Lightweight Software Transactions for Games

Alexandro Baldassin
State University of Campinas, Brazil
alebal@ic.unicamp.br

Sebastian Burckhardt
Microsoft Research, Redmond
sburckha@microsoft.com

Abstract

To realize the performance potential of multiple cores, software developers must architect their programs for concurrency. Unfortunately, for many applications, threads and locks are difficult to use efficiently and correctly. Thus, researchers have proposed transactional memory as a simpler alternative.

To investigate if and how software transactional memory (STM) can help a programmer to parallelize applications, we perform a case study on a game application called SpaceWars3D. After experiencing suboptimal performance, we depart from classic STM designs and propose a programming model that uses long-running, *abort-free* transactions that rely on user specifications to avoid or resolve conflicts. With this model we achieve the combined goal of competitive performance and improved programmability.

1. Introduction

With the exponential growth in power dissipation and limitations on the microarchitectural level, the semiconductor industry has shifted its focus towards multicore architectures. Unfortunately, many applications are not designed to exploit true concurrency available on a multicore processor and do thus no longer get the benefit of steady performance improvements with the succession of new chip generations. The task of filling in the gap between hardware and software and bringing concurrent programming to the masses is being regarded as one of the biggest challenges in computer science in the last 50 years [5], with companies such as Microsoft and Intel investing heavily in academic research.

The most common concurrent programming model today is based on a shared memory architecture wherein a *thread* is the unit of concurrency. Threads communicate through reads and writes from/to shared memory and, to avoid conflicts, they synchronize using locks. Often, locks are used to construct so-called *critical sections*, areas of code that can not be entered by more than one thread at a time. An alternative to lock-based synchronization is known as transactional memory [4]. In this model programmers specify transactions, a block of code that appears to execute atomically and in isolation. The underlying runtime system (implemented in hardware, software or a mix of both) is responsible to detect conflicts and provide a consistent (serializable or linearizable) execution of the transactions.

Transactional memory is an active research field and the approach seems promising: benchmarks show that transactional memory can improve the performance of some code, such as scientific algorithms or concurrent data type implementations. Nevertheless, how to successfully integrate transactional memory into more mainstream applications remains an open question [10]. We believe that programmers will adopt TM only if it can deliver *both* a substantially simplified programming experience *and* competitive performance compared to traditional lock-based designs.

To address this general challenge, we conduct a *parallelization case study* on a full-featured game application, SpaceWars3D [3]. We first restructured the sequential code into cleanly separated modules dealing with the different aspects of the game (Fig.1). Then we tried to parallelize the tasks performed in each frame (such as rendering the screen, updating positions of game objects, detecting collisions, and so on). We quickly realized that traditional synchronization (locks and critical sections) is cumbersome to use. For more convenient programming, we tried to execute each task as a transaction, provided by an STM. Unfortunately, performance was poor because of (1) frequent conflicts and rollbacks, and (2) the large overhead of transactional execution.

To solve these problems, we departed from standard STM designs and developed a novel programming model. It replicates the shared data so that each task can access its local replica without synchronization. The replicas are reconciled (by propagating object updates) between frames. The key design decisions are (1) *tasks never abort*, and (2) we guarantee consistency *per object* only.

The user plays an active role in defining synchronization and consistency between tasks. For one, she declares task dependencies in the form of *task barriers*. Secondly, she helps the runtime to reconcile conflicts (that is, two updates to the same object by different tasks) by specifying *merge functions*.

This final strategy performed well. Although it required the user to specify task barriers and/or merge functions, we felt that there was some value to the exercise: most of the task conflicts found and reported during runtime were either

easily eliminated (some even helped us to find bugs in the code), or revealed an important dependency between tasks that we had not realized before.

1.1 Contributions

Our main contributions are (1) that we report on the challenges we encountered when parallelizing a game application, and (2) that we propose a programming model (based on long-running, abort-free transactions with user-specified object-wise consistency) that achieves good programmability and competitive performance.

1.2 Related Work

There are two main works reporting on experiences in using TM to parallelize large applications. Scott et al. [8] employ a mix of barriers and transactions to create a parallel implementation of Delaunay triangulation. Watson et al. [9] investigate how Lee's routing algorithm can be made parallel by using transactions. Our work differs from theirs in the application domain (we have a game application) and in how we solved the performance problem imposed by long-running transactions. Blundell et al. [2] devise unrestricted transactions as a mean of allowing transactions to perform I/O. Aydonat and Abdelrahman [1] propose conflict-serializability in order to reduce the abort rate of long-running transactions. Our solution allows both I/O and long-running transactions by completely avoiding aborts. Rhalibi et al. [6] present a framework for modeling games as cyclic-task-dependency graphs and a runtime scheduler. However, their framework does not use replication, but is based on the classic lock-based synchronization primitives.

Just like us, Rinard and Diniz [7] use object replication for better concurrency. However, they apply it in a different context (a parallelizing compiler).

2. The Game

Our starting point was a 3D Windows version of the classic game Spacewars, as described in a tutorial book [3] and with source code available on the web.[1] In a nutshell, SpaceWars3D consists of two starships shooting one another in space. The game is coded in C# and uses the ManagedDirectX API. Developed to teach game programming, it features many realistic details including 3D rendered graphics, sound, and a network connection.

Because the original game lacked sufficiently heavy computation to challenge our machine, we added moving asteroids to the gameplay. Asteroids may collide with other game objects. By varying the number of asteroids (we used around 1500 in our experiments) we can conveniently adjust the work performed in each frame.

[1] www.apress.com/book/downloadfile/1486

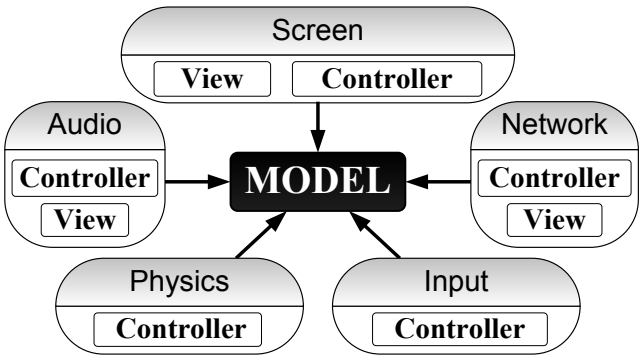


Figure 1. Our Model-View-Controller (MVC) design

2.1 Model-View-Controller Design

As a first step we rearchitected the code following the Model-View-Controller (MVC) design. MVC has been widely used in many different environments. In our case, its role is to express concurrency between the controllers, and to separate shared data (the model) from controller-local data (the views).

Our MVC architecture is shown in Fig. 1. It has a *model* at the center, surrounded by several modules. Each module handles one game aspect (sound, graphics, input, physics, network). The model encapsulates the essential game state. In our case, it consists of hierarchically organized game objects (player ships, asteroids, projectiles). The model is passive — it does not contain any threads, but its objects may be read and updated by the modules.

The *modules* are the gateways between the model and the external world; they respond to external events (incoming network packets, user input) by updating the model, and they in turn send information about the model to external entities (such as the screen, or to remote players). A module contains exactly one controller and may contain one or more views. Controllers are active (they may contain one or more threads). Views encapsulate data relating to a particular external interface. We now describe the modules in turn:

- The screen controller renders the game state once each frame, using the DirectX interface to the graphics card. Parameters and data related to this interface (such as meshes or textures) are encapsulated in the screen view.
- The physics controller performs two tasks each frame. One handles collisions between game objects; the other updates the object positions based on their speed and the elapsed time.
- The input controller processes mouse and keyboard input, firing shots and changing the position and speed of the player ship accordingly.
- The network controller processes incoming packets and periodically sends packets to the remote player.

- The audio controller plays sounds each frame, in reaction to events that happened in this frame.

3. Challenges

We now describe the main challenges we encountered in more detail.

3.1 Finding Concurrency

The original code is almost completely sequential: all tasks (except the receipt of network packets) are performed by the main thread, inside a game loop that repeats every frame. To improve performance on a multicore, we need to find ways to distribute the work for concurrent execution on multiple processors.

In fact, there is plenty of opportunity for concurrency. We distinguish three kinds. For one, as visible in Fig. 1, there is natural concurrency among different controllers. Second, some controllers (such as the physics controller) perform more than one task in each frame, which can be concurrent. Finally, some (but not all) tasks lend themselves to parallelization; for instance, we can split the collision handler into concurrent subtasks.

All three kinds of concurrency are realized in our final solution (see the experiments in Section 5).

3.2 Critical Sections Don't Work

As can be seen from Fig. 1, an easy and safe way to provide synchronization is to use a single lock to protect the entire model, and run the entire task in a critical section. Unfortunately, this scheme would imply that only one task runs at a time, so we would perform no better (and likely worse) than the sequential game.

The standard methodology to address this issue is to use (1) fine-grained locking, and (2) shorter critical sections. We ran into serious problems with this approach, however.

- With finer-grained locks, there is higher overhead. Also, care has to be taken to organize acquires and releases in a fashion that avoids deadlock.
- If a task accesses the same data repeatedly, but the accesses are not all contained in a single critical section, the data may be inconsistent.[2] This can be solved using two-phase locking; but in our game application, this would once more destroy concurrency, because many of our tasks access all game objects in each frame.
- To shorten critical sections, we may need to manually copy shared data to local variables and vice versa. This is cumbersome and hard to maintain.

[2] For example, we ran into the following problem when rendering the screen: the ship position was read in two independent critical sections, once to determine the camera position (the camera is supposedly mounted on the ship) and once to draw the ship hull. Because the ship position was sometimes updated in between the two, the ship hull occasionally appeared in the wrong location on the screen.

```
public class PhysicsController : Controller
{
   public void Start()
   {
      runtime.NewTask("UpdateCollisions",
                      this.UpdateCollisions);
   }
   public void UpdateCollisions(Context context)
   {
      ...
```

Figure 2. Controllers specify periodic tasks, to be called back by the runtime each frame.

- Changing the length or position of critical sections requires nontrivial code changes. It is thus not easy to tune the performance for different machines and for different inputs.

3.3 Optimistic Concurrency Doesn't Work

Next, we turned to software transactional memory (STM) which uses optimistic concurrency control: a runtime system monitors the memory accesses performed by a transaction and rolls them back if there are any conflicts. Unfortunately, we found that in our situation, standard STMs do not perform as envisioned, for the following three reasons.

- STM relies on optimistic concurrency control to amortize the cost of monitoring memory accesses and rollbacks (conflicts are expected to be rare). However, we observed that many of our game tasks were conflicting in every frame, thrashing STM performance.
- Even without conflicts, the overhead of transactional execution is discouragingly large, because we execute large chunks of code within each transaction.
- I/O is not typically supported in a transaction (because it can usually not be rolled back). We found that at least for this game, we can work around this restriction.[3]

Note that all three of these problems could be addressed by reducing the size of transactions; but then we would face similar programmability issues as with critical sections.

4. Solution

Our solution combines the MVC programming model with a classic trick from the concurrency playbook: replication.

First, each controller tells the runtime system the tasks it needs to perform (Fig. 2). The runtime system then calls these tasks concurrently in each frame, giving each task its own replica of the world to work on. At the end of each frame, any updates made to the local replicas are propagated to all replicas.

[3] We can restrict tasks to be either reader tasks (that do not update the model, but may freely do I/O) or updater tasks (that may freely update the model, but not perform I/O). We can use standard STM for updater tasks, and use a special atomic snapshot mechanism for reader tasks.

```
public class Ship
{
   public Versioned<int> score;

   public void ScorePoint(Context c)
    {
        score.set(c, score.get(c) + 1);
        ...
```

Figure 3. The *Versioned* template replicates shared fields under the covers. The context c serves to select the replica.

Rather than replicating all objects in the model using a deep copy, we make shallow copies of the shared fields, by wrapping them with a template that performs the replication under the covers (see Fig. 3). We use three different templates, to wrap value types, collection types, and reference types, respectively.

To deal with task dependencies and conflicting updates, we rely on the user to specify *task barriers* and *merge functions* (Fig. 4). They work as follows:

- If there is a user-specified task barrier ordering task A before task B, then task B does not start until task A is done, and all object updates performed by task A are propagated to task B.
- At the end of each frame, if there are conflicting task updates (more than one task updated the same object), then merge functions specified by the user for this object are consulted to resolve the conflict. If there is no appropriate merge function, the conflict is reported to the user and execution is aborted.

Merge functions are often simple tiebreakers (one update wins, the other one is discarded).

4.1 Implementation and Optimizations

Under the covers, each shared object is instantiated a fixed number of times, in the form of an array (see Fig. 5). Each task (represented by a context object) has an index which gives it access to its own replica. For each task, we record the set of shared objects it has modified.

At the end of each frame, we process all the objects that were modified by some task. If they were modified by just one task, we simply propagate the updated version to all other copies. If they were modified by more than one task, we pairwise reconcile the updates (using the merge functions specified by the user) before we propagate the final result to all replicas.

To reduce the amount of copying, we perform several optimizations. For one, all reader tasks (tasks that do not update any shared objects) share the same "master" replica. Second, we do not actually propagate all updates at the end of each frame. Rather, we perform copy-on-write when a replica is modified for the first time.

```
MakeBarrier("ProcessInput","UpdateWorld");
MakeBarrier("UpdateWorld", "PlaySounds");
MakeBarrier("UpdateCollisions", "PlaySounds");

ship.score.AddMergeFunction(
  (int old, int new1, ref int new2) =>
                      new2 += (new1 - old));
```

Figure 4. The programmer specifies barriers to enforce task dependencies, and merge functions to resolve conflicts.

```
public class Versioned<T>
 {
   private T[] value;
   public T get(TaskContext c)
   {
     return value[c.replica];
   }
   public void set(T newvalue, MovingContext c)
   {
     value[c.replica] = newvalue;
      c.RegisterWriteBack(this);
   }
```

Figure 5. Replicas are implemented as fixed-size arrays.

Experiment	A	B	C
Frames per second	39	59	76
Speedup	n/a	48%	92%
Total time per frame [ms]	25.2	17.0	13.1
Rendering (RefrWnd) [ms]	10.3	10.8	12.9
Collision handling [ms]	14.4	16.0	7.2/6.7/6.4
Position Update [ms]	0.42	0.91	2.0
Number of replicas	1	2	10
Total memory [MB]	85	93	125

Figure 6. Some measurements for the three experiments. All were conducted on a 4-core processor (Intel Q9550) at 2.83 GHz with 2 GB of memory, running Windows Vista, and using a NVidia Geforce 8600 graphics card.

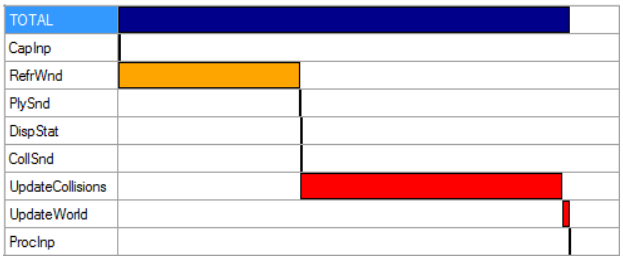
5. Experimental Results

We measured the performance of the game in three experiments on a 4-core machine, using increasing amounts of replication and concurrency, and observing increasing speedups. We show frames per second, speedup, total memory consumption, and the three most time-consuming tasks in Fig. 6. To visualize the results, we instrumented our prototype to capture and display task schedules, and show typical schedules in Fig. 7.

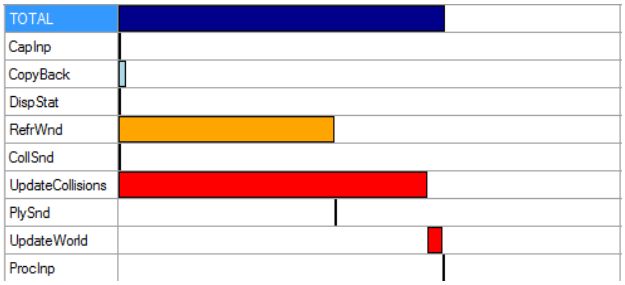
Experiment A (the sequential baseline) performs all tasks sequentially, using no replication and no synchronization. The corresponding schedule in Fig. 7 thus shows no overlap between tasks.

Experiment B (partial concurrency) is similar to traditional double-buffering techniques. It creates two replicas only. One replica is shared among all reader tasks (tasks

Experiment A: Single Replica (no concurrency)



Experiment B: Two Replicas (partial concurrency)



Experiment C: Multiple Replicas (full concurrency)

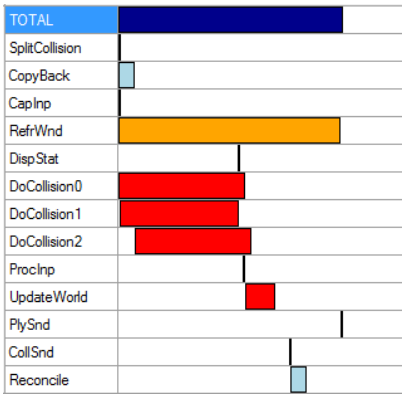


Figure 7. Typical task schedules for the three experiments on our 4-core machine.

that do not mutate the model). The other replica is used by updater tasks (tasks that may mutate the model), which are scheduled sequentially (one updater at a time). The corresponding schedule in Fig. 7 shows significant overlap between the RefrWnd reader task (which renders the screen) and the UpdateCollisions updater task (which handles collision). The total time spent for each frame is thus shorter.

Experiment C (full concurrency) uses one replica per task, as described in the previous section. Furthermore, it breaks the collision detection task into three roughly equal pieces. The corresponding schedule in Fig. 7 shows that on our 4-core machine, we almost reach maximal concurrency for this application and machine (the total frame length can not be smaller than the largest sequential piece, RefrWnd, which is limited by the graphics card).

The (comparatively modest) overhead incurred by our runtime is visible in the form of slightly longer-running tasks, and as two additional tasks (CopyBack and Reconcile).

6. Conclusion and Future Work

We propose a novel programming methodology based on long-running, abort-free transactions with user-specified object-wise consistency. Our experiments on a game application confirm that our programming model can deliver an attractive compromise between programming convenience and multicore performance.

Future work will address how to further simplify the programmer experience (for instance, by supporting default merge functions, events, and observer patterns), and to further engineer the runtime prototype to scale to larger games with many thousands of game objects.

Acknowledgments

We thank the many people that contributed insightful comments and interesting discussions, including (but not limited to) Jim Larus, Tom Ball, Patrice Godefroid, and Trishul Chilimbi.

References

[1] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *Workshop on Transactional Computing*, 2008.

[2] C. Blundell, E. Lewis, and M. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, 2006.

[3] E. Hatton, A. S. Lobao, and D. Weller. *Beginning .NET Game Programming in C#*. Apress, 2004.

[4] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[5] C. O'Hanlon. A conversation with John Hennessy and David Patterson. *Queue*, 4(10):14–22, 2007.

[6] A. El Rhalibi, M. Merabti, and Y Shen. Improving game processing in multithreading and multiprocessor architecture. *Lecture Notes in Computer Science*, 3942:669–679, 2006.

[7] M. Rinard and P. Diniz. Eliminating synchronization bottlenecks in object-based programs using adaptive replication. In *International Conference on Supercomputing*, 1999.

[8] M. Scott, M. Spear, L. Dalessandro, and V. Marathe. Delaunay triangulation with transactions and barriers. In *IEEE International Symposium on Workload Characterization*, pages 107–113, 2007.

[9] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 388–398, 2007.

[10] Workshop on TM Workloads, http://freya.cs.uiuc.edu/WTW/, 2006.