Concurrent Programming with Revisions and Isolation Types

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

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

Daan Leijen
Microsoft Research
daan@microsoft.com

Abstract

Building applications that are responsive and can exploit parallel hardware while remaining simple to write, understand, test, and maintain, poses an important challenge for developers. In particular, it is often desirable to enable various tasks to read or modify shared data concurrently without requiring complicated locking schemes that may throttle concurrency and introduce bugs.

We introduce a mechanism that simplifies the parallel execution of different application tasks. Programmers declare what data they wish to share between tasks by using *isolation types*, and execute tasks concurrently by forking and joining *revisions*. These revisions are isolated: they read and modify their own private copy of the shared data only. A runtime creates and merges copies automatically, and resolves conflicts deterministically, in a manner declared by the chosen isolation type.

To demonstrate the practical viability of our approach, we developed an efficient algorithm and an implementation in the form of a C# library, and used it to parallelize an interactive game application. Our results show that the parallelized game, while simple and very similar to the original sequential game, achieves satisfactory speedups on a multicore processor.

Categories and Subject Descriptors 1.3 [*Concurrent Programming*]: Parallel Programming; 3.3 [*Language Constructs and Features*]: Concurrent Programming Structures

General Terms Languages

Keywords Concurrency, Parallelism, Transactions, Isolation, Revisions

1. Introduction

Despite much research on parallel programming, how to effectively build applications that enable concurrent execution of tasks that perform various functions and may execute asynchronously is not generally well understood. This problem is important in practice as a wide range of applications need to be responsive and would benefit from exploiting parallel hardware.

Consider an application where many tasks are executing in parallel. For example, an office application may concurrently run tasks that (1) save a snapshot of the document to disk, (2) react to keyboard input by the user who is editing the document, (3) perform a spellcheck in the background, and (4) exchange document updates with collaborating remote users. Some of these tasks are CPU-bound, others are IO-bound; some only read the shared data, others may modify it. However, all of them need to potentially access the same data at the same time; thus, they must avoid, negotiate, or resolve conflicts.

Ensuring consistency of shared data while allowing tasks to execute concurrently is often challenging, as it may require not only complex locking protocols, but also some form of data replication. We present a programming model that simplifies the sharing of data between such tasks. Its key design choices are:

Declarative Data Sharing. The programmer uses special *isolation types* to declare what data can be shared between concurrent tasks.

Automatic Isolation. Whenever the programmer forks an asynchronous task (we call these tasks *revisions*), it operates in isolation. Conceptually, each revision operates on a private copy of the entire shared state, which is guaranteed to be consistent and stable.

Deterministic conflict resolution. When the programmer joins a revision, all write-write conflicts (data that was modified both by the joinee and the joiner) are *resolved deterministically* as specified by the isolation type. For example, if there is a conflict on a variable of type *versioned*$\langle T \rangle$ (the most common isolation type), the value of the joinee always overwrites the value of the joiner. Deterministic conflict resolution never fails, thus revisions never "roll back".

These choices ensure deterministic concurrent program execution. Unlike conventional approaches, however, we do not require executions to be equivalent to some sequential

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

```
int x = 0;
task t = fork {
  x = 1;
}
assert(x = 0 ∨ x = 1);
join t;
assert(x = 1);
```

```
versioned⟨int⟩ x = 0;
revision r = rfork {
  x = 1;
}
assert(x = 0);
rjoin r;
assert(x = 1);
```

Figure 1. Comparison of a classic asynchronous task operating on a standard integer variable (left) and of a revision operating on a versioned variable (right). The assert statements show the possible values of x at each point.

execution, which would unnecessarily restrict the available concurrency. Instead, we posit that programmers, if given the right abstractions, are capable of reasoning about concurrent executions directly. For example, see Fig. 1. Under standard asynchronous task semantics (left), accesses to the integer variable by the main thread and the task are interleaved nondeterministically. But when working with revisions and a versioned type (right), the revision is guaranteed to work on an isolated copy. Effects of a revision are only seen once that revision is joined. In the meantime, other revisions can freely access that same data.

Our mechanism eliminates the need to perform any synchronization (such as critical sections) inside tasks. Each task is guaranteed to see a stable snapshot of the whole shared state, on which it can perform reads and writes at any time without risking blocking, interference, aborts, or retries, no matter how long it runs. Our approach is data-centric in the sense that it removes complexity from the tasks (which need no longer worry about synchronization) and adds it to the data declarations (which now need to specify the isolation type).

Our main contributions are:

- (Section 2) We present a precise description of the semantics of revisions and of the various isolation types. We introduce *revision diagrams* as a novel way to reason about visibility of effects.
- (Section 3) We review common implementation techniques for parallelizing application tasks and how they relate to revisions. In particular, we elaborate the semantic differences between revisions and transactions, and discuss related work.
- (Section 4) We describe a case study on how we parallelized an example application, a full-featured multiplayer game called SpaceWars3D [16]. We describe what changes were involved in parallelizing the game, and how we solved some of the more interesting issues. Moreover, we perform a quantitative evaluation in the form of measurements that compare the frame rates, the task execution times, and the overhead of accessing shared state between the sequential and the parallel version.
- (Section 5) We lift the covers and explain our runtime implementation (a C# library) and our optimized algorithm, which uses lazy copy-on-write, disposal of redundant replicas, and has a low overhead for accessing shared data.

Our results show that revisions and isolation types provide an elegant yet efficient mechanism for enabling parallelization of tasks in reactive or interactive applications.

2. Revisions and Isolation Types

In this section, we present a thorough abstract description of the semantics of the two ingredients of our method: revisions and isolation types. Our descriptions are informal, but we include a formal revision calculus in the appendix for reference.

2.1 Revisions

Revisions represent the basic unit of concurrency. They function much like asynchronous tasks that are forked and joined, and may themselves fork and join other tasks. We chose the term "revision" to emphasize the semantic similarity to branches in source control systems, where programmers work with a local snapshot of shared source code.

There are two important differences between revisions and asynchronous tasks. First, we consider the main thread that is executing the program to be a revision as well, called the main revision. Thus, all code executes inside some well-defined revision. Second, the programmer must explicitly join all revisions that she forked. This contrasts with asynchronous tasks for which the join is usually optional.

2.1.1 Revision Diagrams

One of the most important aspects of revisions is that they provide a precise way to reason about how tasks may see or not see each others' effects. To this end, we find it very helpful to visualize the concurrent control flow using *revision diagrams* (Fig. 2). Revisions correspond to vertical lines in the diagram, and are connected by horizontal arrows that represent the forks and joins. We sometimes label the revisions with the actions they perform. Such diagrams visualize clearly how information may flow (it follows the lines) and how effects become visible upon the join.

Note that our use of revision diagrams to reason about program executions is a marked departure from traditional concurrency models such as sequentially consistent memory or serializable transactions, which reason about concurrent executions by considering a set of corresponding totally ordered sequential histories. These traditional models make the fundamental assumption that programmers must think sequentially, and that all concurrency must thus be 'linearized' by some arbitration mechanism. However, such arbitration invariably introduces nondeterminism, which may easily present a much larger problem for programmers than direct reasoning about concurrent executions.

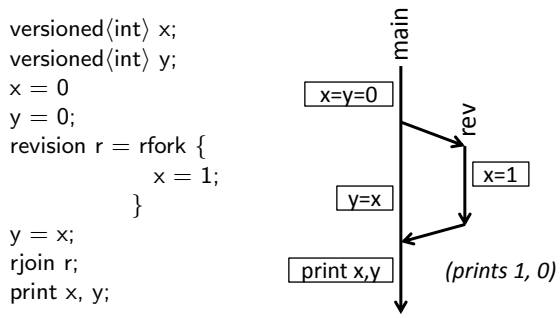


Figure 2. An example of a revision diagram (on the right) representing the execution of a program (on the left). Revisions correspond to vertical lines in the diagram, and are connected by horizontal arrows that represent the forks and joins. We label the revisions with the actions they perform.

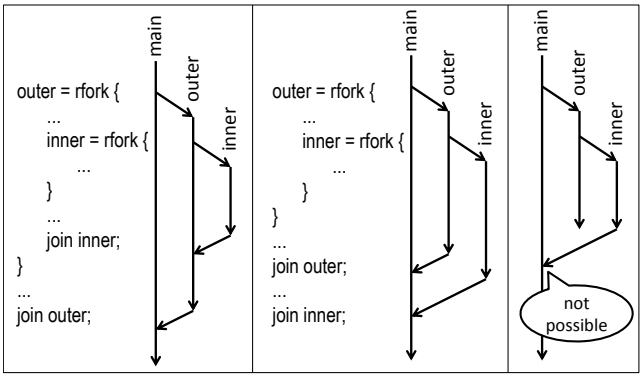


Figure 3. Revisions can be nested. Inner revisions may be joined before or after the outer revision is joined (left, middle). However, some diagrams are impossible (right) because the main revision can not join a revision before it has access to its handle (the handle returned by the second fork becomes accessible to the main revision only after the outer revision has been joined).

In contrast, we reason directly about the concurrent execution by visualizing concurrent effects and isolation guarantees using revision diagrams, while having deterministic join results.

2.1.2 Nested Revisions

To simplify modular reasoning about program executions, it is important to allow revisions to be nested. See Fig. 3 for examples on how revisions may or may not be nested. On the left, we see a revision that forks its own inner revision, then joins it. This corresponds to classical nesting of tasks. In the middle, we show how an inner revision "survives" the join of the revision that forked it, and gets subsequently joined by the main revision. On the right, we show that not all diagrams we can draw are actually possible, because a join can only join revisions for which it has a handle (the handle returned by the second fork becomes accessible to the main revision only after the outer revision has been joined).

Just like asynchronous tasks, revisions are a basic building block that can be used to express many different forms of concurrency or parallelism. Often, we may wish to first fork a number of revisions, and then immediately join all of them. This pattern is sometimes called the fork-join pattern and is common for divide-and-conquer algorithms. Revisions are more general though and their lifetime is not restricted by the lexical scope, and can for example be used to model long-running background tasks. Particularly, there is no implicit join at the end of each function as in the Cilk framework [10, 14, 32].

2.2 Isolation Types

When joining revisions, we wish to merge the copies of the shared data back together. Exactly how that should be done depends on what the data is representing, which can not be easily inferred automatically. We thus ask the programmer to explicitly supply this information by choosing an appropriate type for the data.

Choosing the right isolation type for every single shared object, field or variable may seem daunting at first. However, in our experience with parallelizing the game application we found that just a few isolation types cover almost all situations. Our isolation types fall into the following two major categories:

1. **Versioned** types. When joining versioned types, we first check whether the value has been modified in the revision since it was forked. If not, we do nothing. Otherwise, we change the current value of the revision that is performing the join to the current value of the revision that is being joined (Fig.4). For a basic type T, we write *versioned*$\langle T \rangle$ for the corresponding versioned type.

   In our game application, versioned types were the most common case. They are a good choice both for data on which concurrent modifications do not happen (many variables were concurrently written and read, but only a few were concurrently written to), or for situations in which there is clear relative priority between tasks (in the sense that some tasks should override the effects of other tasks).

2. **Cumulative** types. When joining cumulative types, the combined effect of modifications is determined by a general *merge function*. Such a function takes three values and returns a result value. The three arguments are the original value (value at the time when the revision was forked), the master value (current value in the revision that performs the join), and the revised value (current value in the revision that is being joined). For a basic type T, we write *cumulative*$\langle T, f \rangle$ for the corresponding cumulative type with merge function f.

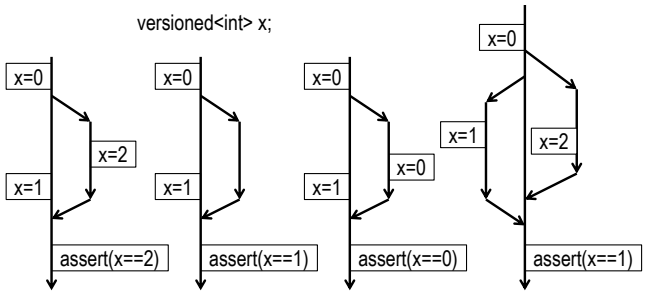


Figure 4. Revision diagrams illustrating the semantics of a versioned integer variable.

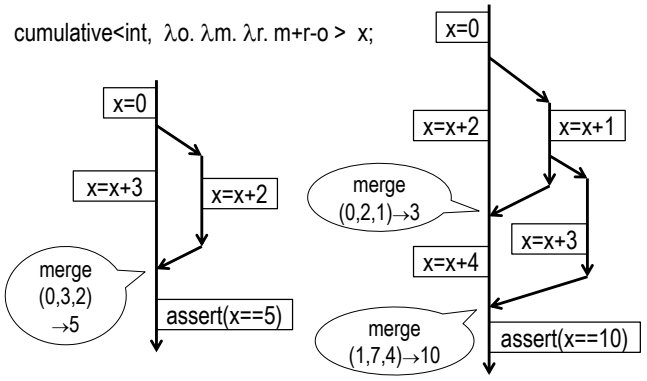


Figure 5. Revision diagrams illustrating the semantics of a cumulative integer variable.

For example, consider an integer variable to which several revisions add some quantity, and for which the cumulative effect of these additions should be the sum. We can then define the following merge function

```
int merge(int original, int master, int revised)
{
  return master + revised − original;
}
```

which produces the desired result (Fig. 5).

In our game application, we used cumulative types for collections (lists or sets) where tasks were adding elements concurrently.

An interesting aspect of using isolation types is the question of data granularity. Sometimes, the values of variables are correlated in the sense that they may be subject to some invariant. For example, valid values for the coordinate variables x and y may be restricted to the unit circle. Then, assigning only one of them may appear to locally preserve that invariant, while it is not globally preserved (Fig. 6). The solution is either to always assign both variables, or to group them together using a composite type.

Our definitions have been mostly informal and "by example". For reference and to remove potential ambiguities, we present a formal calculus for revisions and isolation types in the appendix.

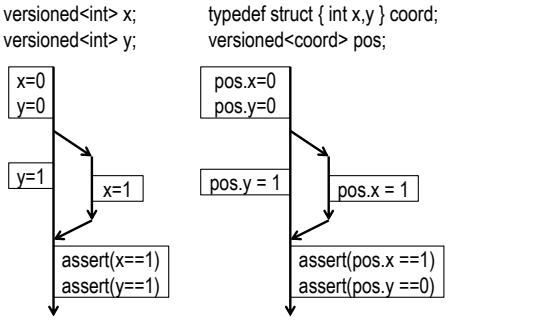


Figure 6. Revision diagrams that illustrate the difference between versioning two integer variables (on the left) and versioning a pair of integer variables (on the right).

3. Related Work

There has been much prior work on parallel or concurrent programming models and languages [1, 2, 6, 12, 14, 28, 32, 35, 36]. Most of them specialize on the expression of parallel algorithms. This purpose is somewhat different from our goal of facilitating the parallelization of applications like servers, games, or GUIs. Specifically, we assume that the different tasks we are trying to execute in parallel have the following characteristics:

- The tasks are coarse-grained.
- The tasks execute different code and have different data access patterns.
- The tasks exhibit many conflicts on the shared data.
- The tasks are mostly independent at a high abstraction level (i.e. most conflicts do not express true data dependencies).
- The tasks have varying characteristics (I/O vs. CPU bound) and may exhibit unbounded latencies.
- The tasks may need to react to external, nondeterministic events such as user input or network communication.

In this section, we review common implementation techniques and how they relate to revisions, provide a comparison between revisions and transactions, and discuss related work on isolation types.

3.1 Traditional Locking and Replication

Sometimes, standard locking schemes are appropriate for safely sharing data between tasks. However, locking complicates the code because it requires programmers to think about the placement of critical sections, which involves nontrivial tradeoffs and complicates code maintenance.

Moreover, locking alone does not always suffice. For example, consider a game application which executes concurrently (1) a physics task which updates the position of all

game objects based on their speed and the time elapsed, and (2) a render task which draws all objects onto the screen. Then, any solution based solely on locks would either hamper concurrency (too coarse) or provide insufficient isolation (too fine), as some of the objects may be rendered at the future position, while others are rendered at the current position.

For this reason, replication is often a necessary ingredient to achieve parallelization of application tasks. Games, for example, may maintain two copies of the shared state (using so-called *double-buffering*) to guarantee isolation of tasks while enabling any number of read-only tasks to execute concurrently with a single writer task. However, this pattern is somewhat specific to the synchronization structure of games, and maintaining just two buffers is not always enough (for example, there may be multiple concurrent modifications, or snapshots may need to persist for more than a single frame). Moreover, performing a full replication of the shared state is not the most space-efficient solution.

Another common replication-based solution is to use *immutable* objects to encode shared state. Any tasks that wish to modify an immutable object must instead create a copy. This pattern can efficiently guarantee isolation and enables concurrency. However, it can introduce new challenges, such as how to resolve conflicting updates, or how to bound space requirements in situations where frequent modifications to the data may cause excessive copying. Revisions solve both of these problems by implicitly linking the copying and merging to the concurrent control flow, and by using programmer-declared isolation types to resolve conflicts deterministically.

3.2 Related Work on Transactions

Like revisions, transactions or transactional memory [15, 21] address the problem of handling concurrent access to shared data. The key difference between transactions and revisions is that transactions (whether optimistic or pessimistic) handle conflicts nondeterministically, while revisions resolve conflicts deterministically. Moreover, revisions do not guarantee serializability, one of the hallmarks of transactions, but provide a different sort of isolation guarantee (as discussed above in Section 2). See Fig. 7 for an example that highlights the semantic difference between revisions and transactions.

Just as we do with revisions, proponents of transactions have long recognized that providing strong guarantees such as serializability [30] or linearizability [18] can be overly conservative for some applications, and have proposed alternate guarantees such as multi-version concurrency control [29] or snapshot isolation (SI) [3]. SI transactions (for example, see SI-STM [33]) are similar to revisions insofar as they operate on stable snapshots and do not guarantee serializability. However, they are more restricted as they do not perform deterministic conflict resolution (but rather abort transactions in schedule-dependent and thus nondeterministic way) and do not support nesting of transactions in a comparably general manner.

```
void foo()
{
  if (y = 0)
    x = 1;
}
```

```
void bar()
{
  if (x = 0)
    y = 1;
}
```

Revisions and Isolation Types:

```
versioned⟨int⟩ x,y;
x = 0; y = 0;
revision r = rfork { foo() ; }
bar();
rjoin r;
assert(x = 1 ∧ y = 1);
```

Transactions:

```
int x,y;
x = 0; y = 0;
task t = fork { atomic { foo(); } }
atomic { bar(); }
join t;
assert((x = 1 ∧ y = 0) ∨ (x = 0 ∧ y = 1));
```

Figure 7. Example illustrating the semantic difference between transactions and revisions. The assert statements indicate the possible final values, which are different in each case. The transactional program has two possible executions, both of which are different from the single (deterministic) execution of the program that uses revisions and isolation types.

Optimistic transactions do not fare well in the presence of conflicts that cause excessive rollback and retry. Moreover, combining optimistic transactions with I/O can be done only under some restrictions [38] because the latter cannot always be rolled back. None of these issues arises with revisions as they are not optimistic and never require rollback.

3.3 Related Work on Deterministic Concurrency

Recently, researchers have proposed programming models for deterministic concurrency [5, 8, 31, 37]. These models differ semantically from revisions, and are quite a bit more restrictive: as they guarantee that the execution is equivalent to some sequential execution, they cannot easily resolve all conflicts on commit (like revisions do) and must thus restrict tasks from producing such conflicts either statically (by type system) or dynamically (pessimistic with blocking, or optimistic with abort and retry). Also, unlike our revisions, some of these models[5, 8] allow only a restricted "fork-join" form of concurrency.

Hardware architects have also proposed supporting deterministic execution[4, 11]. However, these mechanisms guarantee determinism only, not isolation.

3.4 Related Work on Isolation Types

Isolation types are similar to Cilk++ hyperobjects [13]: both use type declarations by the programmer to change the semantics of shared variables. Cilk++ hyperobjects may split, hold, and reduce values. Although these primitives can (if properly used) achieve an effect similar to revisions, they do not provide a similarly seamless semantics. In particular, the determinacy guarantees are fragile, i.e. do not hold for all programs. For instance, the following program may finish with either $x == 2$ or $x == 1$:

```
reducer_opadd⟨int⟩ x = 0;
cilk_spawn { x++ }
if (x= 0) x++;
cilk_sync
```

Isolation types are also similar to the idea of transactional boosting, coarse-grained transactions, and semantic commutativity [17, 19, 20], which eliminate false conflicts by raising the abstraction level. Isolation types go farther though: for example, the type *versioned*$\langle T \rangle$ does not just avoid false conflicts, but resolves true conflicts deterministically.

Note that isolation types do not suffer from the weak- vs. strong-atomicity problem [7] because all code executes inside some revision.

The insight that automatic object replication can improve performance also appears in work on parallelizing compilers[34].

3.5 Related Work on Fork-Join Models

Once a revision is forked, its handle can be stored in arbitrary data structures and be joined at an arbitrary later point of time. The join is always explicitly requested by the programmer: this is important as it has side effects.

Some languages statically restrict the use of joins, to make stronger scheduling guarantees (as done in Cilk++ [14, 32]) or to simplify the most common usage patterns and to eliminate common user mistakes (as done in X10 [24]). In fact, many models use a restricted "fork-join" parallelism [5, 8]. In our experience, such restrictions (while reasonable for data-parallel problems) can make it difficult to write applications that adapt to external nondeterminism or to unpredictable latencies. In our game, for example, we wish to run the autosave task in the background as it has unpredictable latency, rather than forcing a join at the end of the frame.

4. Case Study

We now describe how we parallelized an example application using revisions and isolation types.

We first describe the sequential game application and why parallelization is a challenge. Next, we describe how we used revisions to parallelize the game loop and how we wrapped the shared data using isolation types. We also discuss how we address nondeterminism. Finally, we present experimental results that evaluate the performance characteristics of using revisions and isolation types and measure the gains from parallelization.

4.1 The Game

The game application is a multiplayer game called SpaceWars3D; it was designed to teach DirectX programming with C# [16]. Its conceptual architecture is depicted in Fig. 8 (top left). The square boxes represent tasks of varying sizes and characteristics (CPU-bound or IO-bound). The code amounts to about 12,000 lines. There is ample opportunity for executing different tasks in parallel and for parallelizing individual tasks. The key challenge is to ensure that the data is concurrently available, yet remains consistent.

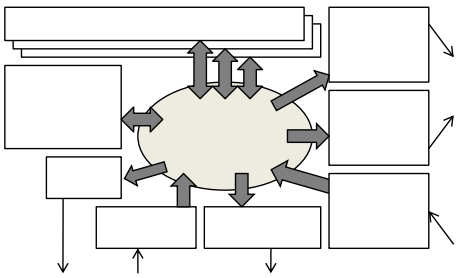
The starting point is a completely sequential game loop design shown in Fig. 8 (bottom left). It suffers from some major performance issues:[1]

1. (not parallel enough) There is room to parallelize tasks. For instance, the CollisionCheck(i) could be executed in parallel but are performed sequentially. Also, although the render task RenderFrameToScreen cannot itself be parallelized (due to restrictions in the framework), it can execute in parallel with other tasks.
2. (not responsive enough) The periodic automatic SaveGame call that occurs every 100 frames has unpredictable latency, and causes annoying freezes in the game experience.

To improve the frame rate and make the gameplay smoother, we would like to fix the issues above. However, there are numerous conflicts between these tasks that we need to pay attention to. For example, consider the coordinates of the game objects (like ships, bullets, asteroids, etc.). All of the following tasks may potentially access these coordinates at the same time:

- RenderFrameToScreen reads the position of all objects.
- UpdateWorld modifies the positions of all objects based on the elapsed time.
- CollisionCheck(i) reads the positions of all objects and may also modify some positions. These modifications are supposed to override the updates done by UpdateWorld.
- SendNetworkUpdates reads positions of local objects and sends them to the remote player.
- HandleQueuedPackets receives updates from the remote player and modifies positions of local objects. These updates are supposed to override the updates done by UpdateWorld. and by CollisionCheck(i).
- AutoSave reads the positions of all objects.

[1] Note that we modified the original game (by creating the asteroids and the autosave feature) for the specific purpose of causing performance issues for illustration purposes. Our naive collision check, while not representative for well-designed games that may employ the GPU and more sophisticated geometric partitioning, does illustrate the problem correctly.



```
while (!done)
{
    input.GetInput();
    input.ProcessInput();
    physics.UpdateWorld();
    for (int i = 0; i ⟨physics.numsplits; i++)
        physics.CollisionCheck(i);
    network.SendNetworkUpdates();
    network.HandleQueuedPackets();
    if (frame % 100 = 0)
        SaveGame();
    ProcessGuiEvents();
    screen.RenderFrameToScreen();
    audio.PlaySounds();
    frame++;
}
```

```
Revision UpWrl, SendNtw, HdlPckts, AutoSave;
Revision[] ColDet = new Revision[physics.numsplits];

while (!done)
{
    input.GetInput();

    UpWrl = rfork {
        input.ProcessInput(); physics.UpdateWorld();
    }
    for (int i = 0; i ⟨physics.numsplits; i++)
        ColDet[i] = rfork { physics.CollisionCheck(i); }
    SendNtw = rfork { network.SendNetworkUpdates(); }
    HdlPckts = rfork { network.HandleQueuedPackets(); )
    if (frame % 100 = 0 ∧ AutoSave = null)
        AutoSave = rfork { SaveGame(); };

    ProcessGuiEvents();
    screen.RenderFrameToScreen();

    join(UpWrl);
    for (int i = 0; i ⟨physics.numsplits; i++)
        join ColDet[i];
    join(SendNtw);
    join(HdlPckts);
    if (AutoSave ≠ null ∧ AutoSave.HasFinished()) {
        join(AutoSave);
        AutoSave = null;
    }

    audio.PlaySounds();
    frame++;
}
```

Figure 8. (top left) Illustration of the conceptual game architecture. **(bottom left)** Pseudocode for the sequential game loop. **(right)** Pseudocode for the parallel main loop.

All of the tasks are expected to work with a consistent view of the data. This can be challenging to achieve without some form of support from the framework.

Although tasks are sensitive to instability of the shared data, it is often acceptable to work with slightly stale data. For example, we could move SendNetworkUpdates to the top of the loop without harm, because it would simply send the positions of the last frame which is perfectly acceptable. This illustrates that the precise semantics of the sequential game loop are not set in stone: parallelization may make slight changes as long as the overall behavior of the game remains the same.

4.2 Parallelization

We now describe the process we performed to parallelize the game. It involved two main steps: Parallelizing the game loop using revisions, and declaring shared data using isolation types. This process involved making choices that require understanding of the semantics of the game: to achieve better parallelism, the parallel loop is not fully equivalent to the sequential loop, but only "close enough".

See Fig. 8 (right) for pseudocode[2] representing our parallel version of the game loop. All tasks are now inside concurrent revisions, except for four tasks that have to remain on the main thread because of restrictions of the GUI and graphics frameworks.

In each iteration, we fork revisions and store their handles. Each CollisionCheck(i) is in a separate revision. AutoSave only forks a revision every 100 frames, and only if there is not an autosave still in progress. After forking all revisions, the main thread performs the render task and processes GUI events. Then it joins all the revisions; however, it joins the autosave revision only if it has completed. Note that the concurrent revisions are joined in an order such that conflicting updates are correctly prioritized (collision check overrides update, network packets override both).

[2] We show pseudocode rather than the actual C# code for the sake of using a more concise syntax and omitting details unrelated to the point.

4.2.1 Declaring Isolation Types

We replaced a total of 22 types with isolation types. Identifying all the shared fields was a matter of identifying the "model" state (the game follows vaguely a model-view-controller architecture). Note that the majority of fields and variables do not need to be versioned (for example, they may be readonly, or may never be accessed concurrently). Overall, we used the following isolation types (we describe these types in more detail later, in Section 5.2), listed in the order of frequency:

- VersionedValue⟨T⟩ (13 instances). This was the most frequently used isolation type, and the type T ranged over all kinds of basic types including integers, floats, booleans, and enumerations.
- VersionedObject⟨T⟩ (5 instances). These were used for game objects such as photons, asteroids, particle effects, as well as for positions.
- CumulativeValue⟨T⟩ (3 instances). 2 instances were used for sound flags (which are essentially a bitmask implementation of a set), and one was used for a message buffer that displays messages on the screen.
- CumulativeList⟨T⟩ (1 instance). This was used for the list of asteroids; new asteroids are added when old ones burst, which happens on collisions.

4.2.2 Deterministic Record and Replay

At an abstract level, concurrent revisions do guarantee deterministic execution for correctly synchronized programs (that is, programs that join each revision they fork exactly once, and that do properly declare all shared data to have an isolation type).

In our parallel loop (Fig. 8 (right)) this guarantee does not hold completely, however, because we query whether the revision AutoSave has completed before joining it. Because timing varies between runs, this test does not always return the same result in each execution and thus introduces nondeterminism. This example showcases an important dilemma: if we want to enforce complete determinism, we cannot dynamically adapt to unpredictable latency variations. Thus, there is a fundamental tension between determinism and responsiveness.

We observe that there are in fact many sources of nondeterminism that quickly nullify deterministic execution even in the completely sequential game. Examples include user input, network packet timing, and random number generators. Thus, we decided to adjust our goal from 'deterministic execution' to 'deterministic record and replay'. By recording and replaying all sources of nondeterminism we can recover some of the benefits of determinism, such as a better debugging experience. Note that record/replay of revisions is much easier than record/replay of standard shared-memory programs[23] because there are only a few ordering facts that need to be recorded.

4.3 Quantitative Evaluation

In this section we analyze the performance characteristics of revisions in the parallelized SpaceWars3D game (Fig. 8, right). We are particularly interested in the following questions:

- What is the overhead of executing tasks inside revisions? This question is significant because in a revision, every read of a shared location must look up the correct version, and every first write to a shared location must create a copy.
- What is the overhead of forking and joining revisions? In particular, is it expensive to resolve conflicts during join?
- What is the memory overhead of maintaining replicas of the shared state?
- Did the gameplay experience improve? Specifically, (a) how much did the frame rate increase, and (b) did we successfully eliminate the freezes caused by the automatic periodic save?

4.3.1 Methodology

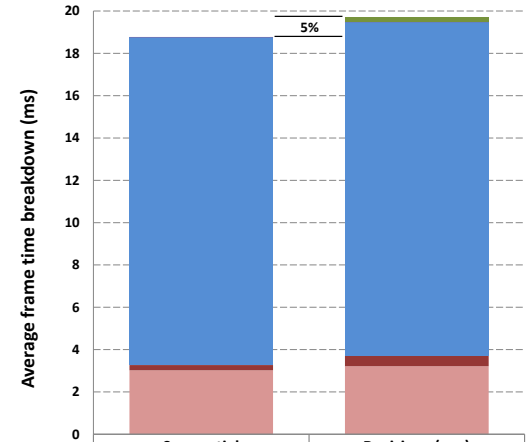
The evaluation methodology devised for the experiments takes advantage of the ability to record and replay a game as pointed out in Section 4.2.2. As the first step, we record 2500 frames of a network game session with two players chasing each other but without any asteroids being destroyed. We call such game session a *typical execution*. We exclude asteroid explosions from our initial experiments since they agressively decrease the number of frames executed per second (see Section 4.3.4 for an analysis with varying number of asteroids). We also disabled auto-save when recording.

After the game is saved, we replay it and examine only the last 2000 frames. The first 500 frames are skipped so that the runtime system can be appropriately warmed up. Notice that when replaying a game to collect the results only one machine is actually used. Although no network packages are sent during replay, we do record the packages received and replay them appropriately. Unless stated otherwise, all results are reported as the mean of five replays.

Time measurements are performed with the high-resolution performance counter provided by Windows, which is accurate within 383 nanoseconds in our workstation. While collecting timing information we only kept the game application opened in order to reduce external interferences.

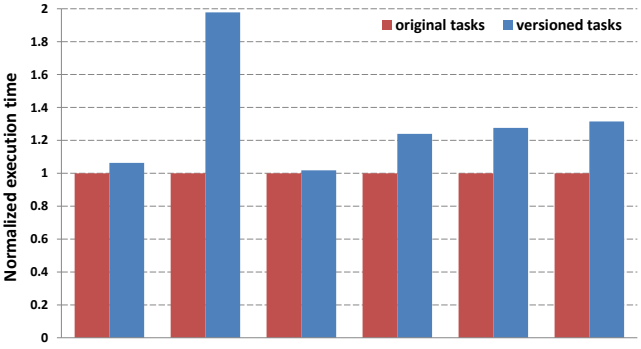
All experiments were conducted on a 4-core machine, an HP Z400 workstation running a 64-bit version of Windows 7 Enterprise, with 6GB of DDR3 RAM, a NVIDIA Quadro FX580 512MB graphics card, and a quad-core 2.66GHz Intel Xeon W3520 processor [3].

[3] Although hyperthreading is supported (totalizing 8 logical cores), this feature was disabled during our measurements.



	Sequential	Revisions (seq)
Join	0	0.239477711
Others	0.023597575	0.030098449
ColDet	15.48415	15.76237
UpWrl	0.2467515	0.4880668
Renderer	3.022258	3.214011

Figure 9. Average frame time breakdown. The overall slowdown due to revisions in sequential mode (on the right) is only 5% when compared to the pure sequential execution (on the left).



	Renderer	UpWrl	ColDet	Input	HdlPckts	PlaySnds
R/W	848/0	68/835	30986/0	0/2	2/18	3/3

Figure 10. Normalized execution time for each task in SpaceWars3D. At the bottom, the number of versioned reads and writes performed by each task. Baseline for normalization is the sequential execution.

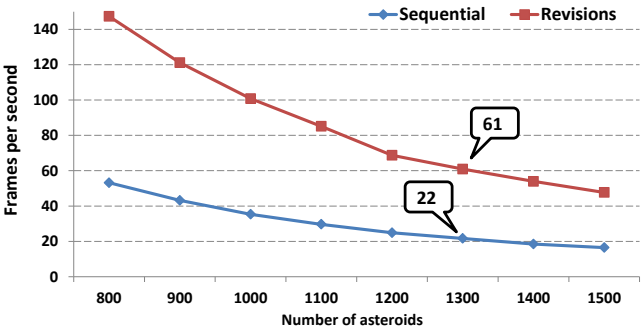


Figure 12. The frame rate drops as more asteroids are added to the game. With 1300 asteroids, the sequential version is mostly unplayable (22 FPS), whereas concurrent revisions still provide smooth gameplay (61 FPS) on a quad-core machine.

4.3.2 Revisions Overhead

To quantify the overhead of revisions in SpaceWars3D we recorded a typical execution of the game using 800 asteroids. This number was chosen because it provided a reasonable frame rate on our workstation (around 54 FPS in the single-threaded case).

We then replay the game and compare the results for two distinct execution modes: (i) sequential, and (ii) sequential with revisions. Scenario (i) serves as the sequential baseline to which revisions are compared. In scenario (ii), we enforce a sequential execution of the revisions by running them synchronously and performing the join operations at the end of the frame. This mode allows us to accurately quantify the single-thread overhead incurred by revisions for each game task (see Fig. 9 and 10).

As can be seen from Fig. 9, the revisioning framework only caused a slowdown of 5% if compared to the sequential execution. Two observations are worth mentioning. Firstly, the collision detection task (`ColDet`) accounts for approximately 82% of the total execution time per frame. This task, therefore, is the main target for parallelization as discussed in more detail in Section 4.3.4. Secondly, revisions incur an extra overhead since it is necessary to perform a join operation for each revision forked. Note, however, that this overhead contributed only 1.2% of the total execution time, showing that for the game application the join costs are mostly negligible.

To gain further insight into the performance of revisions we present the normalized execution time for each SpaceWars3D task in Fig. 10, along with the number of versioned reads and writes issued by each one. Notice that for the tasks dominated by reads (`Renderer` and `ColDet`) the overhead is mininum. This clearly shows that our current implementation is quite effective for read-based workloads. The remaining tasks suffer from versioned writes in different degrees, most notably `UpWrl` (almost 2x). However, since these tasks do not have a huge impact in the overall execution time, their cost is not a major concern in SpaceWars3D. For workloads where writing to versioned types dominates, we would expect higher overhead.

4.3.3 Memory Consumption

Every time a revision writes to a versioned object, the revisioning subsystem needs to create a clone in order to enforce isolation. It is therefore important to quantify the overhead of maintaining replicas of the shared state.

Table 1 presents the number of bytes of managed memory allocated by SpaceWars3D after a replay. We show

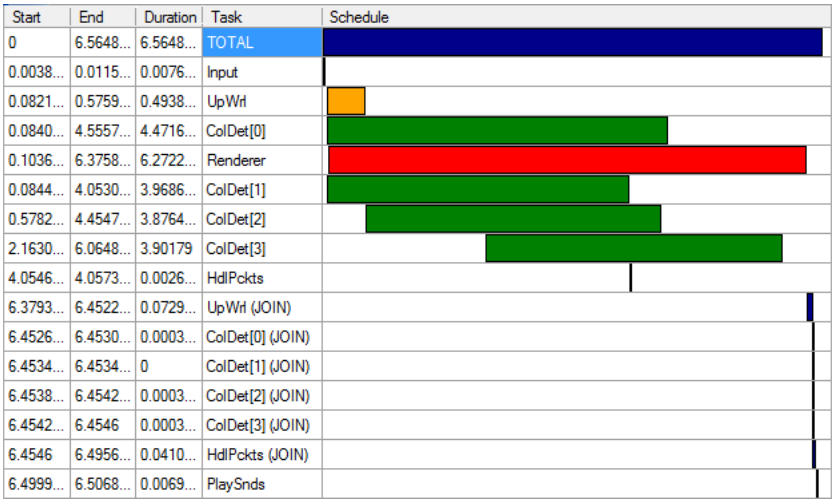


Figure 11. An illustration of a typical task scheduling for an individual frame on our quad-core machine. The extra cost incurred by the revisioning subsystem is represented by slightly longer-running tasks and the join operations.

# asteroids	Sequential	Revisions	Overhead
800	1,162,588	1,578,660	1.36
900	1,199,388	1,654,100	1.38
1000	1,236,196	1,734,200	1.40
1100	1,277,092	1,814,296	1.42
1200	1,324,932	1,914,504	1.44
1300	1,361,732	1,991,304	1.46
1400	1,398,532	2,068,104	1.48
1500	1,435,332	2,144,904	1.49

Table 1. Comparison between allocated managed memory with and without revisions (approximate value in bytes), for a varying number of asteroids.

the best available approximation as retrieved from a call to the .NET garbage collector method `GetTotalMemory` [4], for both the sequential version and revisions. As expected, the total amount of memory allocated increases as more asteroids are added to the game. The overhead of revisions slightly increases when compared to the sequential version due to more versioned objects being read/written.

4.3.4 Parallel Performance

We now proceed to analyze the performance of the game when executing the revisions concurrently on our quadcore machine. This performance depends on exactly how tasks are scheduled in each frame, which can vary quite a bit (our runtime does not perform the scheduling itself, but relies on an off-the-shelf dynamic task scheduler). We show a typical frame schedule in Fig. 11. Time proceeds from left to right, and the bars indicate when a task begins and ends. We also show bars for all the joins performed. Note that the bars do not show which tasks are currently scheduled vs. waiting to be scheduled, thus there are sometimes more than 4 tasks active at a time even though there are only 4 cores.

The schedule shown corresponded to parameters under which we achieved an average speedup of 2.6x relative to the sequential baseline.

How good is a speedup of 2.6x on 4 cores? Not bad for the circumstances. As we can see, the frame rate is limited mostly by the render task, which takes about 95.5% of the total frame time and can not be parallelized. Thus, even if everything else took zero time our speedup could not be better than $2.6 * 100/95.5 = 2.72$.

By increasing the number of asteroids, we change the proportion of the program that can be parallelized and achieve better speedups (Fig. 12), up to 3.03 for 1500 asteroids. We believe that even better speedups are possible with our library, for the following reason. Assuming a workload that is fully parallelizable except for the joins which amount to about 1.5%, Amdahl predicts a maximal speedup on 4 cores of $(1/(1 - 0.985 + 0.985/4) = 3.83$, so once we account for the average execution overhead of 5% the best possible speedup is still $3.83 * 0.95 = 3.64$. In this example, we do not achieve such good speedups even if we increase the number of asteroids to increase the proportion of parallelizable work. We are not sure why, but observe an unexpected slowness of the render task for which we believe some form of contention (outside of our runtime library) to be responsible.

Besides speeding up the average frame rate, our parallelization also improved the responsiveness: unlike in the sequential version of the game, the periodic automatic save had no perceptible effect on the gameplay.

[4] We allow the system to collect garbage and finalize objects before returning the number of bytes allocated.

```
class Versioned⟨T⟩ : Versioned {
  Map⟨int,T⟩ versions;
  ...
}

class Revision {
  Segment root;
  Segment current;
  ...
}

class Segment {
  int version;
  int refcount;
  Segment parent;
  List⟨Versioned⟩ written;
  ...
}
```

Figure 13. A quick overview of the classes our algorithm is built on, and how they relate.

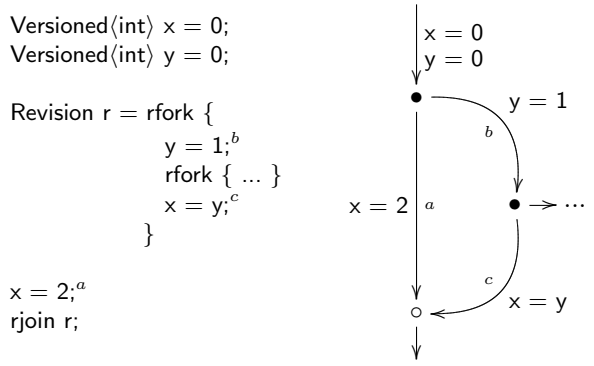


Figure 14. Example revision diagram.

5. Implementation

We now describe how we actually implemented revisions and isolation types. The key design principles of our algorithm are:

- The (amortized) cost of a `Get` or `Set` operation on a versioned object must be very efficient, as it is called every time a revision accesses shared data.
- To save time and space, we must not copy data eagerly (such as on every fork), but lazily and only when necessary (that is, when the write may invalidate somebody else's snapshot).
- We must release copies that are no longer needed as soon as possible.

In this section we first describe our algorithm that satisfies these requirements, starting with a stripped-down version in pseudo object-oriented code. Then we describe various extensions and optimizations that we used to implement our C# library.

5.1 The Essential Algorithm

See Fig. 13 for a quick overview of the three classes we use and how they relate. We give detailed listings of these classes in Figures 16, 17 and 18, but first discuss the high-level idea.

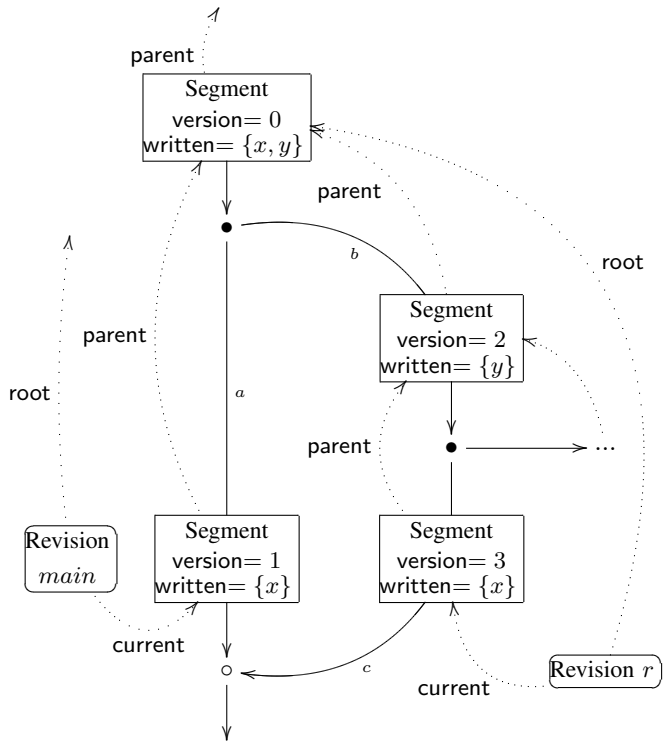


Figure 15. The state of our implementation for the example in Figure 14, right before the rjoin r statement. For illustration purposes, We show the Segment objects superimposed on the respective segments of the revision diagram.

Revision objects represent revisions as defined earlier. Recall that in revision diagrams, revisions are the vertical lines, which consist of one or more line segments separated by forks or joins. Revision objects are created when a revision is forked, and released after a revision is joined. Each revision object has a current segment (the currently last segment of this revision) and a root segment (the line segment right above the fork that created this revision).

Segment objects correspond to vertical line segments in the revision diagrams, and are uniquely identified by their version number (the field version). Segment objects form a tree (by parent pointer). Segment objects are created when line segments are added (each fork creates two new segments, each join creates one new segment), and are released when refcount reaches zero. Segment objects also maintain a list of all Versioned objects[5] that were written to in this segment.

Versioned objects contain a versions map that stores several versions of the data, indexed by version numbers. It stores for each line segment the last value written to this

[5] The Versioned⟨T⟩ class derives from the Versioned class so we can create a homogenous list of non-generic versioned objects.

object in that segment, or the special value $\perp$ to signify that there was no write to this object in that segment.

In the next few subsections, we are going to discuss several aspects of our implementation in more detail. First though, we show a small example program and its revision diagram in Fig. 14, which will serve as a running illustration example. We labeled some of the segments with a, b, and c, and assume they will have version numbers 1, 2, and 3 respectively. To explain the design of the algorithm, we now discuss the state of our implementation right before the rjoin r statement. At that point, we have two Revision objects and five Segment objects, and they are related as shown in Fig. 15. At the same point of time, the versions map for the variable x is $\{0 \mapsto 0, 1 \mapsto 2, 3 \mapsto 1\}$, and the map for y is $\{0 \mapsto 0, 2 \mapsto 1\}$. As we can see, only the last writes to a value are in the versions map; i.e. even though y is read in the c edge, there is no entry for version 3 in the versions map of y.

5.1.1 Accessing Versioned Data

To access versioned data, we use the public Get and Set methods in Fig. 16. These methods first consult the thread-local static field Revision.currentRevision (see Fig. 17) to automatically find the correct revision for the current thread.

The Get method then returns the current value associated with a particular revision. It cannot just return the content of versions[r.current.version] since only the last write is stored in this map. If the revision has not written to this particular object, we need to follow the parent chain to find the last write.

The Set method sets the current value for a particular revision. It first looks to see if the entry for the current segment is uninitialized. If so, it adds this versioned object to the written list of the segment, before writing the new value to the map.

5.1.2 Fork

The Fork operation (Fig. 17) starts with creating a fresh revision r for the forked off branch. We first create a new revision using our current segment as its root, and then create a new current segment. For example, in Fig. 15 we create segments with version numbers 1 and 2. After creating a new revision r, we create a new concurrent task that assigns the new revision to the thread local currentRevision. Here we assume that Task.StartNew starts a new concurrent task with the provided action delegate (anonymous function). Light-weight concurrent tasks based on work stealing are provided by .NET 4.0; on other frameworks we can use any similar kind of way to start concurrent threads [22, 25, 27]. Finally, the new revision r is returned such that it can be joined upon later.

```
class Versioned {
  void Release();
  void Collapse(Revision main, Segment parent);
  void Merge(Revision main, Revision joinRev, Segment join);
}

public class Versioned⟨T⟩ : Versioned {
  Map⟨int,T⟩ versions; // map from version to value

  public T Get() { return Get(Revision.currentRevision); }
  public void Set(T v) { Set(Revision.currentRevision, v); }

  T Get(Revision r) {
    Segment s = r.current;
    while (versions[s.version] = ⊥) {
      s = s.parent;
    }
    return versions[s.version];
  }

  void Set(Revision r, T value) {
    if (versions[r.current.version] = ⊥) {
      r.current.written.Add(this);
    }
    versions[r.current.version] = value;
  }

  void Release( Segment release ) {
    versions[release.version] = ⊥;
  }

  void Collapse( Revision main, Segment parent ) {
    if (versions[main.current.version] = ⊥) {
      Set(main, versions[parent.version]);
    }
    versions[parent.version] = ⊥;
  }

  void Merge(Revision main, Revision joinRev, Segment join) {
    Segment s = joinRev.current;
    while (versions[s.version] = ⊥) {
      s = s.parent;
    }
    if (s = join) { // only merge if this was the last write
      Set(main, versions[join.version]);
    }
  }
}
```

Figure 16. The Versioned class.

5.1.3 Join

The Join operation (Fig. 17) first waits till the associated concurrent task of the revision is done. Note that if an exception is raised in the concurrent task, it is re-raised in the call to Wait and in that case we will not merge any changes. When Wait succeeds, the actual written objects in the join revision are merged.

```
public class Revision {
  Segment root;
  Segment current;
  Task task;
  threadlocal static Revision currentRevision;

  Revision( Segment root, Segment current ) {
    this.root = root;
    this.current = current;
  }

  public Revision Fork( Action action ) {
    Revision r;
    r = new Revision(current, new Segment(current));
    current.Release(); // cannot bring refcount to zero
    current = new Segment(current);
    task = Task.StartNew( delegate (){
      Revision previous = currentRevision;
      currentRevision = r;
      try { action(); }
      finally { currentRevision = previous; }
    });
    return r;
  }

  public void Join(Revision join) {
    try {
      join.task.Wait();
      Segment s = join.current;
      while (s ≠ join.root) {
        foreach (Versioned v in s.written) {
          v.Merge(this,join,s);
        }
        s = s.parent;
      }
    }
    finally {
      join.current.Release();
      current.Collapse(this);
    }
  }
}
```

Figure 17. The Revisioned class.

In a while loop, we visit each segment from join.current upto its root. If we look at our example in Figure 15, joining on r would visit the segments with versions 3 and 2. Indeed, together the written lists of those segments contain all objects that need to be merged back. For each segment, we iterate over all written objects and call their Versioned⟨T⟩.Merge method with three arguments: the main revision, the joined revision, and the current segment. When we look at the implementation of that method in Figure 16 we see that it first finds the first segment that wrote to this object. Only if the merged segment join is the same will we do a merge. If the merged segment is not equal, it means that that segment did not do the last write to that object and we should not merge older versions. If this happens to be the last write, we merge by simply overwriting the value in the main revision (if it exists).

```
class Segment {
  Segment parent;
  int version;
  int refcount;
  List⟨Versioned⟩ written;
  static int versionCount = 0;

  Segment( Segment parent ) {
    this.parent = parent;
    if (parent ≠ null) parent.refcount++;
    written = new List⟨Versioned⟩();
    version = versionCount++;
    refcount = 1;
  }

  void Release() {
    if (--refcount = 0) {
      foreach (Versioned v in written) {
        v.Release(this);
      }
      if (parent ≠ null) parent.Release();
    }
  }

  void Collapse( Revision main ) {
    // assert: main.current = this
    while (parent ≠ main.root ∧ parent.refcount = 1) {
      foreach (Versioned v in parent.written) {
        v.Collapse(main,parent);
      }
      parent = parent.parent; // remove parent
    }
  }
}
```

Figure 18. The Segment class. For the purpose of reference counting, we assume that ++ and -- are atomic operations.

Finally, the Join function releases the reference count on the joined revision, and calls Collapse on our current segment. We describe these situations in more detail in the following two sections.

5.1.4 Releasing Segments

Each Segment object (Fig. 18) maintains a refcount to keep track of how many parent and current fields are pointing at that segment (it does not count the root fields). The Release method is called by revisions to decrease the reference count, and whenever the reference count drops to zero, we can release any objects referenced by this version.

Since only written objects are stored in the versions map of Versioned⟨T⟩, the objects referenced by the version of the segment are exactly those that are in its written list. The Release method calls the Versioned⟨T⟩.Release method

on each of the objects in its written list and then releases its parent segment. When we look at the Versioned⟨T⟩.Release method in Figure 16 we see that it simply clears the entry for that object in the versions map. In our example in Figure 15 the segment with version 3 will be released and the versions map of x will become $\{0 \mapsto 0, 1 \mapsto 1\}$ after the join. Note that the map for y becomes $\{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 1\}$ since the segment for version 2 is *not* released as the inner forked revision could potentially still refer to that version of y.

5.1.5 Collapsing Segments

The Collapse method (Fig. 18) is only ever called on some current segment and it is the case that main.current = this when this method is called from Revision.Join. The Collapse method tries to merge the parent segment into the current segment. In particular, when we join on some revision, we might find that our parent segment has a reference count of 1 and that we are the only one holding on to it. By collapsing with that segment we both reduce the chain of segments (which improves reads), but more importantly, we might release older versions of objects that are never referenced again. The Collapse operations ensures that we do not leak memory over time.

Collapse visits the parent recursively while the reference count is 1. For each written object in the parent we call Versioned⟨T⟩.Collapse on that object with the current revision and the parent segment. After visiting each written object, we overwrite the parent field with the parent of the parent, effectively removing the parent segment (which is now collapsed into the current revision). The implementation of Versioned⟨T⟩.Collapse is shown in Figure 16. If the current revision has not written this object yet, we set it to the value of the parent revision. Finally, the parent version is cleared releasing its reference.

5.2 Additional Isolation Types

Our presentation of the algorithm includes the single isolation type Versioned⟨T⟩ only. This type is actually called VersionedValue⟨T⟩ in our library, which contains a variety of isolation types. For example, the type CumulativeValue⟨T⟩ enables users to specify a specific merge function. This merge function needs to know the original snapshot value, which our implementation can access by following the Revision.root pointer of the revision being joined.

For reference values, we implement VersionedObject⟨T⟩ and CumulativeObject⟨T⟩, which version all fields of an object as a whole (cf. Fig. 6). To access such objects, Get and Set are not appropriate, but we use similar operations

```
T GetForRead( Revision r );
T GetForWrite( Revision r );
```

where GetForRead is used to get a readonly reference to an object, while GetForWrite is used to get a mutable version of the object. Ideally, if natively supported by a language, the use of these operations could be hidden and inserted automatically by the compiler.

Beyond those isolation types, our library also supports the cumulative collection classes CumulativeList⟨T⟩ and CumulativeSet⟨T⟩ with their natural merge functions, and a VersionedRandom class that serves as a deterministic pseudorandom generator.

5.3 Optimization

We employ a number of optimizations:

- We use a specialized mostly lock-free implementation for the versions map. It uses arrays that may be resized if necessary.
- To further speed up the Get operation, we maintain a cache that contains the version and corresponding index of the last read or write to this object. It is implemented as a 32 bit word that contains a version number in the lower 16 bits, and an index in the upper 16 bits. By keeping it the size of a word, we can atomically read and write this cache without using locks.
- When forking a new revision, we first check if the current segment contains any writes. If not, it can stay the current segment, and we can use its parent as the parent of the new segment.
- When merging objects, we can distinguish many special cases that can be handled a bit faster. In our optimized implementation, the Versioned⟨T⟩.Merge function is the most complicated part, consisting of eight separate cases. Partly the complexity is due to the application of merge functions for cumulative objects, and partly because we release slots directly during the merge and try to reuse and update slots in-place whenever possible.

6. Conclusion and Future Work

We have presented a novel programming model based on revisions and isolation types. First, we explained what guarantees it provides, using revision diagrams as the basic means of reasoning. Then we demonstrated how an example game application can take advantage of this model. Finally, we elaborated on how to build an efficient runtime library.

Our results show that revisions and isolation types provide an elegant yet efficient mechanism for executing different tasks within a reactive or interactive application.

As future work, we would like to apply revisions and isolation types to more general settings, such as applications that execute some tasks on the GPU, applications that run on many-core processors without full shared-memory guarantees, or applications that run in the cloud.

Acknowledgments

We thank Tom Ball, Ben Zorn, and Tim Harris for helpful comments and discussions.

References

[1] S. Aditya, Arvind, L. Augustsson, J.-W. Maessen, and R. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In Paul Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, pages 35–49, June 1995.

[2] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. Steele Jr. Project fortress: A multicore language for multicore processors. In *Linux Magazine*, September 2007.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of SIGMOD*, pages 1–10, 1995.

[4] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[5] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for c/c++. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[6] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[7] C. Blundell, E. Lewis, and M. Martin. Deconstructing transactions: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2005.

[8] R. Bocchino, V. Adve, D. Dig., S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[9] S. Burckhardt and D. Leijen. Semantics of concurrent revisions (full version). Technical Report MSR-TR-2010-94, Microsoft, 2010.

[10] J. Danaher, I. Lee, and C. Leiserson. The jcilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005.

[11] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared-memory multiprocessing. *Micro, IEEE*, 30(1):40 –49, jan.-feb. 2010.

[12] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *Rice University*, pages 209–220, 1995.

[13] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 79–90, 2009.

[14] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Programming Language Design and Impl. (PLDI)*.

[15] T. Harris, A. Cristal, O. Unsal, E. Ayguadé, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.

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

[17] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 2008.

[18] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[19] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Principles of Programming Languages (POPL)*, pages 19–30, 2010.

[20] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *Programming Language Design and Impl. (PLDI)*, 2007.

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

[22] D. Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[23] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[24] J. Lee and J. Palsberg. Featherweight x10: a core calculus for async-finish parallelism. In *Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[25] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[26] A. Martin, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Principles of Programming Languages (POPL)*, pages 63–74, 2008.

[27] Microsoft. Parallel extensions to .NET. `http://msdn.microsoft.com/en-us/concurrency`, June 2009.

[28] L. Moreau. The semantics of scheme with future. In *In In ACM SIGPLAN International Conference on Functional Programming (ICFP'96*, pages 146–156, 1996.

[29] P.A.Bernstein and N.Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[30] P.A.Bernstein, V.Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[31] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for java futures. *SIGPLAN Not.*, 39(10):206–223, 2004.

[32] K. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[33] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2006.

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

[35] V. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[36] G. Steele. Parallel programming and parallel abstractions in fortress. In *Invited talk at the Eighth International Symposium on Functional and Logic Programming (FLOPS)*, April 2006.

[37] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 439–453, 2005.

[38] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 285–296, 2008.

A. Revision Calculus

For reference and to remove potential ambiguities, we now present a formal calculus for revisions and isolation types. It is based on a similar calculus introduced by prior work on AME (automatic mutual exclusion) [26].

Before looking at the calculus, let us introduce a few notations we use to work with partial functions. For sets A, B, we write $A \rightharpoonup B$ for the set of partial functions from A to B. For $f, g \in A \rightharpoonup B$, $a \in A$, $b \in B$, and $A' \subset A$, we adopt the following notations: $f(a) = \perp$ means $a \notin dom f$, $f[a \mapsto b]$ is the partial function that is equivalent to f except that $f(a) = b$, and $f :: g$ is the partial function that is equivalent to g on $dom\ g$ and equivalent to f on $A \setminus dom\ g$. In our transition rules, we use patterns of the form $f(a_1 : b_1) \ldots (a_n : b_n)$ (where $n \geq 1$)) to match partial functions f that satisfy $f(a_i) = b_i$ for all $1 \leq i \leq n$.

We show the syntax and semantics of our calculus concisely in Fig. 19. The syntax (top left) represents a standard functional calculus, augmented with references. References can be created (*ref* e), read ($!e$) and assigned ($e := e$). The result of a fork expression *rfork* e is a revision identifier from the set *Rid*, and can be used in a *rjoin* e expression (note that e is an expression, not a constant, thus the revision being joined can vary dynamically).

To define evaluation order within an expression, we syntactically define execution contexts (Fig. 19 right column, in the middle). An execution context $\mathcal{C}$ is an expression "with a hole", and as usual we let $\mathcal{C}[e']$ be the expression obtained from $\mathcal{C}$ by replacing the hole with e'.

The operational semantics (Fig. 19, bottom) describes transitions of the form $s \rightarrow_r s'$ which represent a step by revision r from global state s to global state s'. Consider first the definition of global states (Fig. 19, top right). A global state is defined as a partial function from revision identifiers to local states: there is no shared global state. The local state has three parts (σ, τ, e): the snapshot σ is a partial function that represents the initial state that this revision started in, the local store τ is a partial function that represents all the locations this revision has written to, and e is the current expression.

The rules for the operational semantics (Fig. 19, bottom) all follow the same general structure: a transition $s \rightarrow_r s'$ matches the local state for r on the left, and describes how the next step of revision r changes the state.

The first three rules (apply, if-true, if-false) reflect standard semantics of application and conditional. They affect only the local expression. The next three rules (new, get, set) reflect operations on the store. Thus, they affect both the local store and the local expression. The (new) rule chooses a fresh location (we simply write $l \notin s$ to express that l does not appear in any snapshot or local store of s). The last two rules reflect synchronization operations. The rule (fork) starts a new revision, whose local state consists of (1) a snapshot that is initialized to the current state $\sigma :: \tau$, (2) a local store that is the empty partial function, and (3) an expression that is the expression supplied with the fork. Note that (fork) chooses a fresh revision identifier (we simply write $r \notin s$ to express that r is not mapped by s, and does not appear in any snapshot or local store of s). The rule (join) updates the local store of the revision that performs the join by merging the snapshot, master, and revision states (in accordance with the declared isolation types), and removes the joined revision. It can only proceed if the revision being joined has executed all the way to a value (which is ignored).

As usual, we define the step relation $\rightarrow$ to be the union of the local step relations $\rightarrow_r$. We call a global state s an *initial state* if it is of the form $s = \{(r, (\epsilon, \epsilon, e)\}$. We call a sequence of steps $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$ an *execution* if s_0 is an initial state, and *maximal* if there exists no s' such that $s_n \rightarrow s'$.

We refer readers that are interested in additional details to our Tech Report [9] where we discuss various properties of variations of this calculus, such as determinacy, and that the revision diagrams are semilattices.

Syntactic Symbols

$$
\begin{array}{rcll}
v & \in & \mathit{Value} & = & c \mid l \mid r \mid \lambda x.e \\
c & \in & \mathit{Const} & = & \mathit{unit} \mid \mathit{false} \mid \mathit{true} \\
l & \in & \mathit{Loc} & & \\
r & \in & \mathit{Rid} & & \\
x & \in & \mathit{Var} & & \\
e & \in & \mathit{Expr} & = & v \mid x \\
 & & & \mid & e\ e \mid (e\,?\,e : e) \\
 & & & \mid & \mathit{ref}\ e \mid\ !e \mid e := e \\
 & & & \mid & \mathit{rfork}\ e \mid \mathit{rjoin}\ e
\end{array}
$$

State

$$
\begin{array}{rcrcl}
s & \in & \mathit{GlobalState} & = & \mathit{Rid} \rightharpoonup \mathit{LocalState} \\
 & & \mathit{LocalState} & = & \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr} \\
\sigma & \in & \mathit{Snapshot} & = & \mathit{Loc} \rightharpoonup \mathit{Value} \\
\tau & \in & \mathit{LocalStore} & = & \mathit{Loc} \rightharpoonup \mathit{Value}
\end{array}
$$

Execution Contexts

$$
\begin{array}{rcl}
\mathcal{C} & = & [\,] \\
 & \mid & \mathcal{C}\ e \mid v\ \mathcal{C} \mid (\mathcal{C}\,?\,e : e) \\
 & \mid & \mathit{ref}\ \mathcal{C} \mid\ !\mathcal{C} \mid \mathcal{C} := e \mid l := \mathcal{C} \\
 & \mid & \mathit{rjoin}\ \mathcal{C}
\end{array}
$$

Operational Semantics

$$
\begin{array}{lrcll}
\text{(apply)} & s(r : \langle \sigma, \tau, \mathcal{C}[\lambda x.e\ v] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{C}[[v/x]e] \rangle] & \\
\text{(if-true)} & s(r : \langle \sigma, \tau, \mathcal{C}[(\mathit{true}\,?\,e : e')] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{C}[e] \rangle] & \\
\text{(if-false)} & s(r : \langle \sigma, \tau, \mathcal{C}[(\mathit{false}\,?\,e : e')] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{C}[e'] \rangle] & \\
\text{(new)} & s(r : \langle \sigma, \tau, \mathcal{C}[\mathit{ref}\ v] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{C}[l] \rangle] & \text{if } l \notin s \\
\text{(get)} & s(r : \langle \sigma, \tau, \mathcal{C}[!l] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{C}[(\sigma{::}\tau)(l)] \rangle] & \text{if } l \in \mathit{dom}\ \sigma{::}\tau \\
\text{(set)} & s(r : \langle \sigma, \tau, \mathcal{C}[l := v] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{C}[\mathit{unit}] \rangle] & \\
\text{(fork)} & s(r : \langle \sigma, \tau, \mathcal{C}[\mathit{rfork}\ e] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{C}[r'] \rangle][r' \mapsto \langle \sigma{::}\tau, \epsilon, e \rangle] & \text{if } r' \notin s \\
\text{(join)} & s(r : \langle \sigma, \tau, \mathcal{C}[\mathit{rjoin}\ r'] \rangle)(r' : \langle \sigma', \tau', v \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \mathit{merge}(\sigma', \tau, \tau'), \mathcal{C}[\mathit{unit}] \rangle][r' \mapsto \bot] &
\end{array}
$$

$$
\text{where} \quad \mathit{merge}(\sigma', \tau, \tau')(l) \quad = \quad \begin{cases} \tau(l) & \text{if } \tau'(l) = \bot \\ \tau'(l) & \text{if } \tau'(l) \neq \bot \text{ and } l \text{ is of type } \mathit{versioned}\langle T \rangle \\ f(\sigma'(l), \tau(l), \tau'(l)) & \text{if } \tau'(l) \neq \bot \text{ and } l \text{ is of type } \mathit{cumulative}\langle T, f \rangle \end{cases}
$$

Figure 19. Syntax and Semantics of the revision calculus.