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 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 mainstream is being regarded as one of the greatest challenges in computer science in the last 50 years [5], with companies such as Microsoft and Intel investing heavily in academic research in this area.

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, blocks 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 for detecting conflicts and providing 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 conducted 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, etc.). 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 resolve 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 updates between frames. The key design decisions are (1) *tasks never abort*, and (2) data consistency requirements are specified by the programmer using object-oriented techniques.

The user plays an active role in controlling concurrency. First, she may restrict concurrency by enforcing specific task orderings using *task barriers*. Second, she may enable concurrent modifications of certain objects by specifying how to reconcile conflicts (using priorities or a general merge function).

This final strategy performed well. Although we ask the programmer to think quite a bit about data sharing and data consistency, we feel that there is a substantial value to the exercise as it leads to a more declarative and more concurrent programming style than traditional approaches. 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 detected 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 merge-based 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. Where our work differs from theirs is 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; rather it is based on the classic lock-based synchronization primitives.

As in our work, 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 enhanced 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, the game consists of two starships shooting one another in space. It 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 as-
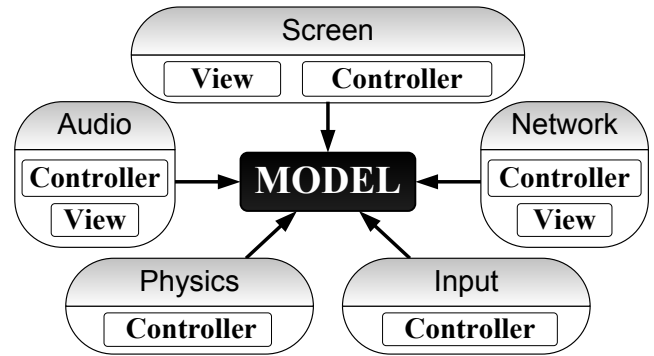

Figure 1. Our Model-View-Controller (MVC) design

teroids 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 workload.

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 occurred within 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. First, 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 also 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]

```
public class PhysicsController : Controller
{
  public void Start()
  {
    runtime.AddTask(this.UpdateCollisions);
  }
  public void UpdateCollisions()
  {
    ...
```

Figure 2. Controllers specify the periodic tasks. In each frame, the runtime calls all tasks concurrently, and waits for all of them to complete before proceeding to the next 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.

- 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

---

[2] For example, consider concurrent execution of the rendering task (which draws all game objects on the screen) and the physics task (which moves all objects individually, based on the elaped time). Fine grained locking led to noticeable inconsistencies as some objects would appear in their old position and others in their new position.

[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.

```
class Ship
{
   SharedValue<ShipState> state;
   SharedObject<Position> position;
   SharedList<Shot> shots;
   ...
```

Figure 3. We share data between tasks by using wrapper templates provided by the runtime library, as shown in these three examples.

```
void Shoot(Ship ship)
{
   if (ship.state.get() != ShipState.Dead)
      ship.shots.GetForWrite().Add(new Shot());
}

void Render(Ship ship)
{
   DrawShip(ship.position.GetReadonly());
   foreach(Shot shot in ship.shots.GetReadonly())
      DrawShot(shot);
}
```

Figure 4. Two method examples that show how to read and update the task-local replica.

its own replica of the world to work on. At the end of each frame, any updates made to the task-local replicas are propagated to all other replicas.

To share data between tasks, the user wraps it with templates provided by the runtime library (Fig. 3). The runtime then creates task-local replicas automatically and propagates updates at the end of each frame. We use different template variations to wrap values, objects, and collections, respectively.

Accessing the replicated data from within the tasks is straightforward (Fig. 4). For shared values, we use simple `get()` and `set(value)` methods to read or modify the task-local replica. For shared objects and collections, we gain access to the local replica by calling `GetReadonly()` or `GetForWrite()`, depending on whether we plan to modify the object.

By default, our runtime throws an exception if shared objects are modified by more than one task. To allow truly concurrent modification, our runtime currently supports three kinds of specialized wrappers (Fig. 5).

- The *SharedGrowingList* allows each task to add elements to the list, and merges all additions in the end. We used this data type for the list of asteroids, which may grow as a result of asteroids bursting into pieces.

- The *SharedObjectWithPriority* wrapper allows the user to associate a priority with each update. Conflicts are then resolved according to their priority. We used this wrapper template to express that incoming network packets

```
SharedObjectWithPriority<Position,Priority> pos;
SharedGrowingList<Asteroid> asteroids;
SharedObjectWithMerge<Sound> sound;
```

Figure 5. Specialized templates enable concurrent modification of shared data, as shown in these three examples. Conflicting updates are automatically reconciled at the end of the frame.

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

Figure 6. The programmer can specify barriers to enforce task order, as shown in these two examples.

update the opponent's ship position with a higher priority than the local simulation.

- The *SharedObjectWithMerge* captures the general case by supporting a user-specified merge. The wrapped data type must implement a merge method that computes the merged value given the initial value and two modified values. We used this wrapper for the sound data type, a bitmask whose bits may be set by various tasks. The merge function is a simple bitwise or.

Clearly, we are asking our users to make many choices. However, we believe that these choices are not boring and tedious, but represent important aspects of the implementation that are well worth documenting. Forcing the programmer to be explicit about intent does not only improve concurrency, but facilitates code maintenance. It may even be worthwile to adopt such techniques for purely sequential code. If there are many tasks that share data in undocumented ways, maintaining the correct semantics can be challenging even in a sequential game loop.

In addition to wrapper templates, our library supports *task barriers* (Fig. 6). If the user specifies a task barrier ordering task A before task B, then task B does not start until task A is finished, and all object updates performed by task A are propagated to task B.

4.1 Implementation and Optimizations

The runtime implementation takes care of managing the shared data (Fig. 7). It instantiates a fixed number of replicas of each shared object, equal to the number of tasks. Each task uses its (unique) task number to index into the array and access its own replica. In each frame, we record the set of shared objects each task 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, either by direct assignment (for shared values) or by calling an assignment method (for shared objects and collections). If an object was modified by more than one task, we pairwise reconcile the updates (using the priority, or

```
public class SharedValue<T>
 {
   private T[] value;
   public T get()
   {
     return value[my_task_id];
   }
   public void set(T newvalue)
   {
     value[my_task_id] = newvalue;
     Frame.MarkModified(this, my_task_id);
   }
   ...
```

Figure 7. A simplified illustration of our implementation, using a fixed-size array to contain the replicas. Modified replicas are marked, and propagated at the end of the frame.

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 8. 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.

using a merge function) 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 perform a copy-on-write optimization: rather than propagating updates eagerly at the end of the frame, we keep accessing the master read-only copy until the first time a replica is modified, at which time we perform the propagation.

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. 8. To visualize the results, we instrumented our prototype to capture and display task schedules. We show typical schedules in Fig. 9.

Experiment A (the sequential baseline) performs all tasks sequentially, using no replication and no synchronization.
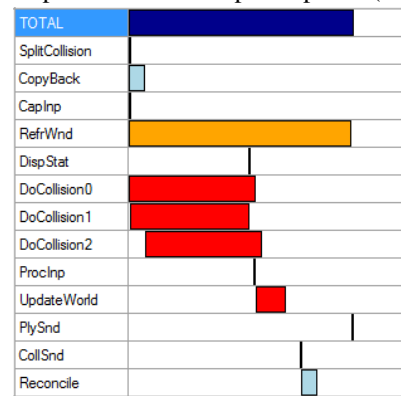
Experiment A: Single Replica (no concurrency)
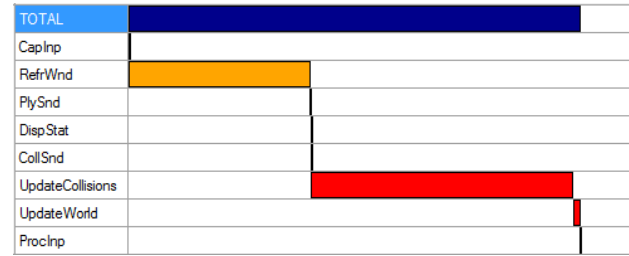
Experiment B: Two Replicas (partial concurrency)
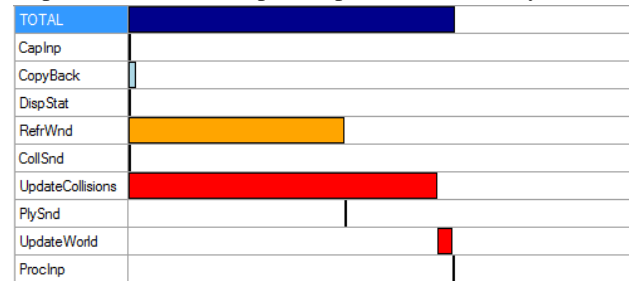
Experiment C: Multiple Replicas (full concurrency)

Figure 9. Typical task schedules for the three experiments on our 4-core machine.

The corresponding schedule in Fig. 9 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 that do not modify the model). The other replica is used by updater tasks (tasks that may modify the model), which are scheduled sequentially (one updater at a time). The corresponding schedule in Fig. 9 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. 9 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 simplify the programmer experience further (for instance, by supporting events and observer patterns), and to engineer a runtime prototype that scales to larger games with many thousands of game objects.

Acknowledgments

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.