Towards Scalable and Transparent Parallelization of Multiplayer Games using Transactional Memory Support

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, krickw, amza}@eecg.toronto.edu,
[‡]donpinto@cs.toronto.edu

Abstract

This work addresses the problem of parallelizing multiplayer games using *software* Transactional Memory (STM) support. Using a realistic high impact application, we show that STM provides not only ease of programming, but also *better* performance than that achievable with state-of-the-art lock-based programming.

Towards this goal, we use SynQuake, a game benchmark which extracts the main data structures and the essential features of the popular multiplayer game Quake, but 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 potentially 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 a higher scalability factor of STM-SynQuake versus lock-based SynQuake, due to a higher degree of false sharing in the latter.

Categories and Subject Descriptors D.0 [*Software*]: GENERAL

General Terms Performance

1. Introduction

Transactional Memory (TM) is an emerging paradigm for parallel programming of generic applications, with a goal to facilitate more efficient, programmer-friendly use of the plentiful parallelism available in chip multiprocessors, and on cluster farms. The main idea is to simplify application programming in parallel and distributed environments through the use of transactions. TM allows transactions on different processors to manipulate shared in-memory data structures concurrently in an atomic and serializable i.e., correct manner.

Many commercial and research prototypes for supporting TM in software have been introduced recently. However, efforts from the research or commercial communities towards parallelizing realistic applications using STM have been scarce, mainly because existing STM-based parallelizations typically perform substantially worse than simple single-mutex based implementations.

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.], 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.

In Quake, each player action is 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], this translates into eagerly acquiring ownership of all potentially affected objects of the game map within this bounding box, before processing the action. This 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, collision detection is performed dynamically, hence more accurately, as the avatar encounters various obstacles, which potentially change its direction of movement, as shown by the intermediate steps in Figure 1. The atomicity and consistency of the player action is automatically provided by the underlying STM library. STM support thus results in reduced false sharing overall in both space and time.

Our comparative performance evaluation shows that the main factor affecting overall scaling is the effect of the false sharing inherent in the parallelization scheme. As a result, we experimentally show that the scaling of STM-based SynQuake from one to four server threads is better than the scaling of lock-based SynQuake in all game scenarios we study.

Finally, parallelization with STM automatically hides the details of game consistency maintenance across the boundaries of an underlying partitioned world, thus offering the players the view of a large seamless world which could be hosted on parallel, distributed, or hybrid platforms, completely transparently.

PPoPP'10, January 9–14, 2010, Bangalore, India.
ACM 978-1-60558-708-0/10/01.

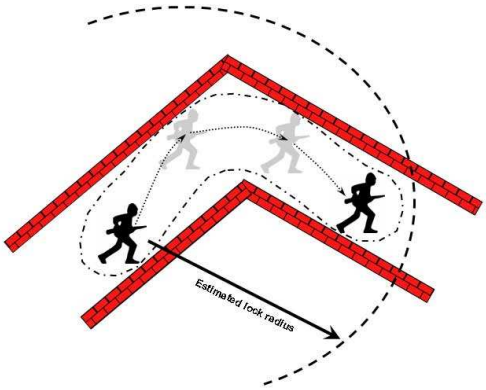

Figure 1. Processing player move: Radius-based locking around player position

2. Environment: SynQuake game

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, based on the popular 3D shooter Quake. The SynQuake game server captures all the basic interactions in a typical multi-player game, and accurately represents network and system features of a game server. In order to replicate massive multiplayer 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 needing to regenerate life, fighting with other players, or fleeing if being chased by a stronger opponent.

3. Parallelization of SynQuake

We use two orthogonal parallelization techniques for SynQuake: a parallel version using lock-based synchronization and the other leveraging a *transactional memory* runtime system. We use a software transactional memory library called libTM, which enables concurrent manipulation of shared in-memory data structures in a data-race-free manner. The runtime library guarantees correct execution by detecting conflicts on shared memory accesses, then rolling back and restarting the conflicting transactions.

In terms of parallelization challenges, we analyzed the false sharing effects, which are significant in the lock-based version because of the conservative locking needed when performing collision detection, and reduced in the case of the STM approach.

4. Experimental Results

In our experimental evaluation, we explore the scalability of STM versus lock-based SynQuake in a variety of in-game scenarios, using quest configurations to simulate high, medium and low contention scenarios.

First, in a low contention scenario with no active quests, we noticed that both STM and lock-based synchronization achieve high scaling factors. This is the result of the low level of contention, since players processed by different threads are scattered all over the map, and have a very small probability of impacting each others' areas of interest.

In the medium contention scenario, we guided the players towards 4 quests positioned symmetrically around the center of the map. In this situation, we can see how the false sharing induced by conservative locking dramatically affects performance, whereas the STM achieves a scaling factor similar to the one obtained when contention was at its lowest level.

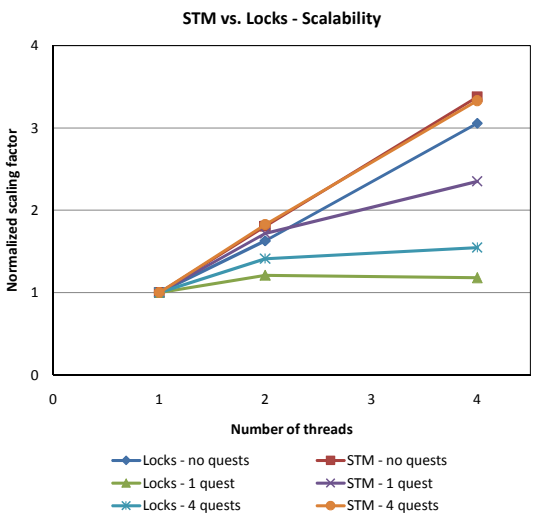

Figure 2. Scalability of STM-based vs Lock-based versions of SynQuake in different quest scenarios.

Finally, in the high contention scenario, we have all players guided towards a single quest positioned in the center of the map. This scenario allows us to observe the behavior of each synchronization scheme under the highest level of contention possible. As shown in Figure 2, lock-based synchronization fails to achieve any scaling while the STM version delivers a scaling factor of 2.35x at 4 threads. The difference in scaling between the STM and lock-based versions arises from the way they acquire ownership of the affected area of interest during an action. While STM acquires ownership of entities gradually as the transaction progresses, lock-based synchronization needs to be more conservative and acquires ownership of the entire area of interest at the beginning of the action.

5. Conclusion

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.

Our results show higher scalability 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 brought about by STM support for decomposing a player action into sub-actions, and performing collision detection on the fly. The consistency of the action as a whole is automatically provided by the STM, as opposed to the lock-based implementation where conservative locking for the entire player action becomes unavoidable.

References

A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. *IPDPS 2004*, page 72a.

D. Lupei, A. Czajkowski, C. Segulja, M. Stumm, and C. Amza. Automatic adaptation of transactional memory state management to application conflict patterns. In *Interact 2009*, pages 47–56.