Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games

Daniel Lupei* Bogdan Simion* Don Pinto‡ Matthew Misler* Mihai Burcea*
William Krick* Cristiana Amza*

*Department of Electrical and Computer Engineering, University of Toronto, Canada
‡Department of Computer Science, University of Toronto, Canada
*{daniel, bogdan, mislerma, burceam, williamkrick, amza}@eecg.toronto.edu,
‡donpinto@cs.toronto.edu

Abstract

In this paper, we study parallelization of multiplayer games using *software* Transactional Memory (STM) support. We show that the STM provides not only ease of programming, but also *better* performance than that achievable with state-of-the-art lock-based programming, for this realistic high impact application.

For this purpose, we use a game benchmark, SynQuake, that extracts the main data structures and the essential features of the popular game Quake. SynQuake can be driven with a synthetic workload generator that flexibly emulates client game actions and various hot-spot scenarios in the game world.

We implement, evaluate and compare the STM version of SynQuake with a state-of-the-art lock-based parallelization of Quake, which we ported to SynQuake. While in STM-SynQuake support for maintaining the consistency of each complex game action is automatic, conservative locking of surrounding objects within a bounding box, for the duration of the game action is inherently needed in lock-based SynQuake. This leads to higher scalability of STM-SynQuake versus lock-based SynQuake, due to a higher degree of false sharing in the latter. Task assignment to threads has a second-order effect on the scalability of STM-SynQuake, due to its impact on the application's true sharing patterns. We show that a dynamic locality-aware task assignment to threads provides the best trade-off between load balancing and conflict reduction.

EuroSys'10, April 13–16, 2010, Paris, France.

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

General Terms Performance

Keywords Software Transactional Memory, multiplayer games, synchronization, scalability, load balancing

1. Introduction

Transactional Memory (TM) is an emerging paradigm for parallel programming of generic applications, with a goal to facilitate programmer-friendly use of the plentiful parallelism available in chip multiprocessors. The main idea is to simplify application programming in parallel environments through the use of transactions.

Many commercial and research prototypes for supporting TM in software have been introduced recently, including Intel's freely available STM compiler [Intel Corporation. 2008], RSTM [Marathe et al. 2006], TL2 [Dice et al. 2006], etc. However, STM uptake from the wider programming community has been slow for two main reasons. First, demonstrable efforts from the research or commercial communities towards parallelizing realistic applications using STM have been scarce. This is at odds with the claim that TM makes parallelization easy. Second, existing STM-based parallelizations typically perform substantially worse than simple single-mutex based implementations.

In this paper, we introduce the first case study of a realistic high impact application where STM support provides both ease of programming and *better* performance than that achievable with state-of-the-art lock-based programming. Specifically, we study parallelizing multiplayer online game server code on a game benchmark modeled after Quake 3. Towards this, we leverage an existing *software* TM library, *libTM* [Lupei et al. 2009], that can be used in conjunction with generic C, or C++ programs.

Parallelization of multi-player game code for the purposes of scaling the game server is inherently difficult. Game code is typically complex, and can include use of spatial data

structures for collision detection, as well as other dynamic artifacts that require conservative synchronization. The nature of the code may thus induce substantial contention due to false sharing, as well as true sharing between threads, in a parallel lock-based game implementation.

Substantial false sharing can occur in a parallel implementation of the popular first person shooter game Quake [Abdelkhalek and Bilas 2004], as follows. Each Quake player action usually includes dynamically evolving sub-actions; a person may move while shifting items in their backpack, throwing an object at a distance, grabbing a nearby object, and/or shooting, which together constitute a single player action. Since the terrain within the potentially affected area may contain mutable objects, all sub-actions need to be processed together as an atomic, consistent unit for the purposes of collision detection with other player actions.

In Quake 3, each player action is thus processed based on a bounding box estimate, e.g., given as a sphere, around the initial player position, for the possible range of their intended action, as shown in Figure 1. In a parallel lock-based server code implementation [Abdelkhalek and Bilas 2004], this translates into eagerly acquiring ownership of all potentially affected objects of the game map within this bounding box, before processing the action. On the other hand, as we can see from Figure 1, the actual player trajectory can affect a very restricted area of the game map, due to collisions. Thus, conservative locking induces unnecessary conflicts, by locking more objects than necessary, and holding these locks for longer periods than needed.

In contrast, with Transactional Memory support, a player action can be split into segments, or its constituent sub-actions. Collision detection can be performed dynamically, for each segment, hence more accurately, as the avatar encounters various obstacles, which potentially change its direction of movement; this is shown by the intermediate steps in Figure 1. The atomicity and consistency of the whole player action is automatically provided by the underlying transactional support. STM support thus results in reduced false sharing overall, in terms of both number of conflicts and duration of conflicts.

Aside from false sharing, game code scalability is also orthogonally affected by the true sharing induced by player actions on objects located at the boundary between partitions of the game world assigned to different threads. Consequently, task to thread assignment for load balancing purposes needs to be done in a locality-aware fashion in the game world; this reduces the number of boundaries, hence the potential conflicts on boundary objects.

In order to facilitate experimentation, and to study a range of game genres and in-game scenarios, we have developed an in-house game benchmark, called SynQuake. SynQuake is a 2D version of the Quake 3 game. We use the same game data structures as Quake 3, and a similar server frame structure. Specifically, we use the Quake 3 area node tree as a

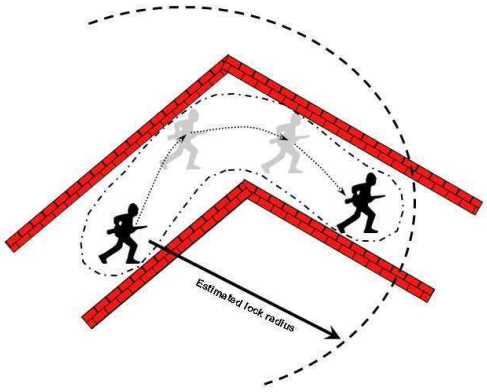


Figure 1. Processing player move: Radius-based locking around player position

standard game code spatial data structure facilitating storing and retrieval of the location and attributes of game objects on the game map. The main reason why SynQuake is a 2D game is that Quake 3 comes with inadequate test cases, due to the small game map available, which does not allow for scaling to large numbers of players. Previous work on Quake code parallelization was forced to run either with a hyper-crowded map [Abdelkhalek and Bilas 2004], or with a maximum of 8 players [Zyulkyarov et al. 2009]. We synthetically create a variety of meaningful game scenarios and associated test cases on a simpler 2D game map. SynQuake can be driven by either human players, or robot players, and can emulate synthetically generated quest scenarios, i.e., hot-spots in the game world to facilitate experimental evaluation.

With SynQuake's default settings, where we model only 2D collisions with obstacles, we showcase the worst case scenario for the performance of an STM-based parallelization of *any* realistic multiplayer game. Any commercial game would have substantial additional physics computation in 3D e.g., gravity, 3D explosions, etc. As in Quake, this additional computation would likely occur only on *non-TM* data, and would almost completely hide the TM overhead that we currently have. Hence, intuitively, our favourable performance comparison for STM is very likely valid for many other multiplayer games. Furthermore, with SynQuake's synthetic workload generator, and adjustable game code settings, we can cover a larger range of games, not just Quake.

Our detailed comparative performance evaluation shows that the main factor affecting overall scaling is the effect of the false sharing inherent in the parallelization scheme. Since the STM substantially reduces the degree of false sharing, we experimentally show that the scaling of STM-based SynQuake is better than that of lock-based SynQuake. More importantly, the STM also achieves better overall performance by a factor of 1.34x on average at eight server threads, in all scenarios involving a minimum of physics computation.

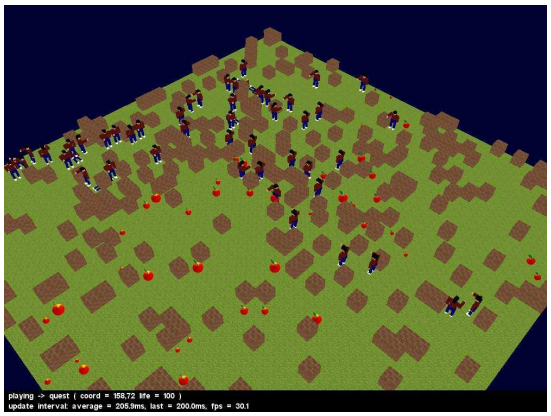


Figure 2. Screen shot of SynQuake. Blocks represent "walls"/"obstacles", while resources are represented by apples.

The degree of spatial false sharing in the STM is a function of the consistency unit granularity. Our evaluation shows that increasing the consistency granularity for STM reduces its inherent access tracking overhead significantly. There is, however, a trade-off between reducing this overhead and increased false sharing for STM in high contention scenarios. Overall, an object-based consistency unit performs best by avoiding false sharing, while providing significant overhead reduction compared to using word-level granularity.

We explored several load balancing techniques, which range from a policy that fully focuses on equally distributing tasks among threads, while sacrificing locality, to an approach that minimizes true sharing but offers no guarantee with respect to load distribution. We observed that having a locality-aware thread assignment policy is critical for minimizing true sharing. The load balancing scheme allows for significant improvements in scalability only for STM-SynQuake, and has little effect for a lock-based parallelization, due to the dominant impact of false sharing in the latter.

The rest of the paper is organized as follows. In Section 2 we outline the design of our game benchmark, SynQuake, as well as the architectural features and relevant data structures. We then examine the synchronization and load balancing related challenges of parallelizing SynQuake in Section 3. Section 4 presents the experimental results comparing the performance of the lock-based and STM-based versions of SynQuake, followed by a discussion of related work in Section 5. Section 6 concludes our paper.

2. Environment: SynQuake game

Multiplayer Online Games (MOGs) are not only difficult to build, but also very expensive to maintain and administer. Except in isolated cases, such as, the well-known first person shooter game Quake, popular MOGs do not share their server code. Furthermore, only a few game scenarios are available with Quake 3, and the game map used in a previous study on Quake parallelization [Abdelkhalek and Bilas 2004] is too small; this results in excessive player crowding when scaling the player population to drive a larger number of server threads.

To solve these problems, we have built a 2D version of Quake, called SynQuake. The simplified 2D object representation facilitates generation of game maps, manually or synthetically. This representation results in reduced game physics computation compared to the 3D Quake. However, physics computation either i) consists of complex mathematical formulas, such as, physical laws for gravity, explosions, or ii) is performed on immutable objects, like obstacles. Quake game physics computation does not involve shared transactional data, and is not interesting for the purposes of game parallelization.

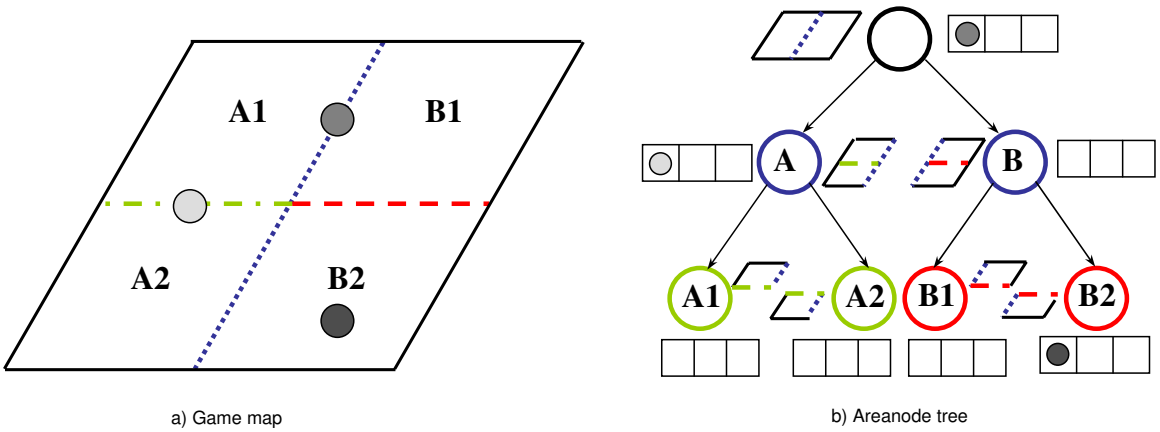
SynQuake models three types of game entities: players, resources (represented by apples) and walls. A typical game map for SynQuake is presented in Figure 2. Each game entity is defined by its position on the game map and by a set of attributes specific to its type. For example, besides its position on the game map, a player is described by its life or health level, its speed and direction. Resources are similar to Quake 3 powerups, such as, weapons and health packs. An attack decreases a player's life, while consuming a resource increases it.

As part of game play, as in Quake 3, players perform short range actions, like moving and consuming resources, fighting with other players, or long range actions, such as, simultaneously moving and shooting. Each of these actions can cause conflicts between different threads processing player actions concurrently. For example, conflicts occur when two players try to move to the same spot, or one player gets attacked while consuming a resource.

To simulate areas of high interest in the game, and the associated pattern of players flocking to a particular area of the map, we have added *quests*, which attract players towards that area with a high probability. These correspond to standard areas attracting players existing in Quake 3, and also in strategy games, such as a camp site, hidden treasure, weaponry location, and health areas. Complex game scenarios can be created in SynQuake by varying the distribution of quests in time and space.

We use the same data structures as in Quake 3, and a similar server frame structure. Furthermore, SynQuake accurately represents network and system features of a game server. In order to create synthetic workload scenarios, players can be driven by a simple AI algorithm that has players moving with high probability towards a quest, if one is present, eating if hungry, fighting with other players, or fleeing if being chased by a stronger opponent.

In summary, aside from almost identical data structures and server processing frame, our game accurately models



a) Game map b) Areanode tree

Figure 3. Areanode tree structure. Each node maintains a list of game entities fully contained in its corresponding game region.

Quake 3 player behavior, although the object representations are different.

2.1 Game architecture and data structures

Our server stores a representation of the game world, and all the game entities, as in Quake 3. We further copied the Quake 3 areanode tree (see Figure 3b), a spatial data structure that facilitates an efficient search for all entities that a player interacts with in any given action. We use the same way of storing information about game objects within the tree nodes.

The areanode tree is a binary space partitioning tree, where each node represents a specific region of the game map. Similar representations using a form of binary space partitioning, i.e., BSP-trees, quad-trees, are present in other games. The tree is constructed by recursively dividing the map into sub-units, starting with the root node, corresponding to the entire game world. Nodes on subsequent levels of the tree are created by splitting the region corresponding to the parent node along its median segment. The splits are performed alternately along the x and y axes, until a predefined tree depth (or the maximum split granularity possible) is reached. The leaves in the areanode tree form a grid of equal-sized regions located on the game map. For the rest of this paper, we will use the terms *grid unit* and *tree leaf* interchangeably.

Each entity on the map is maintained inside the finest-grain tree-node whose corresponding region completely overlaps it. This translates into leaf nodes maintaining objects that are fully contained inside grid units, while placing the rest of the entities in the common ancestor of all the grid units they overlap.

Figure 3 shows a map for which game objects are indexed using a two-level areanode tree. In Figure 3a, we have the resulting grid units corresponding to each leaf, while Figure 3b presents the entire tree structure resulting from splitting the root node vertically and its children horizontally. The map is populated with three objects: one completely situated inside leaf B2 and correspondingly placed in B2's list inside the tree; a second one found on the border between grid units A1 and A2 and as a result maintained in A's list; the last entity is located on the border between A1 and B1 and kept inside the root node, since its corresponding region is the smallest one that completely overlaps it.

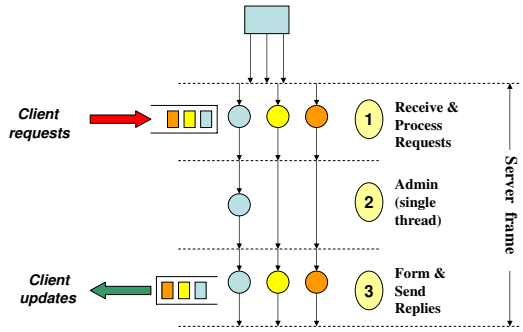


Figure 4. SynQuake: server frame structure and component stages.

Since entities migrate during game play (e.g., players move, apples get respawned after being consumed), the entity lists maintained in the areanode tree must be updated accordingly, in order to reflect the new positions of game entities. This accounts for the most significant source of contention among processing threads when parallelizing the game server.

2.2 Game server structure

In the Quake 3 server code, server processing consists of three stages: world physics update, request processing and reply processing. Following the same design, the SynQuake server loops through three stages which form a server frame or iteration. These stages are: request processing, administrative tasks and reply processing (see Figure 4). In the first stage, the server receives requests from clients (players). A client request gets processed by the worker thread

who is responsible for handling that particular client. The thread assignments are established according to a load balancing policy and updated in the administrative stage. The administrative stage involves rebalancing the load according to a specific policy. The original Quake game was single-threaded and thus didn't need any load balancing, while the parallelization of Abdelkhalek et al. [Abdelkhalek and Bilas 2004] used a naive round-robin assignment of players to threads. In contrast, we use a set of policies which are aimed at reducing synchronization costs and thus improving performance, which we discuss in Section 3.5. Finally, the server threads send updates to their assigned clients in the reply phase.

3. Parallelization of SynQuake

Similar to an existing parallel implementation of Quake [Abdelkhalek and Bilas 2004], in SynQuake, parallelization is performed at the granularity of individual stages, while enforcing serial execution of consecutive stages through synchronization barriers (Figure 4).

As in Quake, synchronization is not necessary during the read-only reply stage, since its execution does not overlap with any of the other stages. This is also true for the administration phase, since its tasks, e.g., load balancing, are executed by a single thread. Consequently, the stage handling client requests is the only stage requiring protection against concurrent accesses to shared data. We describe the parallelization of this stage in detail, including our application programming environment, the parallelization challenges and our strategy for both a lock based implementation and a transactional memory implementation, in the following sections.

3.1 Programming environment

We use an existing Software Transactional Memory library, *libTM* [Lupei et al. 2009], for STM parallelization of SynQuake. We use a low overhead implementation of test-and-set locks with exponential backoff, instead of the standard pthreads interface, for both our lock-based SynQuake parallelization and within our STM.

The STM allows transactions on different processors to manipulate shared in-memory data structures concurrently in a data-race-free manner.

We chose *libTM* over commercially available STM support, such as Intel's STM compiler, or other research prototypes, for two reasons. First, we needed access to, and understanding of, the STM source code in order to understand the STM game performance better. More importantly, *libTM* offers us high flexibility, and versatility in terms of choices of STM protocol, because it implements all possible combinations of eager, and lazy conflict detection policies; *libTM* also implements hybrids based on partial rollback techniques [Lupei et al. 2009], which make these protocol choices relatively workload independent.

Recent studies on STM performance [Spear et al. 2006] on an implementation of RSTM [Marathe et al. 2006] suggest that none of the eager, lazy or mixed approaches to conflict detection in RSTM work best across all workloads.

Other STM environments also implement a variety of conflict detection strategies: pessimistic, such as in TinySTM [Felber et al. 2008, Riegel et al. 2006, 2007], optimistic as in TL2 [Dice et al. 2006], or hybrid such as in SwissTM [Dragojevic et al. 2009]. However, since none of them implement all conflict detections exhaustively, it was unclear to us which of these existing libraries would perform best for our workload. Finally, some STM's, such as, McrtSTM [Saha et al. 2006] have been shown to perform poorly under high contention scenarios [Dice and Shavit 2007].

While a comprehensive description of *libTM* is beyond the scope of this paper, in the following sections we briefly describe some of its features.

3.1.1 libTM library interface

In an STM program supported by the *libTM* library, transactions need to be delineated with `begin_transaction` and `commit_transaction` statements. Furthermore, shared data needs to be distinguished from private per-thread data accessed inside transactions. For this purpose, transactional shared and private variables should be declared using the meta-types `tm_shared` and `tm_private`, respectively. For example, a shared variable, `int x` in the original program, needs to be declared as `tm_shared<int>x`. The definition of each of these meta-types in our library is a C++ template using the original type of the variable as a parameter, (e.g., `tm_shared<original type>`).

Declarations for `tm_types` are used in *libTM* for run-time access tracking through operator overloading.

3.1.2 Access tracking, conflict detection and resolution in libTM

Any read or write accesses on shared and private transactional variables are tracked inside the implementation of the overloaded conversion or assignment operators. Furthermore, *libTM* maintains recovery data for both `tm_shared` and `tm_private` variables updated in transactions, while performing conflict detection and resolution only for `tm_shared` variables.

Access tracking: Meta-data information encapsulated in each `tm_shared` variable allows for access tracking to take place at word-level granularity. *libTM* also provides the capability of varying the granularity of access tracking dynamically, at runtime. This is achieved through a mechanism of redirection that can remap a `tm_shared` variable to any meta-data object indicated by the programmer. By remapping several semantically-linked `tm_shared` variables to the same meta-data item, we can effectively increase the granularity of the access tracking on the fly. Meta-data remapping can factor in semantic aspects that may vary in time. This can handle dynamic data structures, as well as variables that

are not adjacent in memory, whereas application-level objects are statically declared at compile time.

Conflict Detection: *libTM* implements a variety of protocols in terms of the timing of conflict detection from eager to lazy, and hybrids. All our STM versions provide reductions in number of conflicts and conflict duration compared to lock-based parallelization for our game application. In this paper, we use a fully optimistic, but blocking approach to conflict detection, which optimizes conflict duration, as follows. Multiple readers and multiple writers can access a location concurrently. At commit time, each writer obtains exclusive locks for all locations in its write set and resolves any existing conflicts with other transactions.

Conflict Resolution: libTM solves write-write conflicts by maintaining multiple private copies of a shared object, and applying the concurrent writes to shared memory in the order of committing transactions. Any read-write conflicts detected at commit time are resolved by aborting the conflicting reader transactions. For this purpose, our *libTM* library uses an invalidation strategy that relies on `visible-readers`. Specifically, every reader records its access of a memory location in the `visible-readers` set associated with that memory location. Consequently, performing a read also involves a write with this policy. An updating transaction sets the abort status of all reading transactions for the updated locations before committing. To avoid any inconsistent executions, a transaction checks its abort status at every operation on shared state.

In the following, we describe the parallelization challenges for the game when using these programming environments.

3.2 Synchronization issues for player actions: false sharing

For processing a player action, the server needs to perform collision detection against all game objects intersecting the player's trajectory. Since the avatar's direction can be altered by collision with entities situated in its path, the avatar's trajectory and its final position are impossible to predict from the beginning of the action. However, the whole action and its effects on the game world need to appear as a consistent, atomic unit to the players.

Therefore, processing a client request in Quake consists of: i) computing the potential area of the game map impacted by the action, which we will call *area of interest of the action* and then ii) performing the game action, by determining its effects upon game entities. This request processing scheme leads to a potentially significant reduction in the degree of false sharing for a TM-based versus a lock-based game parallelization scheme in terms of both i) the number of objects involved in collision detection (false sharing in space) and ii) the duration of the potential conflicts for these objects (false sharing in time) as we explain in the following.

In a lock-based implementation, the entire *area of interest of the action* needs to be conservatively locked for the duration of all processing related to the action. For example, let's consider the scenario in Figure 5, where the player executes a shoot-after-move compound action. Since the player may stop short of its intended destination, or change direction during the move e.g., due to encountering an obstacle, the long-range (shooting) interaction may occur at any point in time during the move itself. Hence, the lock-based implementation needs to compute an area of interest, and conservatively lock all objects corresponding to the long range interaction at all possible points of the player's trajectory. This could incur substantial false sharing between threads in both space and time.

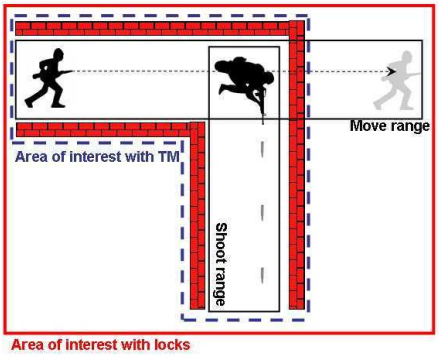


Figure 5. Areas of interest for a move followed by an attack

In contrast, an STM-based implementation implicitly acquires access to objects gradually, as the server progresses through the execution of the action. The move can thus be decomposed into sub-actions, and collision detection can be performed for each sub-action as it dynamically happens, resulting in a substantially smaller bounding box for the overall action, as shown in Figure 5, as well as shorter total time of protected access to the objects involved.

3.3 Synchronization algorithms for request processing

In the following, we describe the synchronization protocol used in the processing stage of the server. We present our two designs for maintaining consistency of the game map: lock-based and STM-based. The algorithms in pseudocode for both versions are included in Figure 6.

In the lock-based version, Figure 6a, we first compute the area of interest corresponding to the currently executing action. Then, ownership of this entire area of interest is acquired by locking all the areanode tree leaves overlapping it, in a predefined order, such that deadlocks are avoided. For simplicity, this ordering is provided by the depth-first traversal of the tree.

Next, we need to apply the effects of the action on all entities overlapping the area of interest. We first process all previously locked leaves, and their associated entities. Then, we process the common ancestor nodes of the locked leaves searching for entities that might intersect multiple grid units. Note that since the tree leaves covering our whole

```
/*Algorithm: Action processing in SynQuake using locks*/
function processAction( actionPlan, player )
{
    //Compute expanded area of interest for the entire action plan
    expanded_range = computeExpandedAreaOfInterest( actionPlan, player );

    /*Lock expanded area of interest*/
    //Get areanode leaves overlapping the expanded area of interest
    expandedLeavesSet = getOverlappingLeaves( area_tree, expanded_range );
    foreach leaf in expandedLeavesSet        Lock( leaf );

    foreach sub-action in actionPlan
    {
        //Get areanode leaves overlapping sub-action's area of interest
        range = computeAreaOfInterest( sub-action, player );
        leavesSet = getOverlappingLeaves( area_tree, range );

        /*Processing sub-action in leaf nodes*/
        foreach leaf in leavesSet
            foreach entity in leaf.entitySet
                perform( sub-action, entity );

        /*Processing sub-action in parent nodes*/
        //Get areanode parents overlapping the area of interest
        parentsSet = getOverlappingParents( area_tree, range );
        foreach parent in parentsSet
        {
            //Temporarily lock parent
            Lock( parent );
            foreach entity in parent.entitySet
                perform( sub-action, entity );
            Unlock( parent );
        }
    }
    /*Unlock expanded area of interest*/
    foreach leaf in expandedLeavesSet        Unlock( leaf );
}
```

(a) Lock-based version

```
/*Algorithm: Action processing in SynQuake using TM*/
function processAction( actionPlan, player )
{
    BEGIN_TRANSACTION();

    foreach sub-action in actionPlan
    {
        //Compute area of interest for the current sub-action
        range = computeAreaOfInterest( sub-action, player );

        /*Processing sub-action in leaf nodes*/

        //Get areanode leaves overlapping the area of interest
        leavesSet = getOverlappingLeaves( area_tree, range );

        foreach leaf in leavesSet
            foreach entity in leaf.entitySet
                perform( sub-action, entity );

        /*Processing sub-action in parent nodes*/

        //Get areanode parents overlapping the area of interest
        parentsSet = getOverlappingParents( area_tree, range );

        foreach parent in parentsSet
            foreach entity in parent.entitySet
                perform( sub-action, entity );

    }

    END_TRANSACTION();
}
```

(b) STM-based version

Figure 6. Pseudo-code for processing actions in SynQuake

area of interest have already been locked, the only purpose of locking these additional parent nodes is to protect the integrity of the entity lists they manage. Indeed, these entity lists may be modified concurrently by other player actions accessing objects co-located in that same parent node, but outside our area of interest. Since parent nodes are sensitive to contention, the locks on parent nodes are released as soon as the processing of their entities is complete. Finally, we relinquish ownership of the area of interest by releasing the locked leaves.

In the transactional version of SynQuake, we first tag all mutable data structures, or parts of them, with `tm_shared` annotations. Players are mutable game entities that can have both their position and attributes modified as a result of game interactions. Resources are partly mutable, e.g., apples can have their attributes affected by game play, but not their position, while walls and the area node tree structure are immutable. As a result, player entities as a whole, the attribute fields of resources, and the entity lists maintained within area nodes are declared as `tm_shared`, with everything else left as private.

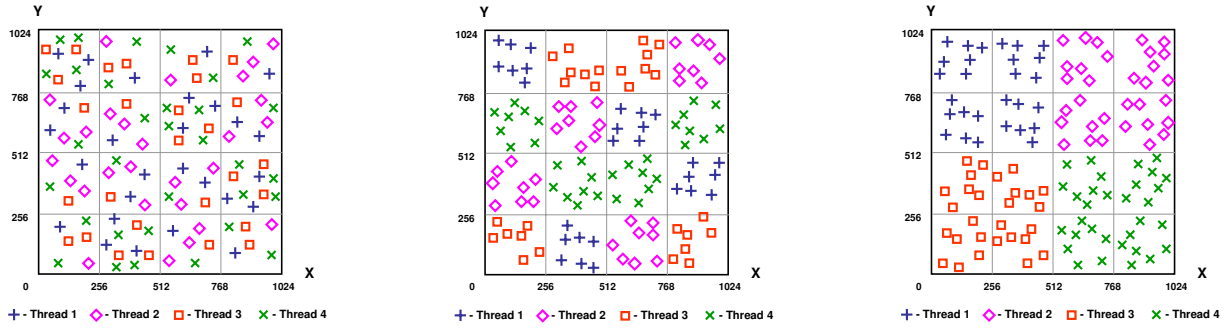
Figure 6b presents our pseudocode. We mark the action transaction with a `begin` and `commit` construct. For each sub-action in the action plan, we compute the area of interest associated with the sub-action. We then process all entities located within both leaves and parent nodes overlapping the area of interest, with consistency for the whole action being seamlessly ensured by the underlying STM library. In contrast to the lock-based version, with STM, we thus did not need to worry about: i) the order of accesses for deadlock prevention ii) locking granularity and interplay of different granularity locks, and iii) optimal placement of lock and unlock operations. The lock-based version of SynQuake took us many months of analytical understanding, code restructuring, and fine-tuning for performance, the availability of the previously parallelized Quake code [Abdelkhalek and Bilas 2004] notwithstanding. In contrast, the STM SynQuake version was completed in less than one month.

3.4 Load balancing issues: true sharing

While concurrency in multiplayer game servers can be severely limited as a result of false sharing, true sharing patterns can also degrade application performance.

Since two different threads may handle players performing actions affecting the same game entities, or interacting directly with one another, true sharing may occur on entities located at the boundary of thread assignments. Therefore, to reduce synchronization costs resulting from true sharing, the load balancing policy should take into consideration the spatial locality of players in the game.

True sharing can thus be mitigated by locality-aware task-assignment policies that allocate the processing of nearby entities in the game to the same thread. However, using locality as a sole criteria in assigning tasks can cause overloading in threads processing highly populated areas. Thread overload results in increased response times and degraded user experience. Moreover, load imbalance results in idle time at barriers for underloaded threads. Consequently, the



(a) Round-robin (b) Spread (c) Locality-aware

Figure 7. Load Balancing Policies

load balancing policy should achieve a good compromise between balanced load, and reduced synchronization.

3.5 Load balancing in SynQuake

We analyzed the trade-off between load balancing, and minimizing synchronization among server threads across three policies. A *round-robin* policy optimizes for an equal distribution of the workload to processing threads. However, since it offers very poor spatial locality for task assignment, players situated next to one another could be handled by different threads (see Figure 7a); this potentially causes high levels of true sharing.

Our *spread* policy is a simple policy, which leverages the spatial locality of players, to perform load balancing at the coarser granularity of grid units. This type of assignment ensures that players located in a given grid unit are handled by the same thread (see Figure 7b), thereby reducing synchronization overheads.

Nevertheless, the grid unit size is static and players may engage in dynamic actions causing conflicts across grid units. To reduce synchronization costs, we developed a dynamic *locality-aware* load balancing policy. Our *locality-aware* algorithm detects player agglomerations on the fly, and assigns each heavily contended area to a separate thread. The algorithm involves a graph representation of the game map, $G = (V, E)$, where V is the set of nodes representing a subset of grid units, which have at least one incident edge, and E is the set of edges. Node u has a connecting edge to node v if: a) u and v are neighbour grid units and b) the number of possible conflicts on the common border exceeds a given threshold.

After constructing the graph, we calculate the *connected components*, using a *union-find with path compression* algorithm. Finally, we distribute the connected components uniformly to threads, possibly splitting some connected components across threads, if they are too few, or too large, preserving locality as much as possible.

This ensures that a highly contended area would be processed by a single thread, hence optimizing for the best spatial locality. In Figure 7c, we show a scenario with four quests located in the middle of the four quadrants of the game map, respectively. As we can see, our dynamic load balancing algorithm assigns players to threads roughly by quadrant in this case.

4. Experimental Results

In our experimental evaluation, we first explore the scalability and performance of STM versus lock-based SynQuake in a default game scenario, and with locality-aware load balancing. We show that STM-Quake outperforms lock-based SynQuake at 2, 4, and 8 threads. We then extend the comparison to a wide range of scenarios with different parameter settings for the game, and the STM. Specifically, we vary the amount of physics computation, load balancing algorithm, quest location, and range of in-game actions in SynQuake. We also vary the consistency granularity of the STM.

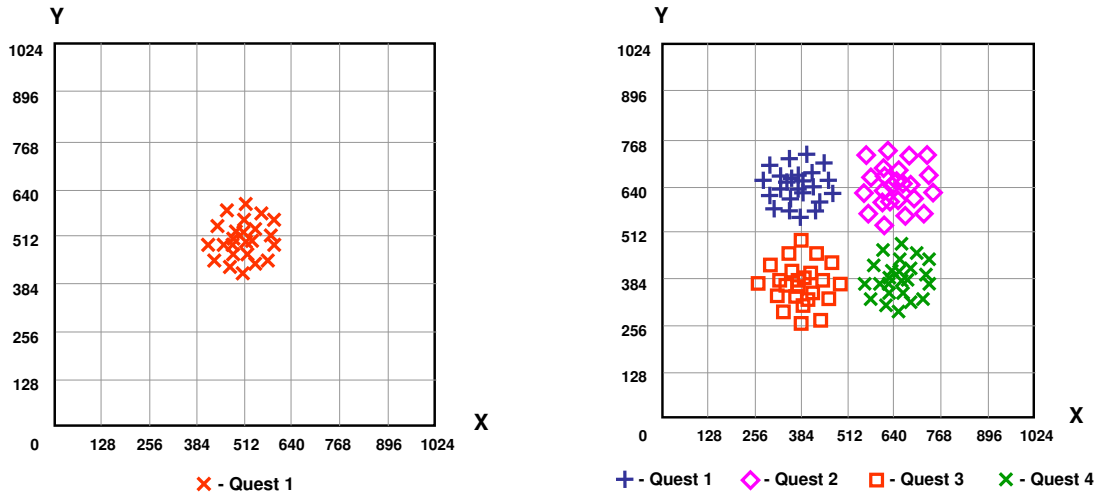
4.1 Experimental setup

Our experimental testbed is an 8-core machine, consisting of 2 Intel(R) Xeon(R) Quad-Core E5472 @ 3.00GHz CPUs, 6MB cache, 3GB RAM, running a Debian Linux kernel 2.6.24 and gcc version 4.2.4.

The experiments on SynQuake have been conducted with a configuration of 600 to 2000 players, running for 1000 server frames on a 1024 by 1024 map, with an areanode tree depth of 8 (resulting in 256 tree leaves).

4.2 Performance Comparison: STM versus Lock-based SynQuake

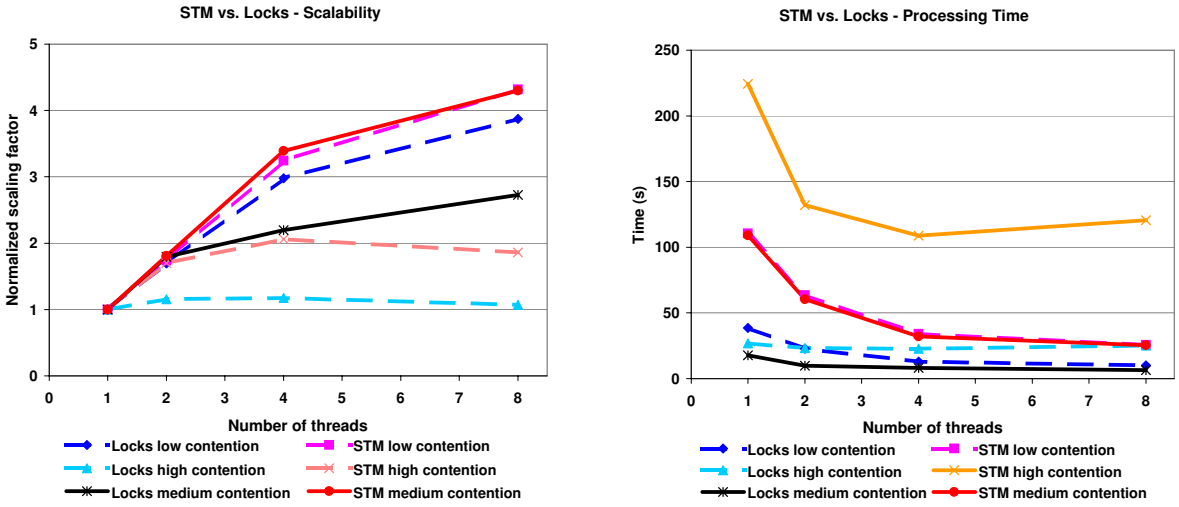
In this section, we compare the performance of the two versions of SynQuake, using the default SynQuake settings. Specifically, as previously described, game physics computation consists of collisions with immutable obstacles/walls on the 2D game map. We use a locality-aware load balancing algorithm, and an entity-level consistency granularity in the STM. We run the game with a medium contention workload scenario with four simultaneous quests positioned around the center of the map (Figure 8b).



(a) Single centered quest generating a high level of contention

(b) Four quests placed around the center of the map generating intermediate levels of contention

Figure 8. Quest scenarios offering different levels of contention



(a) Scalability

(b) Processing Time

Figure 9. Comparison between STM-based vs. Lock-based versions of SynQuake in different quest scenarios.

Figure 10 shows that the performance of STM-SynQuake is worse than that of lock-based SynQuake when running with a single thread, but STM-based SynQuake scales better. Thus, the STM version outperforms the lock-based version in terms of absolute processing time when running with 2, 4 and 8 threads.

4.3 STM versus Lock-based SynQuake: varying in-game contention

We compare the scalability of STM-SynQuake versus lock-based SynQuake in a variety of workload scenarios. We also show the associated performance comparison. In order to stress the STM to the maximum, we run with a game map where all immutable obstacles/walls have been removed. Since walls are *non-TM* data structures, the associated physics computation for collison detection with walls involves no STM overheads.

We use a no quest, low contention scenario and two other quest scenarios depicted in Figure 8, i.e., a high contention single quest scenario, and a medium contention four quest scenario. We can see that, in all scenarios, STM-SynQuake scales significantly better than its lock-based counterpart.

First, in the low contention scenario, both STM and lock-based synchronization achieve high scaling factors. Next, the single quest scenario, shown in Figure 8a, allows us to ob-

serve the behavior of each synchronization scheme under the highest level of contention possible. As shown in Figure 9a, lock-based synchronization fails to achieve any scaling in this case, while the STM version delivers a scaling factor of 2.05x scaling at 4 threads. By acquiring ownership of entities gradually, as opposed to all at once, the STM allows for more parallelism between threads processing nearby players with overlapping areas of interest. Finally, in the third, medium contention scenario, the false sharing induced by conservative locking significantly affects performance for lock-based SynQuake, whereas the STM achieves almost identical scaling to the one obtained in the first scenario, where contention was at its lowest level.

While the STM achieves superior scaling in all scenarios considered, it suffers from high overheads, hence higher processing times, as seen in Figure 9b. To summarize the trend presented in the last two sections, the STM is expected to scale regardless of the complexity of the *non-TM* physics computation. A case with *no physics computation* in the game whatsoever, as presented in this section, would be unrealistic for *any* game. The more complex the game physics computation, e.g., due to wind, gravitation, 3D explosions, the more such computation is expected to hide the STM overheads, the higher the performance advantage of the STM over Locks.

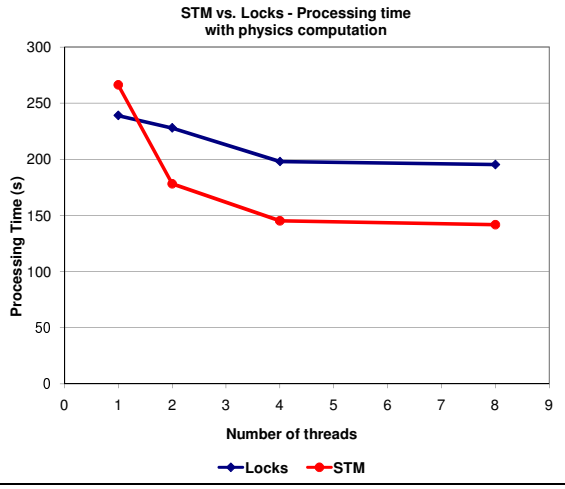


Figure 10. Performance comparison between STM and Locks in default SynQuake setting

4.4 The effect of load balancing on scaling

In the next series of experiments, we study the impact of load balancing for short range, as well as long range actions, under medium contention, when players are guided towards 4 quests, located around the center of the map (Figure 8b). In Figures 11a and 11b, we show the scaling factors obtained by Locks and STM under three load balancing schemes: *round-robin*, *spread* and *locality-aware*.

Figure 11a presents scaling results for all the load balancing policies, under both STM and Locks, when players are performing short range actions. The benefits of maximizing locality with the *locality-aware* policy are illustrated by its significantly higher scaling factors under either Locks or STM. The *spread* policy outperforms *round-robin* also due to its better locality, hence reduced true sharing.

Figure 11b presents the scaling of the different load balancing policies when players perform long range actions. We see a dramatic drop in the performance of Locks compared to the short range action scenario. This is due to the substantial increase in false sharing within conservatively locked, larger areas of interest for long range actions. In the presence of high degrees of *false sharing*, we can see that the different load balancing policies have no significant impact on the overall scaling performance of Locks.

Conversely, in the case of the STM, where the false sharing degree is substantially lower, the load balancing policy has an observable effect by reducing true sharing. Overall, we notice that the STM benefits from *locality-awareness* to the same degree as in the case of short range actions.

4.5 STM detailed statistics

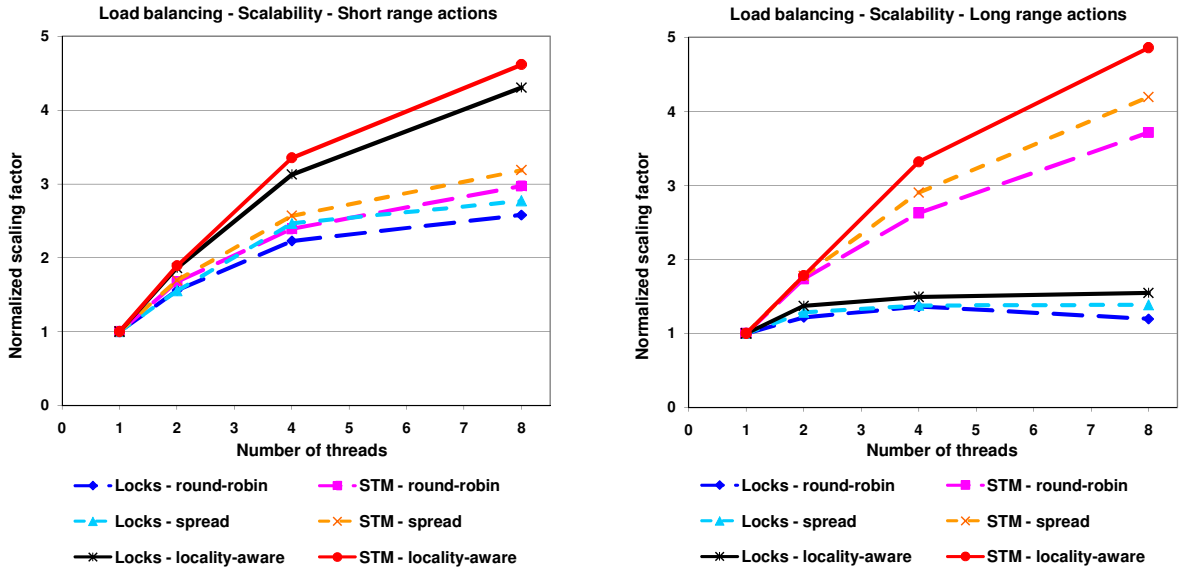
In this section, we present statistics, in terms of abort rates, and the write ratio in the game, when varying the level of contention, and the load balancing algorithm for the STM. We also study how the STM overhead varies with the consistency granularity.

4.5.1 Abort rates

Table 1 shows the abort rates and write ratios in the STM, under several levels of contention: low, medium and high. We use the default game and STM settings from section 4.2.

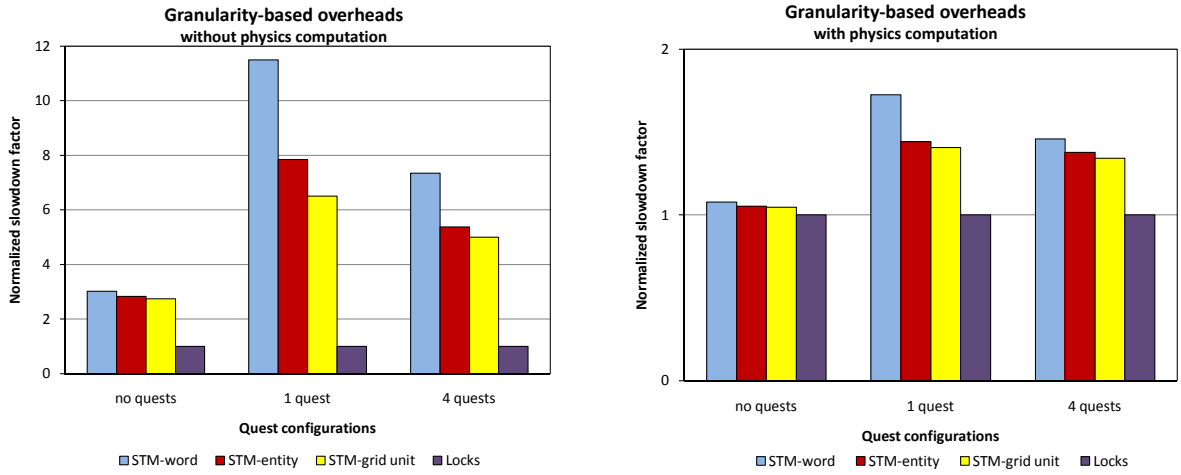
Collision detection involves reads, while updating the player's position involves a write. The number of reads in each transaction grows with a higher number of players situated in each player's area of interest. If the player's movement is blocked by an obstacle, or another player, no update of the player position i.e., no write, takes place. Therefore, a low write ratio is more likely in high contention scenarios, where players are very crowded. For lock-based synchronization, player overcrowding scenarios result in much longer critical sections, hence longer conflict durations, and waiting times, even with a low write ratio. For the STM, the main problem with high contention scenarios is repeated reads of the same data by different processors. This causes high cache-coherence invalidation traffic in the underlying hardware due to the *libTM* update to the `visible-readers` set upon each read.

Experiments with other game settings show higher abort rates. For example, a round robin load balancing policy produces 14% and 6% abort rates at low contention, and high contention, respectively, for the STM, at 8 threads. These results correspond to the same in-game scenarios as above, for the same write ratios as in Table 1. Moreover, if we execute a write in the game even when the move does not change the player position, the maximum abort rates in



(a) Short range actions (b) Long range actions

Figure 11. The effect of load balancing on scaling in STM vs. Locks, for short and long range actions.



(a) Without physics computation (b) With physics computation

Figure 12. STM vs. Locks - overhead comparison for different consistency granularities

the STM reach 59% for the high contention scenario, at 8 threads, when using a round robin policy. For all of these game settings and scenarios, the STM scales the same, or better than the lock-based version of the game.

4.5.2 STM overheads when varying the access tracking granularity

We compare the performance of STM SynQuake under different consistency granularities with that of the Lock-based version. The evaluation was performed by running SynQuake in single threaded mode, under different quest scenarios. The normalized slowdown factors of different STM versions relative to Locks are plotted in Figure 12a.

In the first, no quest scenario, players tend to spread out evenly throughout the map, leading to a low player density. We thus spend much more time traversing the area node tree to search for leaves within the area of interest (non-TM computation) versus accessing the mutable entities themselves (which incurs STM tracking overheads).

Consequently, even without physics computation, in this scenario, the STM experiences a relative slowdown of only 3x. We also notice that, in this case, varying the granularity

Contention Level	No. threads	Abort rate	Write ratio
Low	1	0%	22.74%
	2	1%	22.70%
	4	2%	22.65%
	8	6%	22.56%
Medium	1	0%	4.65%
	2	1%	4.80%
	4	2%	4.76%
	8	3%	4.77%
High	1	0%	0.62%
	2	1%	0.62%
	4	2%	0.63%
	8	2%	0.63%

Table 1. STM statistics for locality-aware load balancing, over 2 million transactions

of access tracking in the STM does not provide significant benefits.

In the second scenario, where we have a single quest in the center of the map (Figure 8a), players tend to cluster in the region of the quest. Player crowding thus results in a low ratio of non-TM computation to STM tracking overheads. Consequently, the STM performance shows a significant slowdown. However, when increasing the granularity of access tracking in the STM, from word-level to entity-level or grid-unit level, the overheads associated with bookkeeping inside the STM library are substantially reduced, resulting in better overall performance.

Finally, the third, medium contention quest scenario results in an intermediate ratio of non-TM computation to STM tracking overhead during the processing of player actions. As a result, the STM experiences intermediate levels of slowdown, while also benefiting from coarser access tracking granularity.

When, in addition, physics computation is part of the processing phase of each action, thus increasing the weight of non-TM computation, we notice that the relative slowdowns between STM and Locks range from 1.07x to 1.72x, as illustrated by Figure 12b. These slowdowns are significantly smaller, compared to the corresponding cases without physics computation.

4.5.3 STM access tracking granularity trade-offs

We evaluate the trade-offs between false sharing reduction and overhead reduction in the STM, by varying its access tracking granularity, in Figure 13. Specifically, we examine the trade-off between entity and grid-unit access tracking granularity in a scenario with one quest in the center of the map (Figure 8a). We can see that the version of STM with grid-unit level granularity incurs lower overheads, corresponding to the results obtained previously when running with a single thread. However, as we increase the number of threads, we notice that the finer granularity of entity-level access tracking, which has the advantage of incurring no false sharing, allows for better scalability, and eventually for better overall performance in spite of its higher initial overheads.

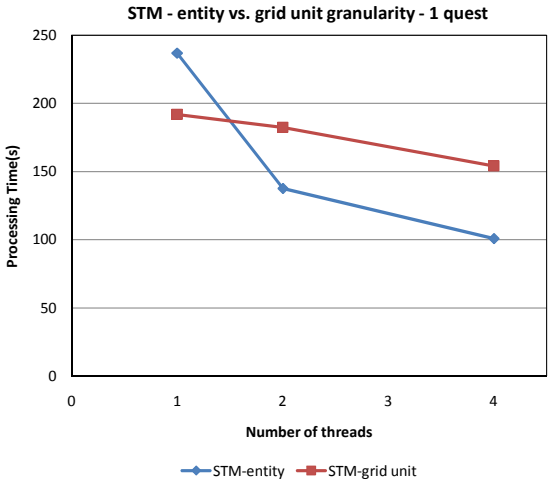


Figure 13. STM - entity vs. grid-unit granularity

5. Related Work

Many recent research efforts focus on parallelizing realistic applications with STM. However, all previous results are either: i) negative in terms of performance, instability, poor programmability, crashes or ii) do not use a real STM, simulating abstract STM primitives instead, as follows.

Kulkarni et al. [Kulkarni et al. 2006] discuss using transactional memory for parallelizing sequential applications, where compiler analysis is unable to detect opportunities for concurrency, due to input-driven dependencies. They argue that, when using STM, conflicts can be discovered dynamically at run-time and enforced only when necessary. Their discussion is centered around an algorithm for Delaunay mesh generation. However they do not provide an implementation, experimental evaluation or comparison between lock-based and STM-based versions.

Scott et al. [Scott et al. 2007] experiment with an implementation of Delaunay triangulation, in which most of the work is privatized. Since this application exhibits very little contention, the results show that the fine-grain and the coarse-grain locking-versions of the application achieve similar results, with the STM version performing 2x worse because of indirection overheads (specific to the RSTM library used). Consequently, this study does not provide a good example of an application that might benefit from using STM.

Kang and Bader [Kang and Bader 2009] analyze the benefits that STM might bring when designing algorithms with irregular access patterns, such as graph algorithms. More specifically, they suggest that STM facilitates scalable and easy to develop versions of such algorithms, as long

as the probability of conflict between transactions remains low. They exemplify their approach with an algorithm for computing a minimum spanning forest of sparse graphs. Even though their STM-based implementation demonstrates remarkable scalability, the high overhead of the STM system completely offsets the speedup due to scalability.

Dragojevic et al. [Dragojevic et al. 2008] use a large scale benchmark, called STMBench7, to expose some of the weaknesses of the current STM implementations: crashes caused by memory management limitations, lack of support for external libraries, and only partial support for object oriented features. They conclude that these issues prove to be a major limitation when adapting STMs for production use.

Multiplayer games have emerged as an important application domain area, as well as an important driver of the market for multi-core processors. Previous studies of game server behavior and performance [Abdelkhalek et al. 2001] have also concluded that game servers have very different access patterns compared to scientific workloads, and point to game parallelization as a largely unexplored research area.

Previous research work in the area of parallelizing game server code [Abdelkhalek and Bilas 2004, Zyulkyarov et al. 2009] has focused on the publicly available game code for Quake, and either lock-based [Abdelkhalek and Bilas 2004], or transactional memory programming paradigms [Zyulkyarov et al. 2009, Gajinov et al. 2009].

Abdelkhalek et al. [Abdelkhalek and Bilas 2004] investigate the parallelization and performance of a multithreaded version of the Quake server with lock based synchronization. This study concludes that games are highly dynamic applications that exhibit fine-grain interactions and achieving good scaling for such applications is a challenging task. They also provide an in-depth analysis of the performance bottlenecks, which are primarily due to lock contention during the processing stages, and load imbalance at global synchronization points.

Zyulkyarov et al. [Zyulkyarov et al. 2009] study an STM-based parallelization of the Quake server using the prototype edition of the Intel C++ STM Compiler [Ni et al. 2008]. The paper is an experience paper, where the authors reveal the difficulties they had and the manual modifications needed to get the code to compile and run, such as getting printf's manually out of transactions. The authors also reveal the high number of aborts they obtain when they run the parallel version of Quake, and the poor scalability, likely due to the inadequate test cases that come with Quake itself i.e., the small map which may have forced the authors to run with a maximum of 8 players. Finally, in contrast to our work, the authors use a simple static load balancing technique, and do not discuss the impact of load balancing on true sharing.

A follow-up paper by the same authors [Gajinov et al. 2009] presenting a different version of parallel STM Quake, called QuakeTM, is also a negative result in terms of performance. The authors use coarse-grained transactions and rely on the Intel C++ STM Compiler. The focus is on simplifying programming rather than performance. Their results show reasonable scaling, but very high STM overheads and abort rates, caused by the coarse-grained transactions.

In contrast to the above papers, our work presents a systematic comparison of the design and implementation issues of each of the two approaches to parallelizing multiplayer games, as well as a comprehensive experimental evaluation. We furthermore show that an STM implementation can provide better performance than a lock-based implementation, due to inherent game code artifacts. This, to our knowledge, is the first application parallelization effort providing a positive outlook for the use of STM in realistic applications with highly dynamic access patterns, such as multiplayer games.

Our load balancing algorithms build on ideas from earlier work in the area of scaling *distributed* game servers. Specifically, Ng et al. [Ng et al. 2002], Cronin et al. [Cronin et al. 2004], and Chen et al. [Chen et al. 2005] study the effects of load balancing in distributed multi-server environments. Their work focuses on network bottleneck aspects, as the primary source of contention, as opposed to synchronization artifacts. Chen et al. [Chen et al. 2005] find that dynamic game world partitioning based on locality awareness improves average response time by effectively aggregating game regions and thus minimizing inter-server communication.

6. Conclusions

In this paper, we show the first case study of a high impact application where leveraging *software* Transactional Memory (STM) support for parallelization provides *better performance* than state-of-the-art lock-based parallelization.

Specifically, we study parallelization of multiplayer game server code, through developing a game benchmark, SynQuake, that extracts the main data structures and the essential features of the popular game Quake. Our results show higher scalability for STM-SynQuake versus lock-based SynQuake, due to a higher degree of false sharing in the latter. Overall performance is better at 4 and 8 threads for STM-SynQuake versus lock-based SynQuake for all realistic game scenarios we studied. The superior performance comes from a reduction of false sharing in the game application. The STM supports decomposing a player action into sub-actions, and performing collision detection on the fly. Thus, STM support substantially reduces the number of conflicts, and the duration of conflicts in the game application. The consistency of the action as a whole is automatically provided by the STM. In contrast, in the lock-based implementation, conservative locking for the entire player action becomes unavoidable.

Finally, in the context of the STM, we explore the effect of the consistency unit granularity on the STM overhead. We also explore, as an orthogonal but important factor, the effect of task assignment on the true sharing patterns between

threads. These design choices have a second-order impact on performance, once false sharing has been reduced; an object-level consistency granularity and a dynamic locality-aware task assignment provide the best performance.

7. Acknowledgements

We thank the anonymous reviewers and our shepherd, Pascal Felber, for their guidance. We would also like to thank Angela Demke Brown for her careful reading and detailed comments on an earlier version of our paper. We further acknowledge the generous support of Intel, IBM Centre for Advanced Studies (CAS), IBM Research, the Natural Sciences and Engineering Research Council of Canada (NSERC), and Ontario Centers for Excellence (OCE).

References

A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. *Parallel and Distributed Processing Symposium, International*, 1:72a, 2004.

A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. In *Cluster Computing*, 2001.

J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *PPoPP*, pages 289–300, June 2005.

E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications*, 23(1):7–30, 2004.

D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, September 2006.

D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization (CGO)*, pages 21–33, March 2007.

A. Dragojevic, R. Guerraoui, and M. Kapalka. Dividing transactional memories by zero. *TRANSACT*, 2008.

A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 155–165, 2009.

P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 237–246, 2008.

V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Quaketm: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 126–135, New York, NY, USA, 2009. ACM.

Intel Corporation. Intel C++ STM Compiler Prototype Edition 2.0 Language Extenstions and User Guide. 2008.

S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 44(4):15–24, 2009.

M. Kulkarni, K. Pingali, and L. P. Chew. Using transactions in delaunay mesh generation. 2006.

D. Lupei, A. Czajkowski, C. Segulja, M. Stumm, and C. Amza. Automatic adaptation of transactional memory state management to application conflict patterns. In *The 13th Workshop on the Interaction between Compilers and Computer Architectures 2009 (Interact 2009)*, pages 47–56, February 2009.

V.J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

B. Ng, A. Si, R. W.H. Lau, and F. W.B. Li. A multi-server architecture for distributed virtual walkthrough. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 163–170, 2002.

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: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 195–212, New York, NY, USA, 2008. ACM.

T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 284–298, September 2006.

T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 221–228, 2007.

B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrtstm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–197, March 2006.

M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC '07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, pages 107–113, Washington, DC, USA, 2007. IEEE Computer Society.

M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, September 2006.

F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *PPOPP*, pages 25–34. ACM, 2009.