Based on the article ”Concurrent Programming with Revisions and Isolation Types”, OOPSLA ’10

Sebastian Burckhardt, Microsoft Research
Alexandro Baldassin, State University of Campinas, Brazil
Daan Leijen, Microsoft Research

Concurrent Programming with Revisions and Isolation Types

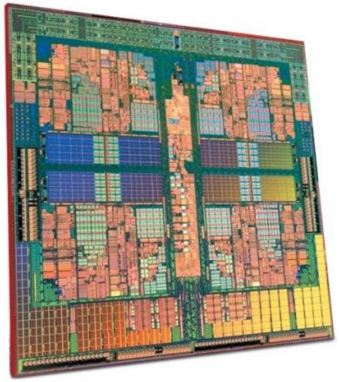
Prepared by Rafał Bereski, MIMUW

Outline

- **Introduction**
  - Main ideas
  - Assumptions of the presented programming model
- **Revisions and Isolation Types**
- **Comparison to the classic locking schemes**
- **Case Study** (SpaceWars3D game)
- **Performance evaluation**
- **Implementation details**
- **Summary**

Introduction

- Applications need to be responsive and benefit from exploiting parallel hardware.
- However, ensuring consistency of the data shared between concurrent tasks is often challenging.



Main ideas

- Reading and modifying data concurrently without complicated locking schemes that may introduce bugs.
- Instead of synchronizing access to the data, enable tasks to do real parallel modifications by creating isolated copy of the data (replication).
- Making concurrent programs as easy to code as thier sequential versions.

Programming model

- Article describes programming model that simplifies sharing data between asynchronous tasks.

- **Declarative Data Sharing**
  - Programmer declares which variables are shared between tasks.
  - ***Isolation Types***

- **Automatic Isolation**
  - Task operates on that data in isolation.

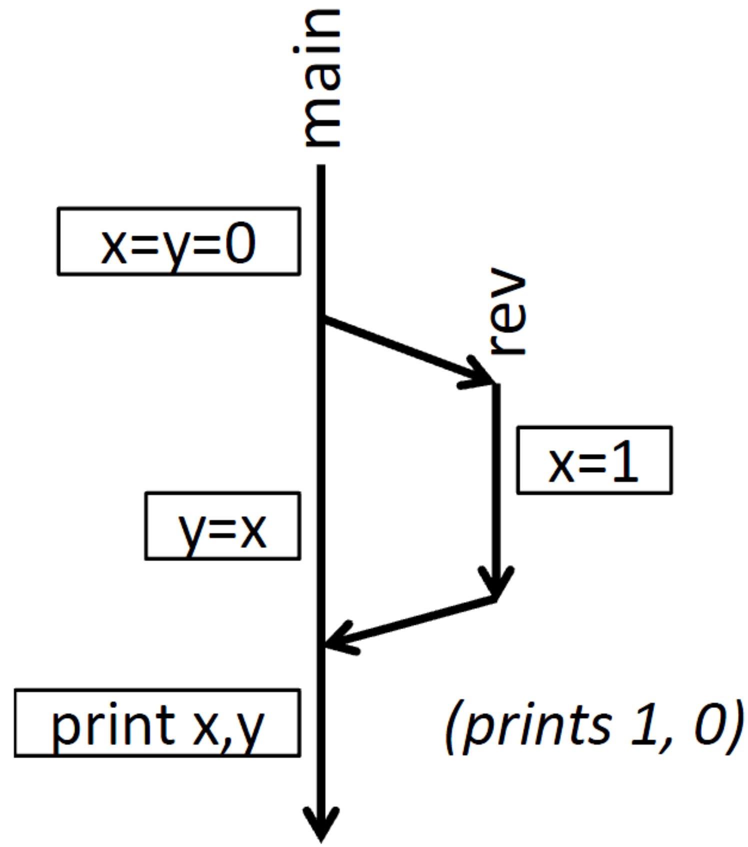
- **Deterministic conflict resolution**
  - After joining tasks (**revisions**) write-conflicts are solved deterministically.
  - Solving conflicts depends on declared **isolation types**.

Revisions

- **Revisions represents the basic unit on concurrency.**
- **Each revision can fork and join other revisions.**
- Applications starts in the main revision.
- **Two operations:**
  - fork – starts(forks) new revision
  - rjoin – joins two revisions (called **joiner** and **joinee**)
- Program execution can be easily presented on the **revision diagrams**.

Revisions (example)

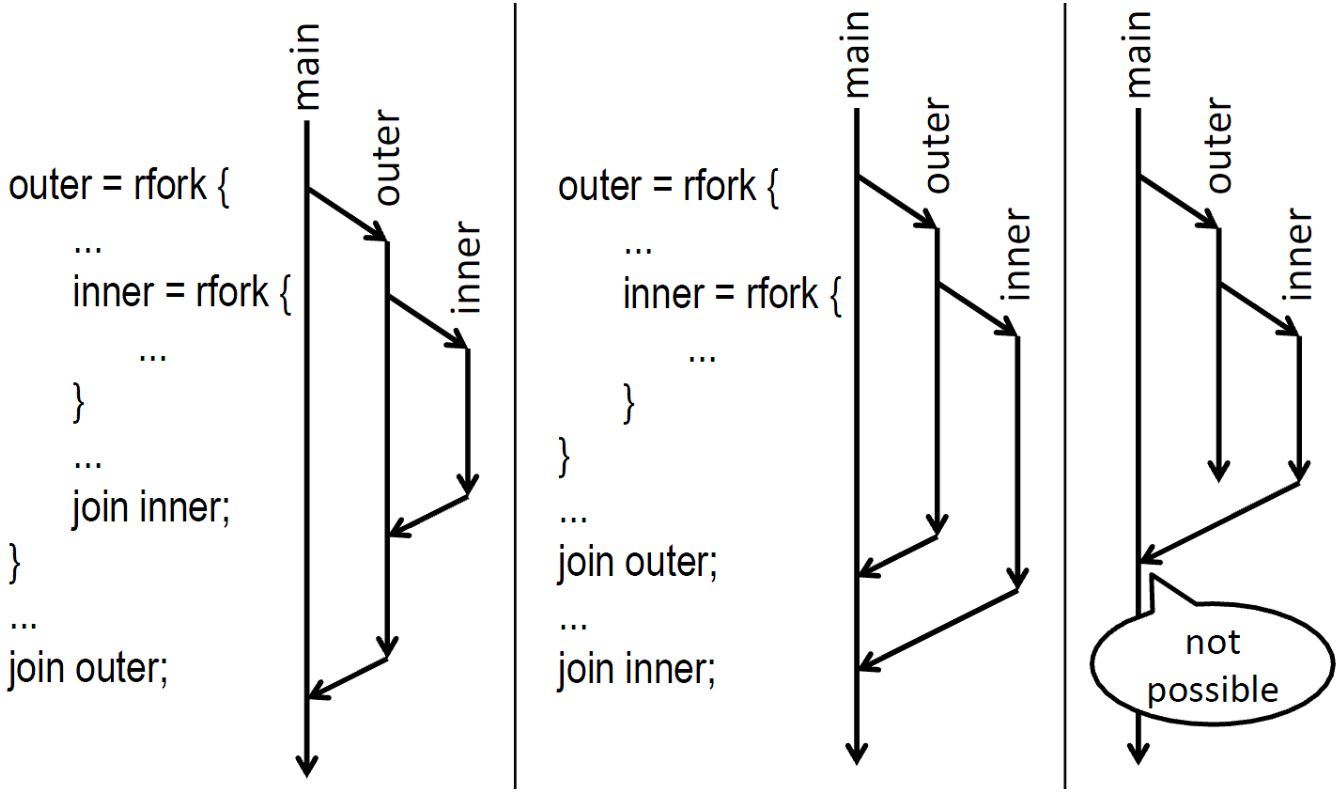
```
versioned⟨int⟩ x;
versioned⟨int⟩ y;
x = 0
y = 0;
revision r = rfork {
                    x = 1;
                  }
y = x;
rjoin r;
print x, y;
```



Nested revisions

- **Simplifies modular reasoning about program execution.**
- **Typically revisions fork its own inner revision and then join it.**
  - Classic nesting of tasks.
- **However, it is possible that inner (more nested) revision „survives" outer revision.**

Nested revisions (example)



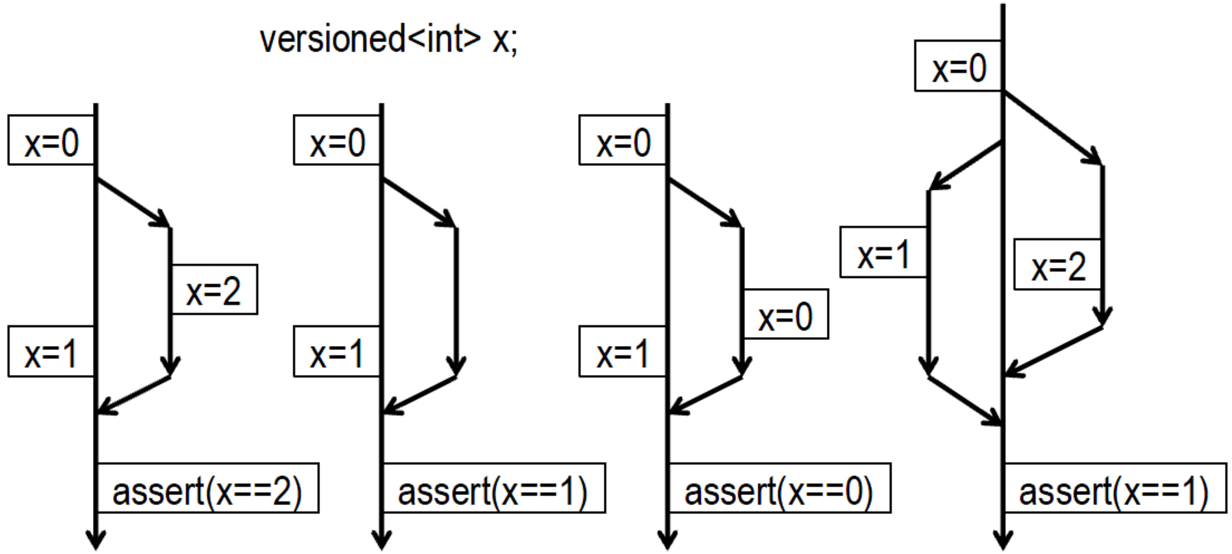
- Third situation is **impossible** because main revision doesn't know handle to the inner revision. **It gets that handle after joining outer revision**.

Isolation types

- When joining revisions, we wish to **merge the copies of the shared data** back together.
- Merging process is defined by **isolation types**.
- Programmer must choose **the right isolation type** for every single shared object.
- Isolation types fall into two categories:
  - **versioned <T>**
  - **cumulative <T, f>**

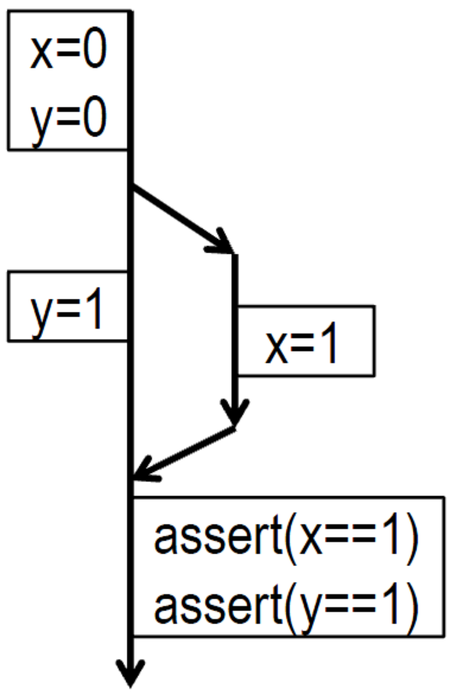
versioned <T>

- Changes the current value to the revision that is performing a join to the current value of the revision that is being joined.
- If there were no modifications in the **joinee** revision then do nothing.
- It is a good choice for tasks that have **relative priorities**.

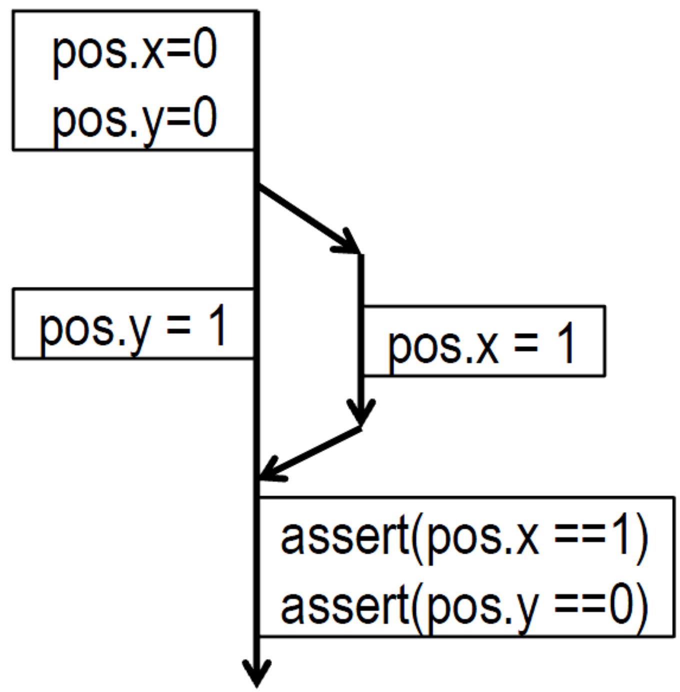


versioned <T>

```
versioned<int> x;
versioned<int> y;
```




```
typedef struct { int x,y } coord;
versioned<coord> pos;
```



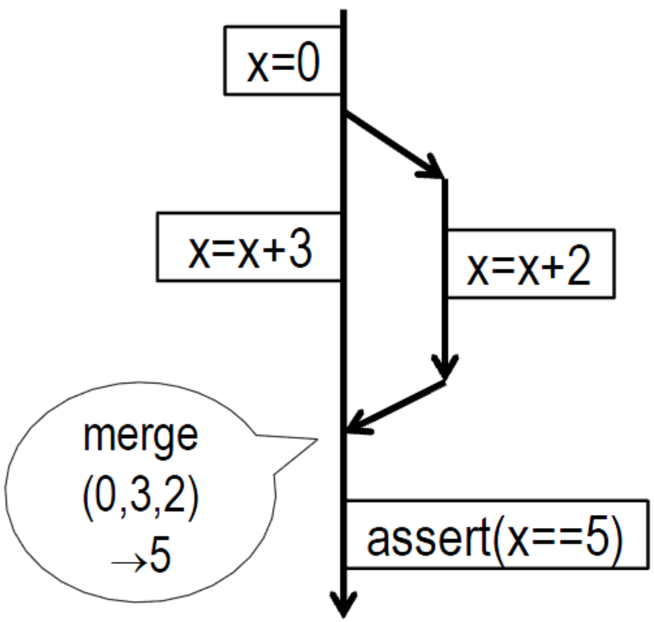
cumulative <T, f>

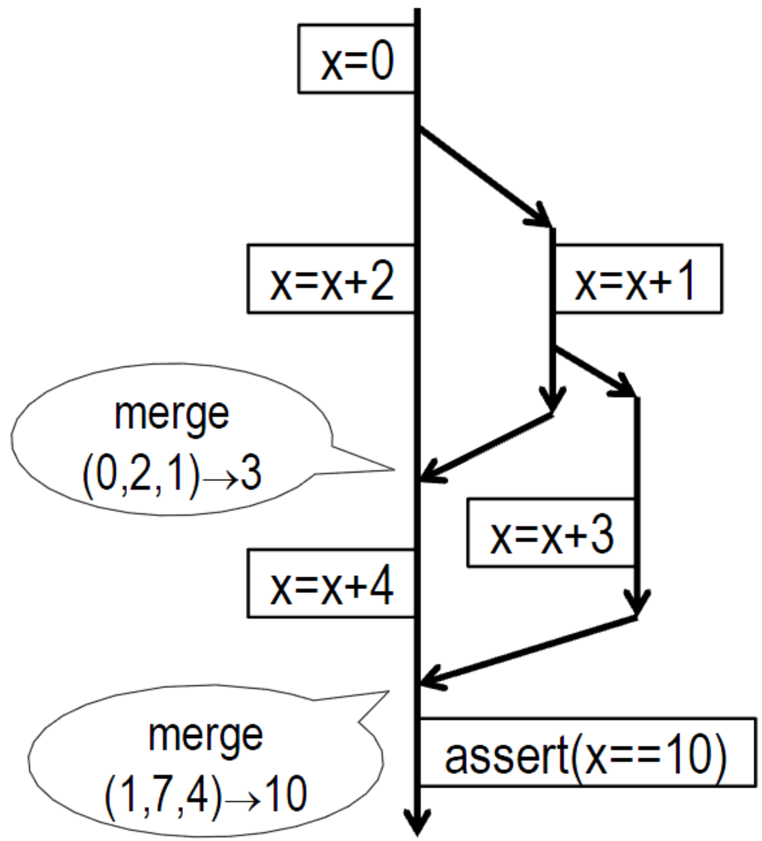
- Effect of the merge is determined by a general **merge function.**

- **Merge function** takes three arguments:
  - **original value**
  - **current value of the joiner revision**
  - **current value of the joinee revision**

cumulative <T, f> example

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





Comparison with locking scheme

- **Traditional locking scheme**
  - Requires programmers to think about placement of critical sections
  - Complicates code readability and maintenance
  - Reduces concurrency by pausing tasks
    - Eg. animating and rendering game objects in separate tasks
- **Presented solution**
  - Isolation of the read-only tasks and single writer task
    - so-called *double-buffering*
  - Might be not the most space-efficient solution

Comparison with transactions

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

Transactions
Two possible ways of execution

Revision
Single (deterministic) execution

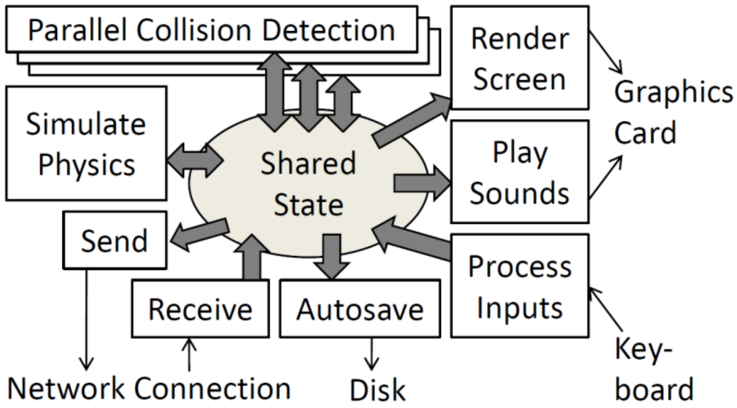
Case Study

SpaceWars3D

Game designed to teach DirectX programming in C#
12 000 lines

Parallelizing the game loop

```
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++;
}
```



RenderFrameToScreen()
It can execute in parallel with other tasks.
Reads objects position.

UpdateWorld()
Reads and modify objects position.

CollisionCheck(i)
It can be parallelized.
Reads and modify objects position.

Objects position is also read by
SendNetworkUpdates()
HandleQueuePackets()

Parallelizing the game loop (2)

```
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;
    }
 ...
```

Parallelizing the game loop (3)

- **Declared isolation types:**
  - VersionedValue<T> - mainly simple types
  - VersionedObject<T> - asteroids, positions, particle effects
  - CumulativeValue<T> - message buffer
  - CumulativeList<T> - lists of asteroids

- **CollisionsCheck** could be executed in parallel. Optimizing

- **RenderFrameToScreen** cannot be parallelized, but it can be executed in parallel with other tasks

- Updates from the network have higher priority than updates done by **UpdateWorld** or **CollisionsCheck**

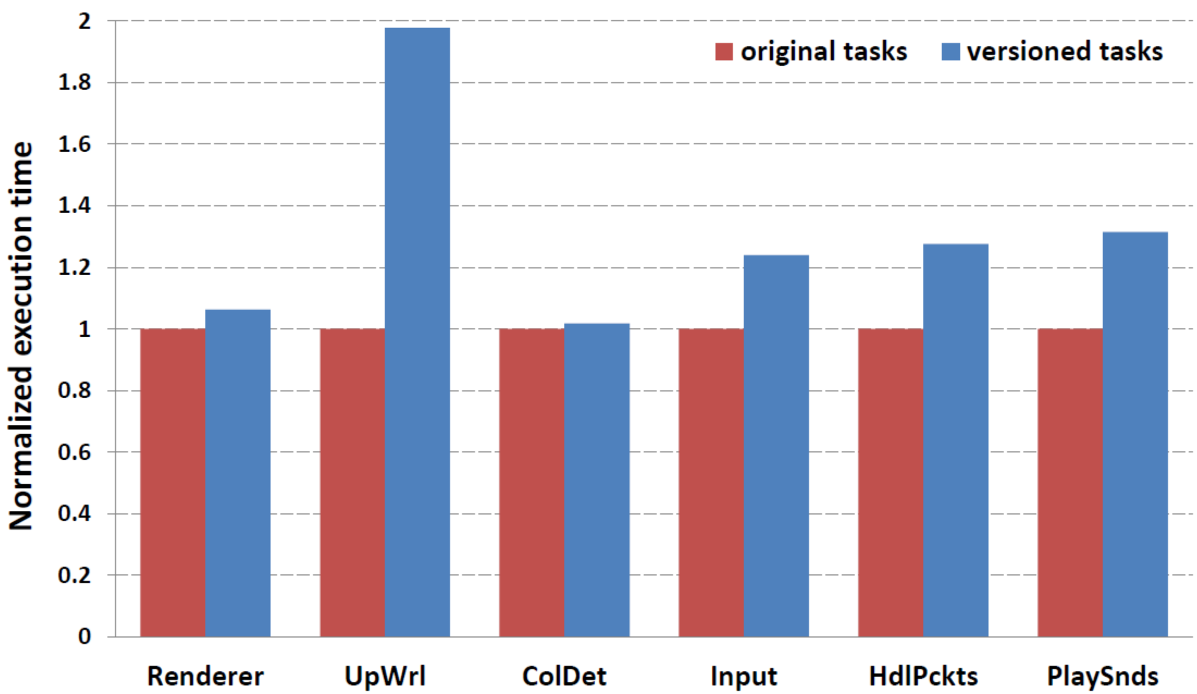
- **Deterministic Record and Replay**
  - Used in performance evaluation

Evaluation

Testing environment:
Intel Xeon W3520 (quad-core) 2.66Ghz, 6GB of DDR3,
NVIDIA Quadro FX580 512MB, Windows 7 Enterprise 64-bit

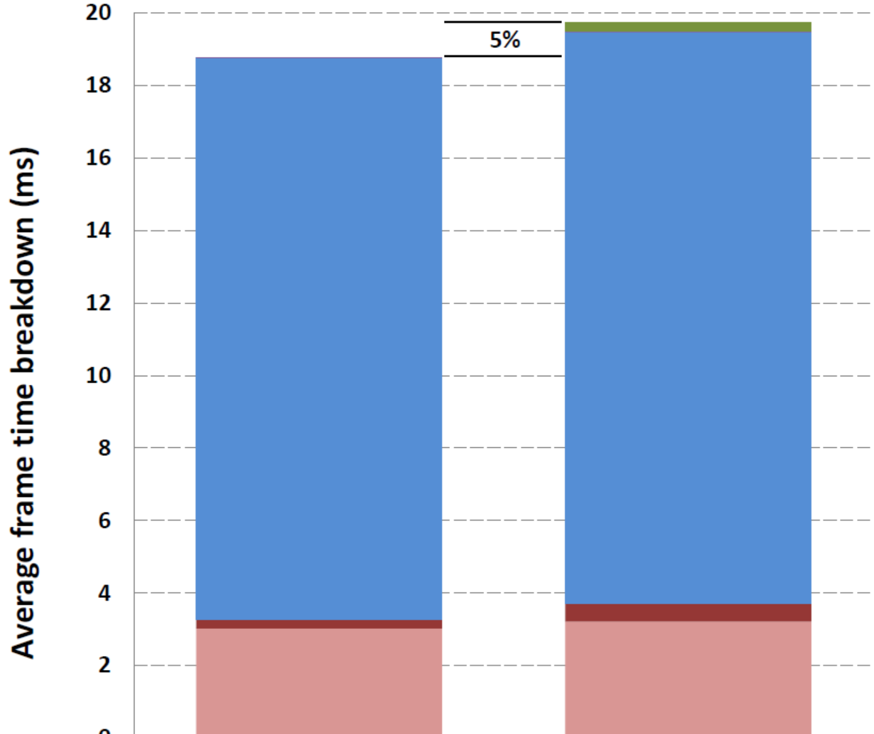
Revisions overhead

- **Revision overhead causes slowdown of 5%**
  - Execution in single thread
  - Collision detection task takes about **82%** of the frame time.



R/W	848/0	68/835	30986/0	0/2	2/18	3/3

Revisions overhead (2)



	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

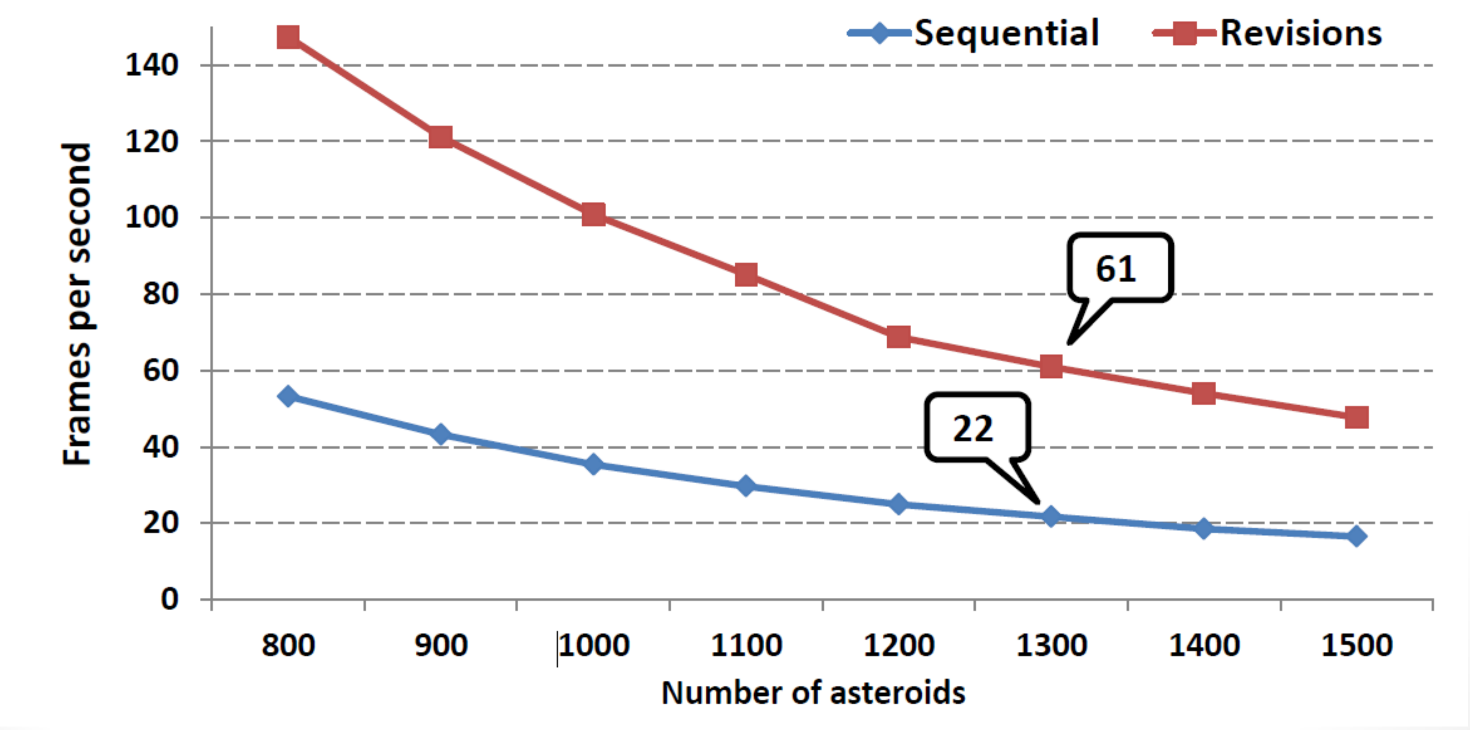
Memory consumption

- If revision writes to a versioned object then revisioning subsystem needs to clone object to enforce isolation.

# 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

Parallel performance

- Average speedup of **2.6x** on quad-core processor
- Up to **3.03x** speedup for 1500 asteroids
- Render task takes **95.5%** of the frame time (*collisions check speedup*)



Implementation

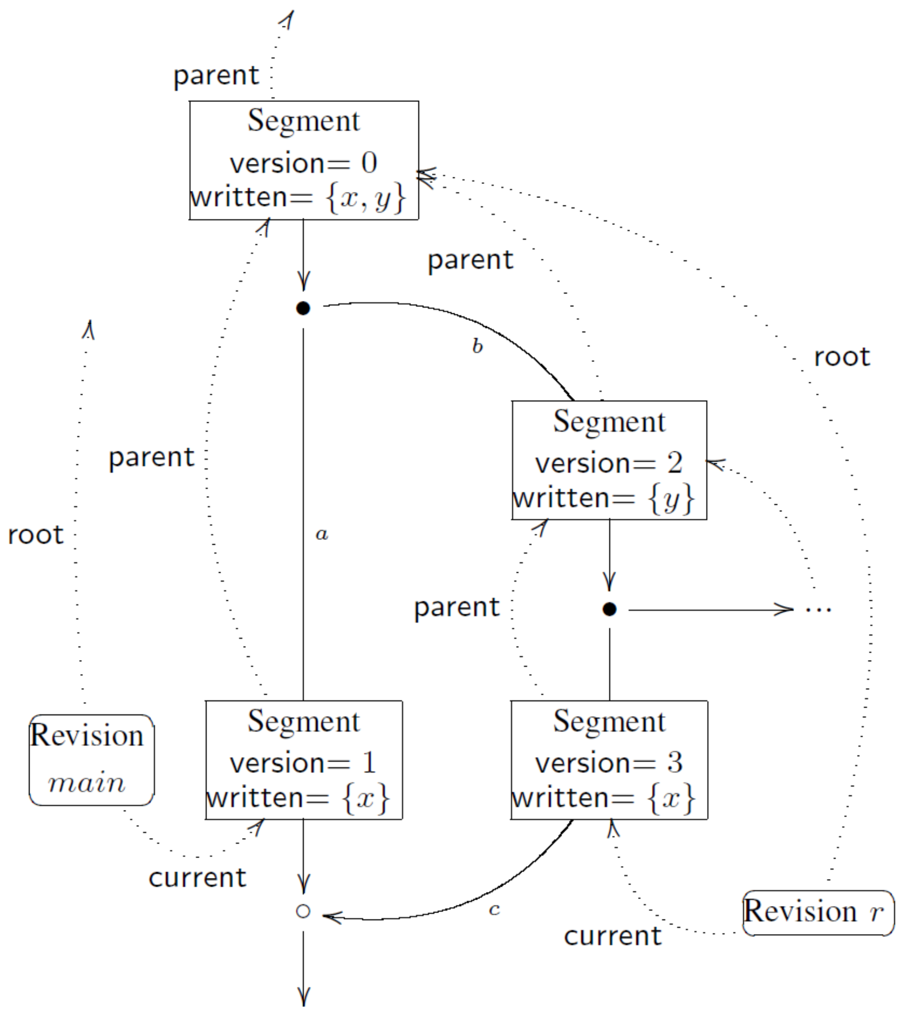
- **C# library**
- **Revisions**
  - Stores current segment that points to the segment created for that revision after last fork
  - Root segment is the segment right above fork
- **Segments**
  - Tree structure
  - Stores versioned data
  - Ancestors of the current segment can be removed after join if there is no other revisions that uses them.

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

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

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

Revisions and segments (example)



Forks and joins

- Fork
  - Creates new current segments for both revisions
  - Assigns new revision with the new thread
- Join
  - Creates one new segment
  - Merges versioned objects from both revisions
  - Releases unused ancestors of the current segment of the joined revision

Summary

- Programming model based on revisions and isolation types.
- Efficient mechanism for executing different tasks within interactive applications.
- Future work:
  - Optimizations
  - Executing tasks on GPU
  - Applications that run on multi-core processors without full shared-memory guarantees
  - Application that run in the cloud

Thank you.

Questions?