QuakeTM: Parallelizing a Complex Serial Application Using Transactional Memory

Vladimir Gajinov*
Barcelona Supercomputing Center
vladimir.gajinov@bsc.es

Ferad Zyulkyarov*
Barcelona Supercomputing Center
ferad.zyulkyarov@bsc.es

Osman Unsal
Barcelona Supercomputing Center
osman.unsal@bsc.es

Adrian Cristal
Barcelona Supercomputing Center
adrian.cristal@bsc.es

Eduard Ayguade*
Barcelona Supercomputing Center
eduard.ayguade@bsc.es

Tim Harris
Microsoft Research UK
tharris@microsoft.com

Mateo Valero*
Barcelona Supercomputing Center
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 work presents *QuakeTM*, a multiplayer game server; a complex real life TM application that was parallelized from the serial version with TM-specific considerations in mind. QuakeTM consists of 27,600 lines of code spread among 49 files and exhibits irregular parallelism and coarse-grain transactions with large read and write sets. In spite of its complexity, we show that QuakeTM does scale, however more effort is needed to decrease the overhead and the abort rate of current software transactional memory systems. We give insights into development challenges, suggest techniques to solve them and provide extensive analysis of transactional behavior of QuakeTM, with an emphasis and discussion of the TM promise of making parallel programming easy.

* Affiliated with Universitat Politecnica de Catalunya

1. INTRODUCTION

Recently, processor manufacturers have executed a right-hand turn away from increasing single core frequency, complexity and heat density to packing multiple processing cores on a chip. In this era of Chip Multiprocessors, efficient parallel programming becomes critical if the new design is to be successful. One of the key technologies that promise to open up practical parallel programming to the masses is Transactional Memory (TM). In this paper, we describe our conclusions, based on code descriptions and examples as well as a through performance evaluation, whether this promise looks realistic. We use a complex gaming application, Quake, as the case study.

As we discuss in Section 2, existing TM benchmarks are either simple data-structure micro-benchmarks, or TM versions of existing concurrent programs, often derived automatically, or semi-automatically from lock-based implementations. For one to really address the issue of whether TM is an enabler technology to make parallel programming easier, we need to start with the serial version of a highly complex application. We wanted to investigate how TM could be used to parallelize a real, complicated, concurrent application. To do this we started from the serial version of the Quake server application and, using OpenMP and transactional memory, we derived QuakeTM, a parallel version which consists of 27,600 lines of code spread in 49 files. It exhibits irregular parallelism and long transactions contained within six different atomic blocks with large read and write sets.

Our idea was not to pursue performance per se, but to discover if it is possible to achieve good results with a coarse grained parallelization approach. This decision was driven by one of the founding principles of TM, to make parallel programming easy by abstracting away the complexities of using locks and achieving performance similar to that of the fine grain lock implementation.

In concurrent work we have been investigating what is involved in making a lock-based implementation of the Quake server perform well using TM. Comparing that approach with the current 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 this further in Section 7.5

In this paper, we present the development process of QuakeTM, comment on the challenges of parallelizing a complex serial application using TM, and provide extensive performance analysis results that provide further insight into the question: “Is parallel programming easier with TM?”

2. RELATED WORK

SPLASH-2 [1] and PARSEC [2] are parallel benchmarks written using Pthreads or OpenMP and are useful in analyzing parallel systems in general, but their applicability to TM systems is limited due to the regular nature of algorithms and short critical sections. When converted to transactional counterparts, these applications are unable to sufficiently stress the underlying TM system.

Lee-TM [5] is a kernel application based on Lee’s routing algorithm implemented in a sequential, lock-based, and transactional manner. The authors claim that it exhibits a fair amount of parallelism, complex contention behavior, and a variety of transaction durations and sizes, even though it has only about 800 lines of code. Unfortunately, we are not presented with a characterization of the serial version, and we lack information about the time spent in critical sections and the sizes of the read and write sets. The performance evaluation, reported by the authors, shows that the transactional version behaves worse than the coarse-grain version which highlights the need for complex benchmarks to stress TM systems.

Delaunay mesh refinement and agglomerative clustering [7, 22, 26] is another kernel application. It comprises [26] of 3200 lines of code and 24 source files and implements 3 atomic blocks which in total protect only 320 lines of code along with the function calls contained inside them. It’s no wonder that the global lock and fine grain lock versions perform equally given the size of critical sections.

STMBench7 [3] is represented by only one synthetic application which can execute four operations on a complex data structure. Varying the parameters, it is possible, to a greater or lesser extent, to stress the underlying STM implementation. STMBench7 is aimed more towards the object-oriented domain and its downside is that the data structure is highly regular, even though it is complex. The size of the application is about 5000 lines of code.

Haskell STM benchmark suite [6], consist 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 widely applicable.

The STAMP benchmark suite [4] consists of eight applications: bayes, genome, intruder, kmeans, labyrinth, ssca2, vacation, and yada. These applications span a variety of domains and runtime transactional characteristics due to differences among applications in transaction lengths, read and write set sizes and amounts of contention. The previous version of STAMP, consisting of three applications: genome, kmeans and vacation, was widely used for evaluation of TM systems. The downside of these applications is the fact that they were manually optimized, with an application level knowledge before hand, which enabled authors to manually implement the optimal number of read/write barriers in the code. As it turns out [9] two common code patterns are repeatedly observed in STAMP: (1) the use of a transaction-aware memory allocator and (2) awareness of shared data structures that remain constant after initialization at the program startup. These two patterns are hard to detect by the compiler which instruments the code with unnecessary transactional barriers. This doesn't help the effort to prove the primary goal of TM which is to make multithread programming easy. 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. [27] library interfaces can remain a useful tool for systems researchers, but application programmers are going to need language and compiler support.

3. QUAKE DESCRIPTION

Quakeworld is the multiplayer mode of Quake 1, the first person shooter game released under the GNU 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 in an infinite loop, where an iteration represents the calculation of a single frame. It blocks on the select system call waiting for client requests. If the requests are present on the receiving port, it starts the execution of the new frame. It is possible to distinguish three stages of the

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, and 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 player's actions, but also the effects induced by these actions, such as when the player hits some object. In such a case it is necessary to determine the applied force, the weight of the object, its shape, the original position, the environment etc. in order to present as realistic view as possible. Thus, server processing is a complex, compute intensive task, and it increases superlinearly with the number of players [13].

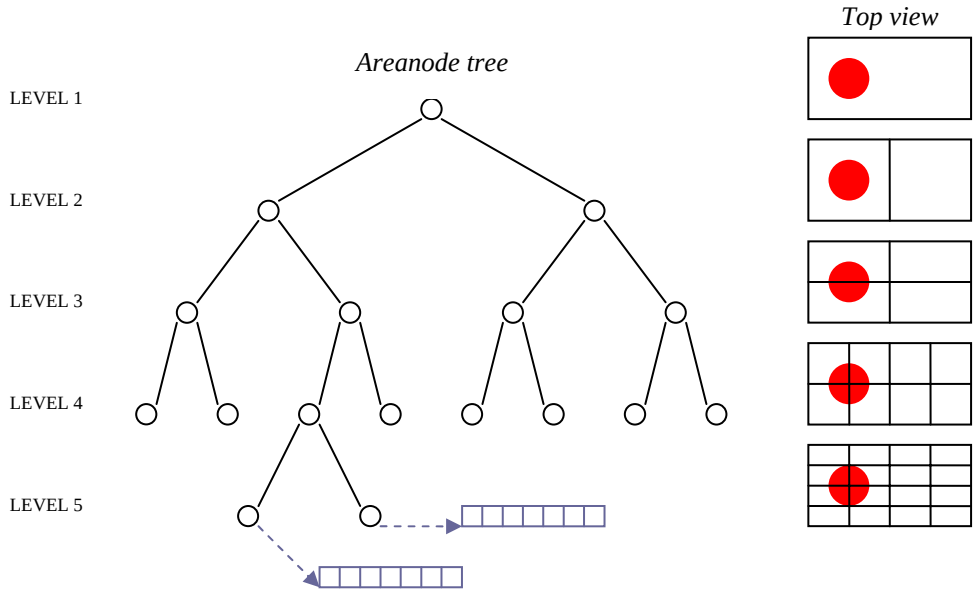


Figure 1: Constructing areanode tree from the BSP map volume. Adapted from [13]

3.1 Map description

A map of the Quake world is represented as a BSP file which holds the binary space partition implementation of the 3D world with all the details relevant to draw and position the objects in the world, such as planes, vertices, nodes, visibility data, texture information, models, brushes etc. [15, 16]. The

level of details contained within the BSP 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 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. Starting from the root, which corresponds to the entire world, the space is divided in two equal parts to form nodes on the next level of the tree. Each division is done using the alternate plane, and after four divisions, an areanode tree with 5 levels, 31 areanodes, and 16 leaves in the lowest level is constructed. Figure 1 demonstrates the building process. The size of the tree can be changed by redefining the maximum depth, but using bigger trees has a minor effect on the performance of the server [14]. The structure of the areanodes has no correlation with the rooms in the world, and likewise the division planes do not correlate with walls or ceilings. All the objects in the game world are either contained within an areanode that is a leaf, or they intersect more than one leaf which means that they cross the division plane, in which case these objects are contained within areanodes on the higher level of the tree. Each areanode has an associated list of objects contained within space defined by that areanode. When an object is moved during gameplay, 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.

3.2 Move execution

Clients influence the gameplay by sending the move command, which specifies various parameters related to the player and his intentions. Those are: (i) angles of the players view, (ii) forward, sideways, and upwards motion indicators, (iii) flags for buttons and jumping and (iv) time to run the command in milliseconds. The move command functionally can be broken into the motion of player's figure and the other interactions the player may initiate with the move command, like firing a gun, or starting an elevator. Using the motion indicators, the origin of the player and the time to run 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 and associate with the move command all the objects contained within this bounding box. 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.

3.3 Shared data

During the gameplay, there are three types of shared data structures: message buffers, 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, holds the updates that reflect the actions of all the players involved in the game session. In the reply stage this buffer is added to the player's buffer for reliable communication which is repeatedly sent in each frame until server gets the acknowledgment from the client that the message has been received.

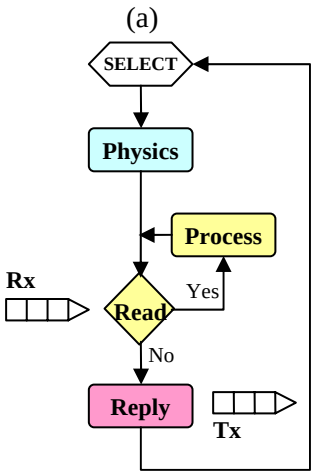
Accesses to the areanode tree are in the form of linked list operations on the object lists associated with each areanode. The access pattern for the request stage has been already covered in the explanation of the move command. A similar pattern is observed in the physics update stage, since physical influences, that 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, all the objects that are touched are updated in a global shared part of the memory. This memory is statically allocated and populated during the server initialization or change of the map. Quake 1 is the first game which used the concept of interpreted core of the game. This means that the essence of the game is coded in a special interpreted language called QuakeC [17]. Using QuakeC, a programmer is able to customize Quake to a great extent by adding weapons, changing game logic and physics, and programming complex scenarios. QuakeC source code is compiled into a byte code kept in a file called progs.dat. During the server initialization this file is loaded into the memory, and appropriate pointers are set according to the layout of the file. It is possible to distinguish the following regions of the file: (1) strings, (2) functions, (3) statements, (4) field definitions, (5) global definitions (6) globals and (7) entities. Once loaded into the memory, specific parts of the engine are accessed using pointers and

statically defined offsets. As an example, we present bellow the usual way to execute PutClientInServer program function, which is called from regular C code to spawn the player into the game:

```
pr_global_struct->time = sv.time;
pr_global_struct->self = EDICT_TO_PROG(sv_player);
PR_ExecuteProgram (pr_global_struct->PutClientInServer);
```

The PR_ExecuteProgram() function acts like an interpreter for the QuakeC code that was previously compiled and loaded into the memory. It determines the function, and accesses the part of the program memory that defines the start address for function arguments and local variables, number of parameters and their size, and the address of the first statement. These are all integer values representing previously mentioned offsets. It allocates the space for parameters and local variables on the internally defined stack and starts the execution of the first statement. Important thing to notice here is that only the entity part of the program memory has to be shared. Other parts can be thread-private to achieve better concurrency.



(b)

```
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 ();
        }
}
```

Figure 2: (a) The frame execution algorithm, (b) The main loop from SV_ReadPackets function.

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

Table 1: Execution breakdown of the sequential server (8 clients)

4. PARALLELIZATION

Parallelization of the Quake server has already been done using Pthreads in [14]. Our work doesn't build on top of the Pthread version, but implements a different parallelization strategy. The goal was to start from the sequential application and parallelize it using OpenMP and transactional memory, in

order to test one of the promises of transactional memory, which is to make multithreaded programming easy. In this section we describe our approach.

The algorithm of the frame execution is given in Figure 2a. An execution breakdown of the serial 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 2b.

Since we are not aware of the number of requests that are pending on the receiving port, we cannot use the *omp parallel for* directive. In such cases it is natural to use a tasking model. The problem which arises here is that the condition for the loop execution is the return value of the function that, at the same time, receives the packet. If there is a packet pending, it will be received in the global buffer and NET_GetPacket() function will return true, which will immediately start processing phase of that packet. Otherwise, if all packets have been processed it will return false and the request processing stage will finish. To enable application of the tasking model, it is necessary to change the way in which the loop execution is controlled, and receive packets outside of the loop. Therefore, 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, making the end of the list marker the condition for the loop exit. There are two justifications for such an approach: (1) it would be more complicated to privatize global receiving buffer once the worker thread starts working on the dequeued task, especially having in mind that this buffer is used throughout different code modules; given our approach privatization is already done and (2) profiling information tells us that the time needed to receive all packets is negligible compared to the time needed to process them. Figure 3 illustrates our approach. The connection related messages are processed as soon as they are received, because they are rare enough and, more importantly, they may involve interactive IO which would serialize transactions in a system which implements irrevocability, or even worse, produce irreversible effects in system that doesn't support IO inside transactions.

```
while (NET_GetPacket ()) {
        // Filter packets

        if (connection related packet){
                SV_ConnectionlessPacket ();
                continue;
        }

        AddPacketToList();
        CopyBuffer();
}

#pragma intel omp parallel taskq shared(packetlist, ...)
 {
while (packetlist != NULL) {
        #pragma intel omp task  captureprivate(packetlist)
        {
                NET_Message_Init(..);
                // check for packets from connected clients
                for (i=0, cl=svs.clients ; i<MAX_CLIENTS ; i++,cl++) {
                        // Do some checking here
                        SV_ExecuteClientMessage (cl);
                }
        }

        packetlist = packetlist->next;
}
```

Figure 3: Pseudocode for the parallelized request processing stage.

```
void SV_RunCmd (usercmd_t *ucmd)
{

T1      TRANSACTION
                . . .
                PR_ExecuteProgram (pr_global_struct->PlayerPreThink);
                . . .
                SV_RunThink (sv_player);
                . . .
                // prepare pmove structure
        TRANSACTION_END
        . . .
T2      TRANSACTION
                AddLinksToPmove ( sv_areanodes );
        TRANSACTION_END

T3      TRANSACTION
                PlayerMove ();
        TRANSACTION_END
        . . .
T4      TRANSACTION
                // link into place and touch triggers
                SV_LinkEdict (sv_player);

                // touch other objects
                . . .
                PR_ExecuteProgram (ent->v.touch);
        TRANSACTION_END
}
```

Figure 4: Pseudocode for the function that executes the move command with transactional block markers.

To synchronize the request processing between threads, our idea was to execute the SV_ExecuteClientMessage() function inside a transaction. This would be a coarse-grained solution since, for the whole request processing stage, we would have only one transaction block. Since the number of retries[†] was overwhelming with more than 98% of all transactions aborting, we were forced to switch to a medium-grained approach and break the one big transaction into six smaller, but still substantially large atomic blocks.

SV_ExecuteClientMessage() parses the request message and executes commands. The most common is a move command which is executed in the SV_RunCmd() function, presented in the Figure 4 together with the transactional block markers. Physics for the client entities is updated in the request processing stage, rather then in the physics update stage, by executing PlayerPreThink program in the PR_ExecuteProgram() function call, at the beginning of the SV_RunCmd() function. Then the SV_RunThink() function executes a "thinking" code for the client. 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. For example if some object falls from the high altitude, the server will need more than one frame to simulate its fall. Along with *pmove* (player move) structure initialization, these actions form the preparation for the actual move execution, and can be contained inside one transactional block. The server continues execution with the AddLinksToPmove() function which is used to determine which entities could be affected by the current move command. Starting with the origin of the player, the areanode tree is traversed and linked objects from the areanode lists observed to check if their position falls into the maximum affected area. Those that do fall into this area are added to the *pmove* structure for further processing in the PlayerMove() function. For the player entity and each entity from the *pmove* entity list, a model box is assigned, and the trajectory is followed from the player's original position to its potential destination, using the parameters from the move command extracted from the received message. If the player's model box clips a model box of the other entity moving along the trajectory line, that

[†] In this paper retry is equivalent to abort, following the Intel nomenclature. It is to be distinguished from the Haskell retry construct [28].

means there is collision between them. Depending on the various parameters of the collided objects and the environment that surrounds them (air, water, solid area, etc.) the final position of the player is calculated. Based on this calculation the player object will be later re-linked in the areanode tree, in the SV_LinkEdict() function call. Since the areanode tree is not traversed during PlayerMove() call it is safe to define transactional blocks in the way presented in Figure 4.

5. EVALUATION

For testing purposes, we have developed an automatic trace client called *TraceBot*. We have used the regular client code, and changed its structure to implement a state machine which controls the client's behavior. We also had to implement certain changes on the server side to be able to synchronize the client's actions with the server response. Namely, each map has a number of spawn points where clients start after joining the game, or restart after being killed. In order to execute the correct trace we have to send the spawn spot information to the TraceBot. On the client side, after TraceBot starts, we have to recognize the right moment to start reading the trace file. We introduced a new string command in the protocol, and when it is parsed from the server packet, we take the spawn spot information, and start randomly one of the traces for that spawn spot. From that point on, TraceBot is just 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 whose 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 gameplay actions.

We run the server on one machine, and the clients on the other, to simulate the real game environment, since network latency and bandwidth are not critical [14]. The server and client's frame rates are synchronized and set to 100 ms 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 of the Intel STM C/C++ compiler [10, 11, 12]. The underlying STM implementation is an extended version of the McRT-STM system [8]. The compiler implements both optimistic and pessimistic concurrency control, and provides single lock atomicity semantics and weak atomicity guarantees. Serial execution mode is also 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

In order to compare and test various aspect of STM performance, we collect results for four different configurations: (1) sequential, (2) global lock, (3) global lock with STM and (4) STM. The global lock with STM implementation is used to measure the overhead of running a transaction (starting the transaction and bookkeeping) since the global lock prevents any possibility of the transaction abort. For the parallel setups we vary the number of threads from 1 to 8. We also vary the number of clients from 1 to 16, and run each test five times to get the average result. Each test runs for 2000 frames which translates to 200 seconds of real time. The results are collected for the last 1000 frames in order to avoid server initialization time and player connection times.

We measure the number of cycles between two events using the *rdtsc* instruction and translate that value into milliseconds. We present the results and charts for the request processing stage only, since it is the most complex stage in the frame execution and the most challenging, and leave the parallelization of other two stages for the future work.

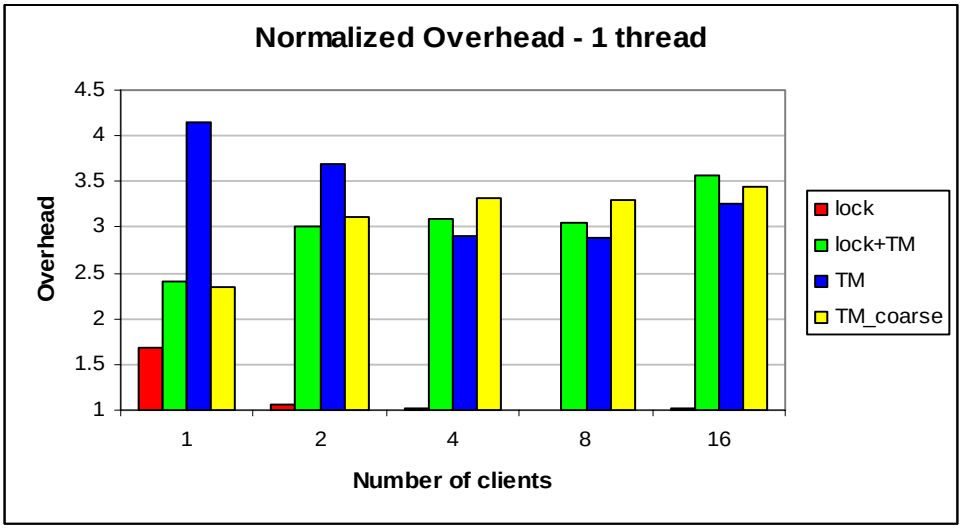


Figure 5: Overhead of parallel implementations normalized against the results of the sequential server (tm_coarse - only one transaction which processes the whole packet).

Figure 5 shows the overhead of the parallel implementations normalized against the results of the sequential server for non-optimized code[‡] and for the parallel section which is in this case the request processing stage. We consider the results for one thread only, to avoid contention in parallel designs. Since there is no contention, the lock version introduces almost no overhead. On the other hand, the overhead of the *lock+TM* version goes from 2.4 times for one client to 3.6 times for sixteen clients and for the transactional version from 2.9 for four clients to 4.2 for one client. These results correspond with 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-grain 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.

Figure 6 shows the comparative performance of all three parallel configurations for different numbers of connected clients. As expected, the global lock version doesn't scale and the transaction overhead remains approximately 3x-4x. The transactional version doesn't scale until the workload becomes sufficient, which happens with eight connected clients. When we run the application with 16 clients, then we start to notice a considerable speedup. Figure 7a gives a better view of this case. The values are normalized to a single thread execution time. The speedup for eight threads is 1.51 which is a

[‡] All configurations were compiled with optimization level 0. Higher optimization levels introduce problems with function call annotations and transaction serialization.

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 7b shows the scalability of the transactional Quake server running with 16 clients. It is obvious that the TM version scales, but it still performs worse than the global lock version.

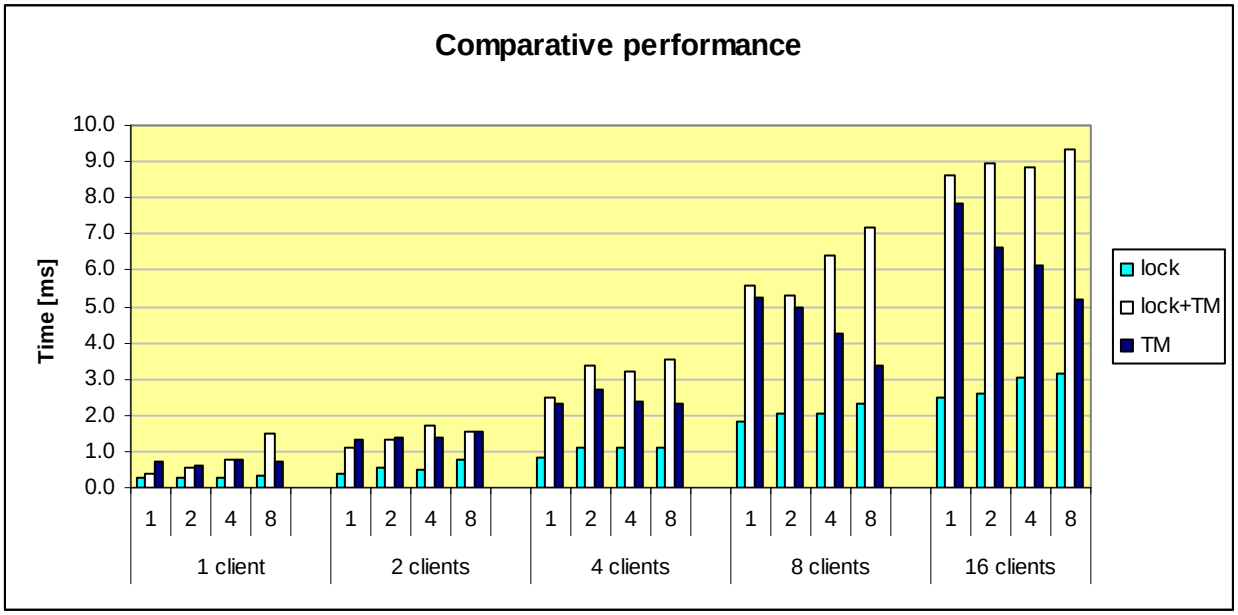
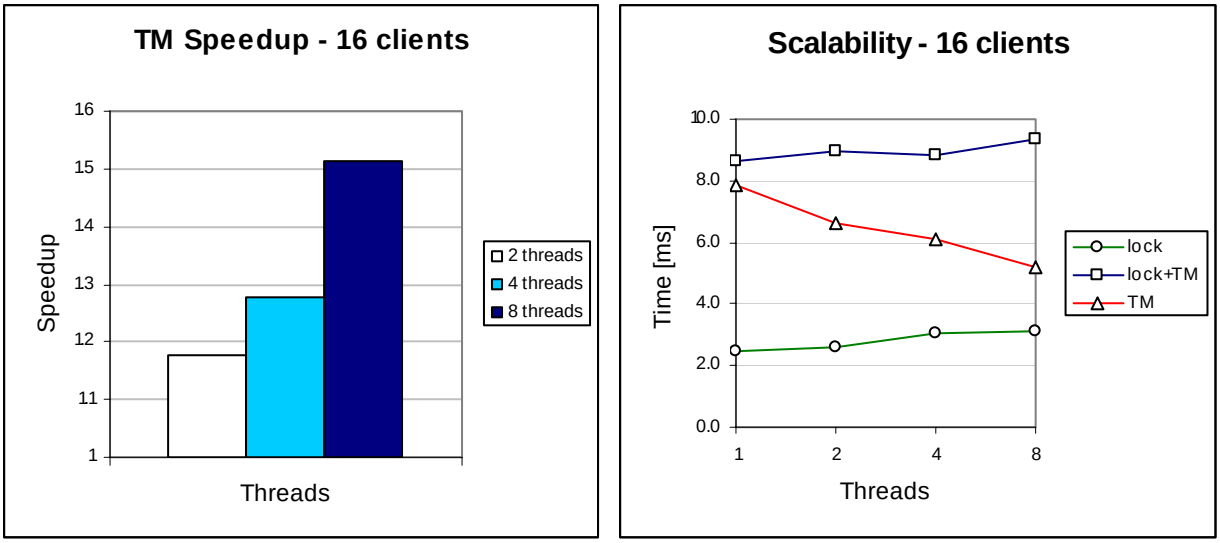


Figure 6: Comparative performance of parallel configuration
(Y-axes represent the average time to execute request processing stage)



(a) (b)

Figure 7: Transactional server running with 16 clients: (a) Speedup, (b) Scalability.

Clients	Transactions	Retries	Retry rate [%]	Serialized transactions		Mean [KB]	Max [KB]	Total [MB]
1	34683	0	0.0	2.2	Reads	3.5	127	114
					Writes	0.5	18	16
2	80031	3692	4.4	4.8	Reads	4.1	1247	304
					Writes	0.6	204	43
4	147043	21990	13.0	11.2	Reads	5.2	1593	734
					Writes	0.7	245	99
8	293326	125138	29.9	28.2	Reads	6.5	1778	1805
					Writes	0.8	237	233
16	351379	221062	38.6	59.8	Reads	7.0	2397	2358
					Writes	0.8	309	280

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

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 the performance perspective, is the transaction abort rate. In the case of sixteen connected clients, 38.6% of transactions retry introducing a significant amount of wasted work. There are examples when a transaction aborted 136 times before it eventually committed. This leads not only to the waste of processor cycles to re-execute the transactional code. During an execution with sixteen clients, a single threaded server reads 1.4 GB and writes 215 MB of data in total, while the server running with 8 threads reads 2.34 GB and writes 280 MB of data. Table 2 also shows that even though the mean value of the read set is about 7 KB there are cases when it grew to 2.4 MB. This is an important factor which could stress the design of any hardware transactional memory.

In order to study the above mentioned problems, we compare the results for three different TM cases: (1) *TM_coarse* - coarse grained transaction case, having only one transaction which processes the whole packet (surrounding the SV_ExecuteClientMessage() function from Figure 3), (2) *TM* - middle grained TM implementation, which is the case we studied so far and (3) *TM_small* - the same implementation as in the second case, only running an extremely small map to increase contention. Figure 8 shows the scalability of these three cases, for eight and sixteen connected clients, and Table 3 presents the corresponding transactional statistics. It is clear that the performance is strongly influenced by the

transaction abort rate. For example, *TM_small* setup scales well with eight connected clients, when the abort rate is still bearable, but with sixteen clients, when the abort rate is 58%, the performance starts to suffer and scalability is lost. In the *TM_coarse* case, we see that almost all transactions abort, leading to 98.6% abort rate with sixteen connected clients, which in combination with extremely large read and write sets, results in a poor performance.

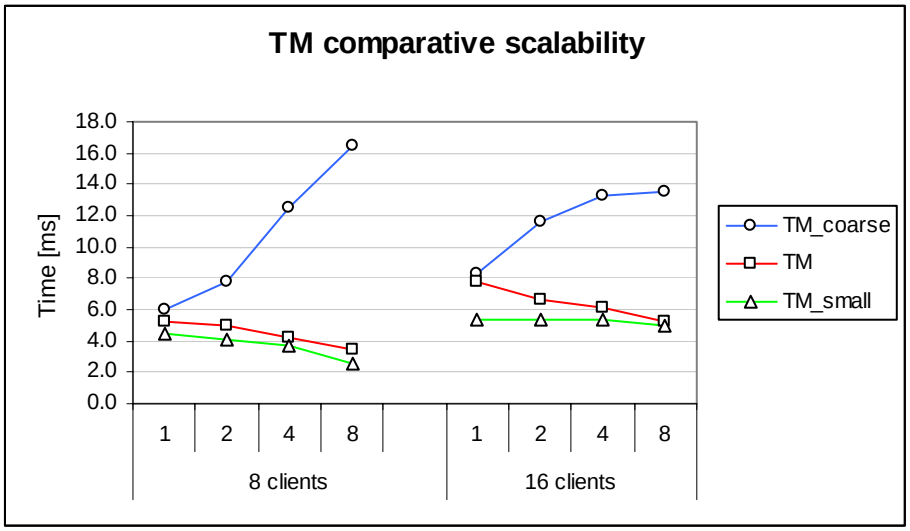


Figure 8: Scalability of coarse grained (TM_coarse), medium grained (TM) and mediume grained implementation running with the small map (TM_small).

	Clients	Transactions	Retries	Retry rate [%]	Reads [mean] [KB]	Writes [mean] [KB]
TM	8	293326	125138	29.9	6.5	0.8
	16	351379	221062	38.6	7.0	0.8
TM_small	8	145448	139204	48.9	8.4	1.8
	16	206568	285118	58.0	9.4	1.7
TM_coarse	8	15979	632175	97.5	121.6	102.6
	16	21772	1484730	98.6	100.5	57.2

Table 3: Transactional statistics for TM, TM_small and TM_coarse configurations running with 8 threads.

7. DISCUSSION

In this section we describe the parallelization and language design issues we faced while porting the Quake server, and present our ideas how to improve the TM programming experience.

7.1 Removing the Object from the Read Set

If we take another look at Figure 4, and recall the function algorithm, we can identify the source of the extremely high abort rate in the TM-coarse case shown in Table 3. Let's assume that we are running with two threads, and that transactions T1 from thread 1 and T2 from thread 2 are simultaneously processing packets from two different clients which are close to each other in the gameworld. Both transactions will traverse the areanode tree in the function AddLinksToPmove(), and latter on, most probably change the list structure for the same node in the tree, either by inserting or removing some objects, in the SV_LinkEdict() function. Since in both functions, the whole list for the particular node is traversed, all list elements will be added to the read sets of both transactions. Therefore any change to the list will abort the other transaction. For the execution with eight threads this leads to a scenario similar to livelock, where transactions abort each other, even though they eventually commit. We think that a solution for this kind of scenario could be to have an explicit language construct, *TM_Unread,* which could be used to remove an object from the transaction read set, similar to the solution suggested by Sonmez et al. [19]. Since such a construct could easily break the correctness of the program, it would be the programmer's responsibility to use it in a correct way. Furthermore, we think that this should be a performance enhancement feature, targeted for expert programmers, and not something used by general programming community, following the streamline of the *tm_waiver* language construct [11].

7.2 Parallelization Issues and ReachPoints

One of the main goals of this work was to test the holy grail of transactional memory [21], the promise of making multithreaded programming easy. We had no prior knowledge of the application itself, so we took some time to understand the code. We have identified the parts of 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, which is normal, having in mind that both OpenMP and STM are new programming models under constant development. 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, especially unnecessary one, leads to bad performance. Even though we dedicated significant time to manually identify global data which could be thread-private, for unmanaged code written in a sequential programming style, where vast amount of data is global, like in our case, it was not enough. The whole work took ten men-months to finish, and we spent the last two months, once we had testing environment ready, trying to boost performance. In the process, we came up with a solution, which we call *ReachPoint*s, that helped us identify the rest of the global variables which could be thread private, and discover the problems that arose from TM cache-line granularity conflict detection implementation.

```
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 9: ReachPoints: Simple solution for discovering conflicting regions of the transactional code.

The *ReachPoints* solution, presented in Figure 9, consists of allocating an array of counters for each thread, taking in account cache line granularity (x*16 elements 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 [20] referred to as false sharing [9], 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 A, under the cache line granularity conflict detection system, a read-only variable B

is also causing conflicts, since the writes to A are treated as writes to B. When we used cache padding for such cases, we noticed a significant performance improvement, due to the 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.

7.3 Drawbacks of Flattening the Nested Transaction

In Section 4, when we were explaining our parallelization method, we said that we were unable to use coarse grain transaction for the request processing stage due to an extremely high abort rate. Using *ReachPoints* we were also able to prove our assumption that areanode operations, executed in the functions AddLinksToPmove() and SV_LinkEdict(), are the main cause for aborts. Before we switched to the medium grained implementation, we tried to insert nested transactions in the same way shown in Figure 4, but there were no improvements in the performance or abort rate. This is the consequence of the closed nesting implementation of the Intel compiler, which flattens nested transactions, causing aborts of the outermost transaction and resulting in the same performance as before. We believe that a different nesting implementation, i.e. one which wouldn't publish the read set of the nested transaction to the outermost transaction, leading to partial aborts [23, 24, 25], would be more useful for programmers.

7.4 Things We Cannot See

At times, not having the source code of the Intel STM compiler created issues. For example, we were unable to discover further details regarding the overhead of transactions and to measure the time spent for starting a transaction, bookkeeping and executing instrumented reads and writes. We are in contact with Intel developers for the possibility of including some of those statistics in the next version of the compiler

Also we cannot conclude what percentage of aborts is caused by false conflicts between addresses from different cache lines [9], which are the consequence of the transaction record aliasing imposed by

the limited size of the ownership table. In the Intel compiler implementation, the hash function utilizes 14 bits out of a 32 bit address to generate a hash value (16K entries). If the program is running with eight threads, the possibility of the false conflict when each thread touches 10 cache lines in an uniform distribution is 15.7% and in a case of 25 cache lines it grows to 66.8%. Given the size of the read and write sets from the Table 2, we suspect this to be a significant cause for aborts in QuakeTM. In such case, in order to decrease local-induced conflicts (if either of the two conflicting addresses is thread-local) it would be necessary to avoid instrumentation of thread-local memory accesses.

7.5 Comparing Two Parallelization Approaches

In concurrent work, we have used fine grain lock implementation of the Quake server to produce a transactional version by converting locks to transactions [29]. Comparing these two parallelization approaches, we can conclude that the solution presented in this paper, one that is derived from the serial implementation, results in a coarser grained parallelization with atomic blocks and read and write sets that are an order of magnitude bigger than in the lock-converted version. Moreover, the scalability obtained is competitive with the lock-converted version. Finally, the code is more structured as a result of not having to deal with the problems inherited from the unstructured use of locks. Given these observations, we believe that this work corresponds with the intended use of transactional memory and presents the way the regular programmer might use TM to parallelize applications. A detailed comparison of QuakeTM with the lock-converted Quake is out of the scope of this paper. Here, we concentrate on introducing and analyzing the performance of QuakeTM.

8. FUTURE WORK

Beside the fact that negligible time is spent in execution of 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 areanode lists, in order to decrease the abort rate and hopefully enable use of coarser transactions. The source code for QuakeTM will soon

be publicly available, and we encourage TM implementers to download and use the application to test their TM systems.

9. CONLCUSION

In this paper, we have introduced QuakeTM, the first complex real-world TM application that was transactionalized from a serial version of the Quake application. We have made a detailed description of the tranactification process, and provide extensive analysis of performance, isolating the overhead of the STM implementation. We have shown that even though it scales, the TM implementation still falls behind the global lock version. As a result, we were surprised by the amount of programmer time investment, wherever appropriate we have also commented on the challenges involved in the paper. TM is touted to make parallel programming easier; our QuakeTM experience stresses the importance of tool support (compiler, debugger, runtime) to realize this goal in the future.

REFERENCES

[1] S. C. Woo, M. Ohara, et al. *The SPLASH2 Programs: Characterization and Methodological Considerations*. In ISCA 1995.

[2] C. Bienia, S. Kumar, et al. *The parsec benchmark suite: Characterization and architectural implications.* Tech. Rep. TR-811-08,, Princeton University, 2008.

[3] Rachid Guerraoui, Micha l Kapa lka, and Jan Vitek. *STMBench7: A benchmark for software transactional memory*. In EuroSys '07: Proceedings of the 2nd European Systems Conference, pages 315–324. ACM Press, March 2007.

[4] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun , *STAMP: Stanford Transactional Applications for Multi-Processing* , In IISWC '08

[5] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. *Lee-TM: A Non-trivial Benchmark for Transactional Memory*. In ICA3PP 2008.

[6] C. Perfumo, N. Sonmez, et al. *Dissecting transactional executions in Haskell*. In TRANSACT '07

[7] M. Kulkarni, K. Pingali, et al. *Optimistic parallelism requires abstractions.* In PLDI '07

[8] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. *McRT-STM: a high performance software transactional memory system for a multi-core runtime*. In PPoPP 2006.

[9] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, Hsien-Hsin S. Lee, *Kicking the tires of software transactional memory: why the going gets tough*. In SPAA 2008.

[10] 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 2006.

[11] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, Xinmin Tian, *Design and Implementation of Transactional Constructs for C/C++*, in OOPSLA08

[12] Adam Welc . Bratin Saha, Ali-Reza Adl-Tabatabai , *Irrevocable transactions and their applications*. In SPAA 2008.

[13] A. Abdelkhalek, A. Bilas, and A. Moshovos, *Behavior and performance of interactive multi-player game servers*. In Proc. of the 2001 International IEEE Symposium on Performance Analysis of Systems and Software (ISPASS01), Nov. 2001.

[14] A. Abdelkhalek, A. Bilas, *Parallelization and Performance of Interactive Multiplayer Game Servers*, In IPDPS 2004

[15] Fuchs, Henry., et. al. *Near Real-Time Shaded Display of Rigid Objects*, Computer Graphics, 17(3), 65-69.

[16] Max McGuire, *Quake 2 BSP File Format*, http://www.flipcode.com/archives/Quake_2_BSP_File_Format.shtml

[17] *QuakeC*, http://en.wikipedia.org/wiki/QuakeC

[18] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, Ali-Reza Adl-Tabatabai, *Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language*, CGO '07

[19] Nehir Sonmez, Cristian Perfumo, Srdjan Stipic, Adrian Cristal, Osman S. Unsal, and Mateo Valero, *UnreadTVar: Extending Haskell Software Transactional Memory for Performance*, in 8th Symposium on Trends in Functional Programming (TFP 2007)

[20] C. Zilles and R. Rajwar. *Implications of false conflict rate trends for robust software transactional memory*. In IISWC'07.

[21] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*. In ISCA 1993.

[22] M. Herlihy and E. Koskinen. *Transactional boosting: a methodology for highly-concurrent transactional objects*. In 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 207{216.

[23] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit , Michael M. Swift, David A. Wood, *Supporting nested transactional memory in logTM*, ASPLOS 2006

[24] J. Eliot B. Moss, *Open Nested Transactions: Semantics and Support*, http://www.cs.utah.edu/wmpi/2006/final-version/wmpi-posters-1-Moss.pdf

[25] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[26] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. *Delaunay Triangulation with Transactions and Barriers*, Benchmarks track, IEEE Intl. Symp. on Workload Characterization , Boston, MA, IISWC 2007.

[27] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott, *Capabilities and Limitations of Library-Based Software Transactional Memory in C++*, TRANSACT 2007.

[28] Tim Harris, Keir Fraser, *Language support for lightweight transactions*, OOPSLA '03

[29] Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, Mateo Valero, *Atomic Quake: Use Case of Transactional Memory in an Interactive Multiplayer Game Server*, To Appear in Symposium on Principles and Practice of Parallel Programming (PPoPP), February 2009