57.9	48.5	83.8	9.4	16.2	35.2

6.1 Comparison with Atomic Quake

The fact that QuakeTM and Atomic Quake had different starting points leads to some differences between them. In QuakeTM we started from the sequential version and were free to choose our parallelization strategy which eventually resulted in the choice of fork-join parallelism supported by OpenMP. Conversely, with Atomic Quake, we inherited the base parallelization structure and changed only the synchronization mechanism from locks to transactions. Therefore, Atomic Quake uses the pthreads model and implements manual thread control where the frame execution logic is also changed to some extent. Moreover, the QuakeTM client-server protocol was slightly changed to enable automatic control of the TraceBot client (we added two control signals on both sides to stop the currently running trace and start a new one from the chosen point in the game world). These were the reasons why we chose not to do a straightforward comparison, but to reimplement fine-grained critical sections and locking techniques the same way Atomic Quake does, with just a few modifications. Therefore, the resulting implementation is equivalent to Atomic Quake in the sense of sharing a fine-grain parallelization approach.

We again start with the overhead graph shown in Figure 8. As expected, in the fine-grained lock implementation of Atomic Quake we see comparatively more overhead compared to the sequential version than in the global lock case of QuakeTM. This is partially due to data copying from global to private buffers and *vice versa*, and partially due to the programming patterns associated with the use of locks which are necessary to avoid lock related problems like deadlock and livelock situations. This overhead is approximately 50%. For the TM case we have the overhead of 2.4x – 3x which is 50% decrease compared to what we had in the QuakeTM case. This can be explained by the smaller size of transactions, especially the read set sizes which are an order of magnitude smaller in a fine-grained implementation.

The comparative performance of the fine-grained lock and Atomic Quake TM implementations is shown in Figure 9, while the speedup and scalability are presented in Figure 10. Speedup of the fine-grained lock version is 1.63 while the speedup of a TM implementation is 1.5. For completeness, Figure 10(b) also shows the performance of the global lock and QuakeTM versions. The fine-grained lock version performs the best followed by the global

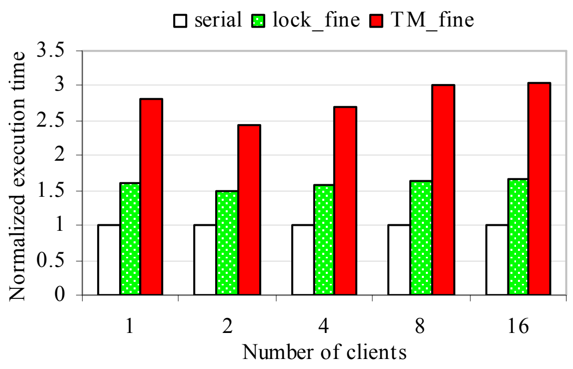


Figure 8: Normalized average frame execution times of the AtomicQuake parallel implementations for the single-thread case. The baseline is the sequential

lock version.The Atomic Quake transactional version comes close to the global lock version while the coarse grained implementation of QuakeTM falls behind. It is now clear that both transactional versions pay a high performance cost associated mainly with instrumentation overhead which is supported by the fact that the abort rate of the Atomic Quake TM server running with eight threads and sixteen clients is only 4.1%. We give the summary TM statistics of 8-thread Atomic Quake in Table 4.

7. FUTURE WORK

Besides the fact that negligible time is spent in execution of the other two stages in the frame, we plan to parallelize them because they exhibit different patterns and could be useful for testing TM implementations. We also plan to modify certain structures, especially the areanode lists, in order to decrease the abort rate and hopefully enable the use of coarser transactions. The QuakeTM source code will be publicly available at `http://www.bscmsrc.eu`.We encourage other implementers to download and use the application to test their TM systems.

8. CONCLUSION

In this paper, we have introduced QuakeTM, the first complex real-world TM application that was developed directly from a sequential version using transactional memory. We have made a detailed description and commented on the challenges involved in the process of doing this work. The emphasis was on testing the TM programmability which led us to take a coarse-grained parallelization approach. As a result, QuakeTM is characterized with large atomic regions that put too much pressure on the underlying STM system. Our evaluation clearly shows that the transactional overhead which results in the 6x slowdown and the abort rate that goes up to 35.3% are excessive and cannot be compensated with the speedup from parallel execution. This leads us to conclude that a coarse-grained approach is not a viable option for the current STM systems. Moreover, we have shown that the read and write set sizes are significant, which could impose serious problems for hardware TM systems. Finally, we were surprised by the amount of programmer time investment.

The overhead results we've seen here have been substantially worse than in our recent work on supporting `atomic` blocks in

C# [1]. When running C# versions of the STAMP benchmarks we typically saw the overhead of running inside an atomic block as substantially less than 2x over unsynchronized sequential code, and we typically saw that the TM implementation outperformed the sequential version as soon as a second core was added.

It remains to be seen whether differences between C and C# mean that similar performance is unlikely in C, or whether these differences merely reflect the ongoing development of prototype implementations which will improve over time. However, unless the aforementioned problems are solved by the future STM implementations in C, the only option left for a programmer is to take a fine-grained parallelization approach. In that case, it remains to be seen whether other characteristics of transactional memory, such as composability and deadlock freedom, are going to justify a switch to the TM programming model.

9. ACKNOWLEDGMENTS

Special thanks to Srdjan Stipic for his help during the work on this paper. We would also like to thank Intel for making the prototype C++ STM Compiler publicly available. This work is supported by the cooperation agreement between the Barcelona

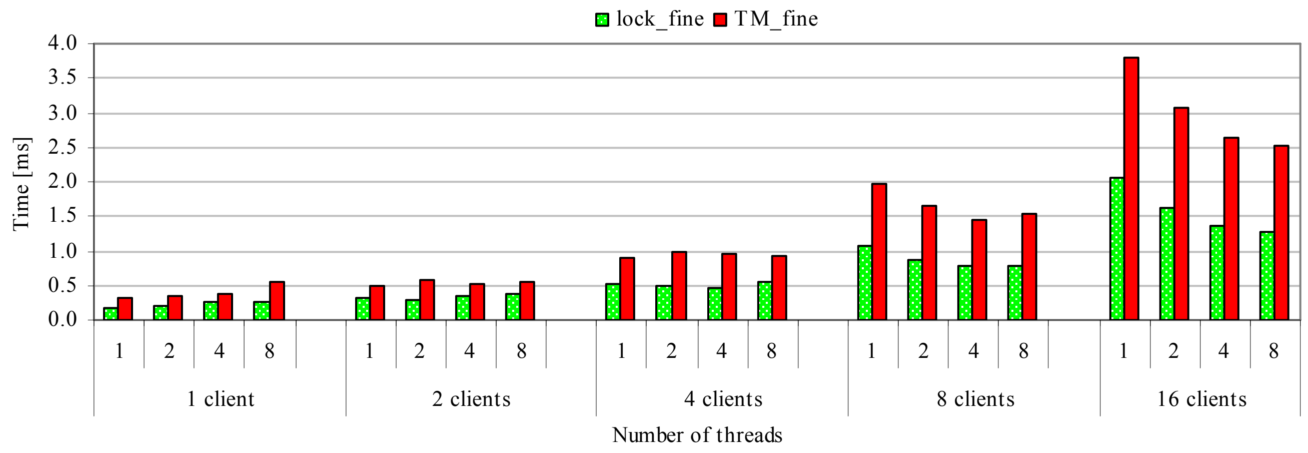
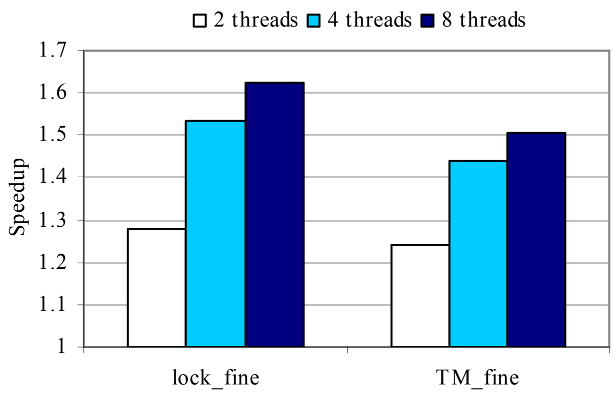
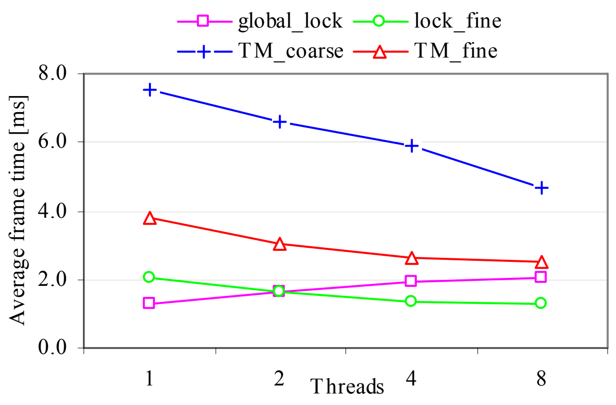


Figure 9: Comparative performance of AtomicQuake parallel configurations



(a)



(b)

Figure 10: Parallel servers running with 16 clients: (a) Speedup, (b) Scalability.

Table 4: Transactional statistic of the AtomicQuake server running with 8 threads

Clients	Transactions	Aborts	Abort rate [%]		Mean [B]	Max [B]	Total [MB]
1	190206	0	0.0	Reads Writes	65.1 5.2	58511 20102	12 1
2	367118	826	0.2	Reads Writes	66.0 5.7	62728 24397	25 2
4	655020	4165	0.6	Reads Writes	83.7 8.2	80275 39726	55 5
8	1439874	20593	1.4	Reads Writes	102.5 9.6	102470 57552	145 14
16	3226759	131814	4.1	Reads Writes	133.3 15.5	231593 211651	192 22

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). Vladimir Gajinov and Ferad Zyulkyarov are also supported by a scholarship from the Government of Catalonia.

10. 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 Symp. on Principles and Practice of Parallel Programming*, Feb. 2009, pp. 185-196

[2] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *IPDPS '04: Proc. 18thIinternational Parallel and Distributed Processing Symposium*, Apr. 2004, pages 72–81.

[3] A. Abdelkhalek, A. Bilas and A. Moshovos. Behavior and performance of interactive multi-player game servers. In *ISPASS'01: Proc. of the 2001 International IEEE Symposium on Performance Analysis of Systems and Software*, Nov. 2001, pages 355-366.

[4] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006, pp. 26-37

[5] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham and I. Watson. Lee-TM: A Non-trivial Benchmark for Transactional Memory. In *ICA3PP'08: Proc. 8th International Conference on Algorithms and Architectures for Parallel Processing*, June 2008, pages 196-207.

[6] L. Dalessandro, V. J. Marathe, M. F. Spear and M. L. Scott. Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In *TRANSACT'07*: *Proc. 2nd ACM SIGPLAN Workshop on Transactional Computing*, Aug 2007.

[7] H. Fuchs, G. D. Abram and E. D. Grant. Near Real-Time Shaded Display of Rigid Objects. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, 1983, pages 65-72.

[8] R. Guerraoui, M. Kapalka and J. Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys'07: Proc. 2nd European Systems Conference*, Mar. 2007, pages 315-324.

[9] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008, pages 207–216.

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

[11] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI'07: Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007, pages 211-222.

[12] M. McGuire. Quake 2 BSP File Format. http://www.flip code.com/archives/Quake_2_BSP_File_Format.shtml

[13] C. C. Minh, J. Chung, C. Kozyrakis and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08: Proc. 11th IEEE International Symp. on Workload Characterization*, Sep. 2008, pp. 35-46.

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

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

[16] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06: Proc. 11th Symposium on Principles and Practice of Parallel Programming*, Mar. 2006, pp. 187-197.

[17] M. L. Scott, M. F. Spear, L. Dalessandro and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *IISWC '07: Proc. 10th IEEE International Symposium on Workload Characterization,* Sep. 2007, pages 107-113.

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

[19] A. Welc, , B. Saha and A.-R. Adl-Tabatabi. Irrevocable transactions and their applications. In *SPAA '08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures*, June 2008, pages 285–296.

[20] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris and M. Valero. Atomic Quake: Use Case of Transactional Memory in an Interactive Multiplayer Game Server. In *PPoPP'09: Proc. 14th Symp. on Principles and Practice of Parallel Programming*, Feb. 2009, pages 25-34.

[21] 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*, June 2008, pages 265-274.