Concurrent Revisions

A novel deterministic concurrency model

Daan Leijen

Microsoft Research

Side effects and TPL

- TPL is great for introducing parallelism where the tasks are independent: i.e. their side effects are disjoint.
- For example, for matrix multiplication:

```
int size;
int[,] result,m1,m2;
Parallel.For( 0, size, delegate(int i) {
  for (int j = 0; j < size; j++) {
    result[i, j] = 0;
    for (int k = 0; k < size; k++) {
      result[i, j] += m1[i, k] * m2[k, j];
    }
  }
});
```

Side effects and TPL

- But it is easy to make mistakes and introduce data races:

```
int x,y;
int sum = 0;
Task t = fork {
   sum += x;
}
sum += y;
join t;
```

- We would like to guarantee that `sum==x+y` but without fences/locks we could get different answers

Puzzler

- Suppose we have a sequentially consistent memory model. What are the values that we can observe for x and y?

```
int x = 0;
int y = 0;
Task t = fork {
   if (x==0) y++;
}
if (y==0) x++;
join t;
```

Answer

- Depending on the schedule:
  - first do task (1) then (2) : **(x==0 && y==1)**
  - or the other way around : **(x==1 && y==0)**
  - or do the test “if (x==0) …” in task (1) first, switch to the task (2), and increment x, then switch back to (1) again and increment y : **(x==1 && y==1)**

```
int x = 0;
int y = 0;
Task t = fork {
   if (x==0) y++;  (1)
}
if (y==0) x++;     (2)
join t;
```

Do locks or STM help?

- With locks or software transactional memory we can denote statements that should be executed atomically.
- What values can we get now for x and y?

```
x = 0;
y = 0;
task t = fork {
  atomic { if (x==0) y++; }
}
atomic { if (y==0) x++; }
join t;
```

Answer

- Depending on the schedule:
  - first do task (1) then (2) : **(x==0 && y==1)**
  - or the other way around : **(x==1 && y==0)**
  - but no other schedule is possible
- Still, there is non-determinism introduced by the scheduler.

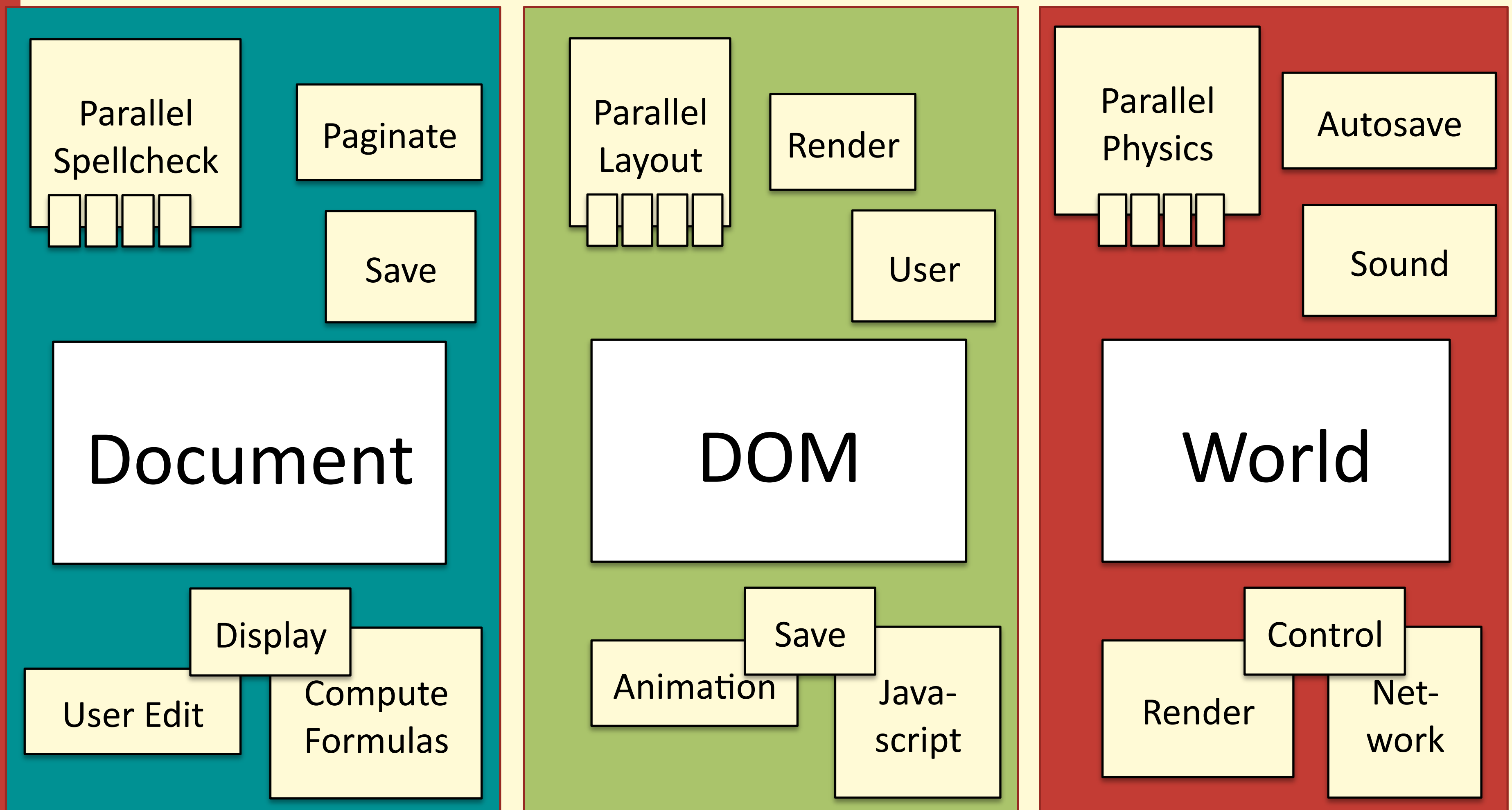
```
x = 0;
y = 0;
task t = fork {
  atomic { if (x==0) y++; }
}
atomic { if (y==0) x++; }
join t;
```

Huge problem in practice

- For many applications we have
  - large shared data structures
  - the updates in tasks are dynamic and can conflict
  - complex invariants that could span over lots of code

3 Examples of this Pattern:

Office Browser Game




THE EXPERT'S VOICE
Beginning
.NET Game Programming in C#
David Weller, Alexandre Santos Lobão, and Ellen Hatton
Apress
VECTOR TO OPPONENT
Range: 1598
Parallel Physics
Autosave
Sound
World
Control
Render
Net-work

The Game:
Can you parallelize this 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++;
}
```

Conflicts on object Coordinates:

Reads and writes all positions

Reads all, **Writes** some positions

Writes some positions

Reads all positions

Reads all positions

Parallelization Challenges

Example 1: read-write conflict

- Render task reads position of all game objects
- Physics task updates position of all game objects

  => Render task needs to see consistent snapshot

Example 2: write-write conflict

- Physics task updates position of all game objects
- Network task updates position of some objects

  => Network has priority over physics updates

Conventional Concurrency Control

Conflicting tasks can not efficiently execute in parallel.

- pessimistic concurrency control (locks)
  - use locks to avoid parallelism where there are (real or potential) conflicts
- optimistic concurrency control (STM)
  - speculate on absence of true conflicts rollback if there are real conflicts

either way: true conflicts kill parallel performance.

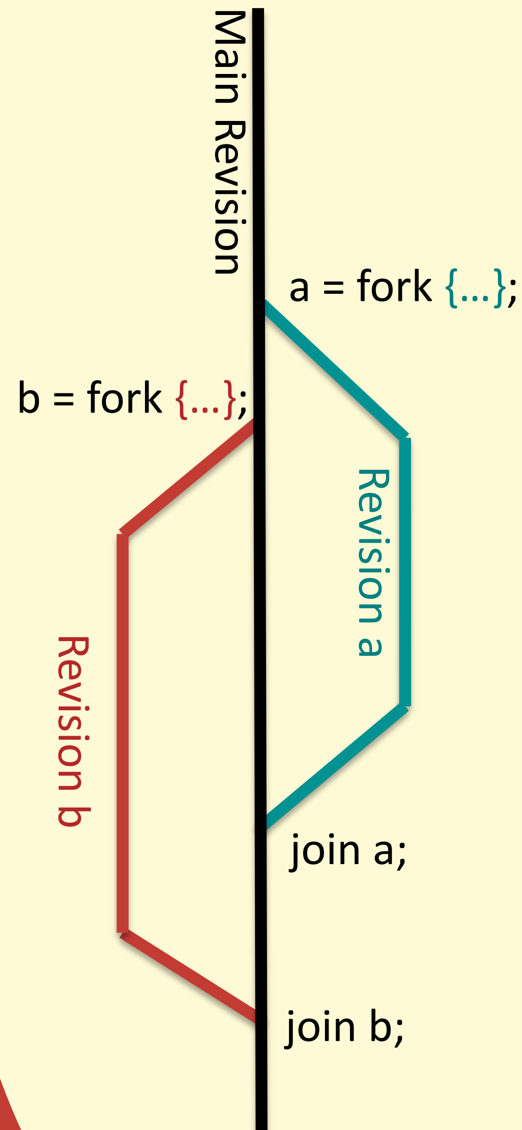
Concurrent revisions

- A *deterministic* and *concurrent* programming model
- Apply 'source control' on program state

The model:

- Programmer forks & joins revisions (eg. tasks)
- Each revision gets its own logical copy of the shared state and works fully isolated
- On the join, all changes in a child are merged into the main revision.

The Programming Model



- Revision shared memory
  - Program explicitly forks/joins revisions
  - Revisions are isolated: operate on their own copy of the state
  - Revisions can be scheduled for parallel execution
  - At time of join, conflicts are resolved deterministically (always, **never roll back**)
- Deterministic Semantics ✓

How does it look?

fork revision:
forks off a private copy of the shared state

Traditional Task

```
int x = 0;
Task t = fork {
   x = 1;
}
assert(x==0 || x==1);
join t;
assert(x==1);
```

Concurrent Revisions

```
Versioned<int> x = 0;
Revision r = rfork {
   x = 1;
}
assert(x==0);
join r;
assert(x==1);
```

isolation:
Concurrent modifications are not seen by others

join revision:
waits for the revision to terminate and writes back changes into the main revision

Sequential Consistency

```
int x = 0;
int y = 0;
Task t = fork {
   if (x==0) y++;
}
if (y==0) x++;
join t;

assert(   (x==0 && y==1)
       || (x==1 && y==0)
       || (x==1 && y==1));
```

Transactional Memory

```
int x = 0;
int y = 0;
Task t = fork {
  atomic { if (x==0) y++; }
}
atomic { if (y==0) x++; }
join t;

assert(   (x==0 && y==1)
       || (x==1 && y==0));
```

Puzzler: What are x and y for concurrent revisions?

```
Versioned<int> x = 0;
Versioned<int> y = 0;
Revision r = rfork {
   if (x==0) y++;
}
if (y==0) x++;
join r;
```

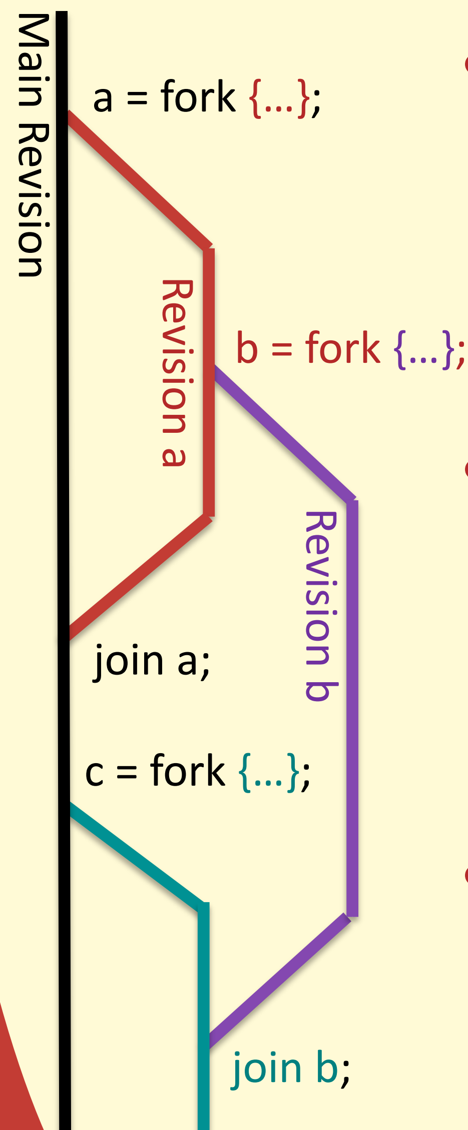
Answer:

- In concurrent revision, you can always reason within one revision as if you are sequential (since there is full isolation). Therefore, both revisions (tasks) will increment: **x==1** and **y==1**
- The answer is always independent of any particular schedule*

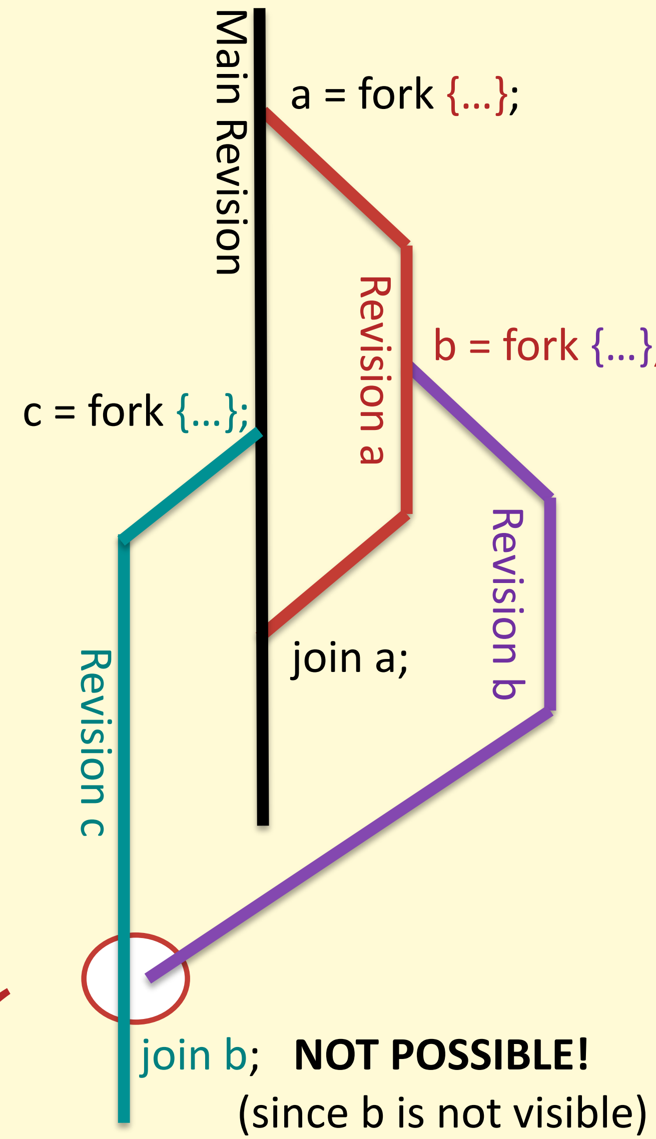
```
Versioned<int> x = 0;
Versioned<int> y = 0;
Revision r = rfork {
   if (x==0) y++;
}
if (y==0) x++;
join r;
```

*) only if deterministic actions are executed

Revision Diagrams ≠ SP-graphs, DAGs

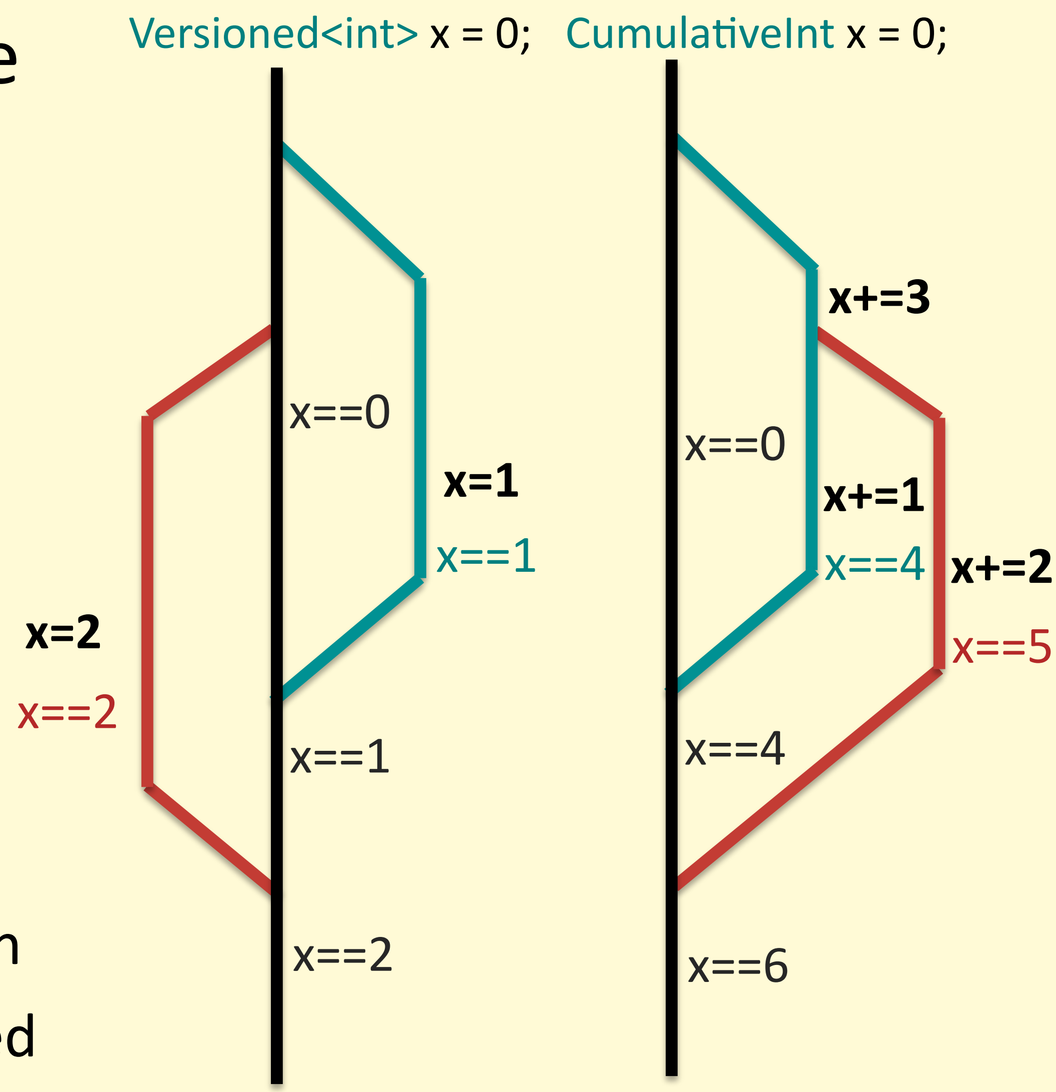


- Not limited to "fork-join" parallelism
- Not as general as arbitrary task graphs
- Revision Diagrams are semi-lattices ✓



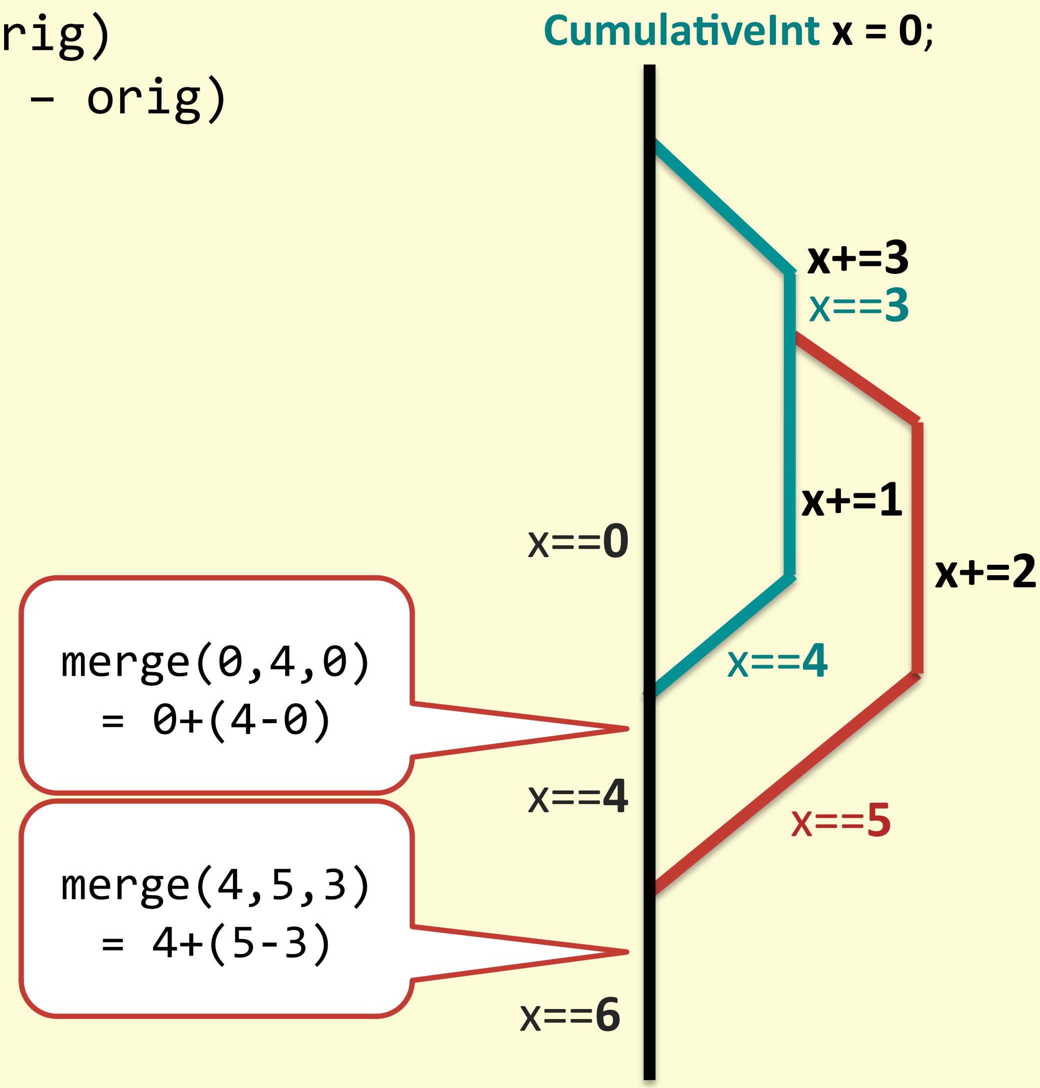
Conflict Resolution: By Type

- Declare Isolation Type per var/field/struct.
- Two categories:
  - Versioned Types
    - Last join wins
    - Good for controlling write order
  - Cumulative Types
    - Combines effects
    - Good for accumulation
    - Built-in or User-Defined



Custom merge functions

```
merge(main,join,orig)
   = main + (join – orig)
```



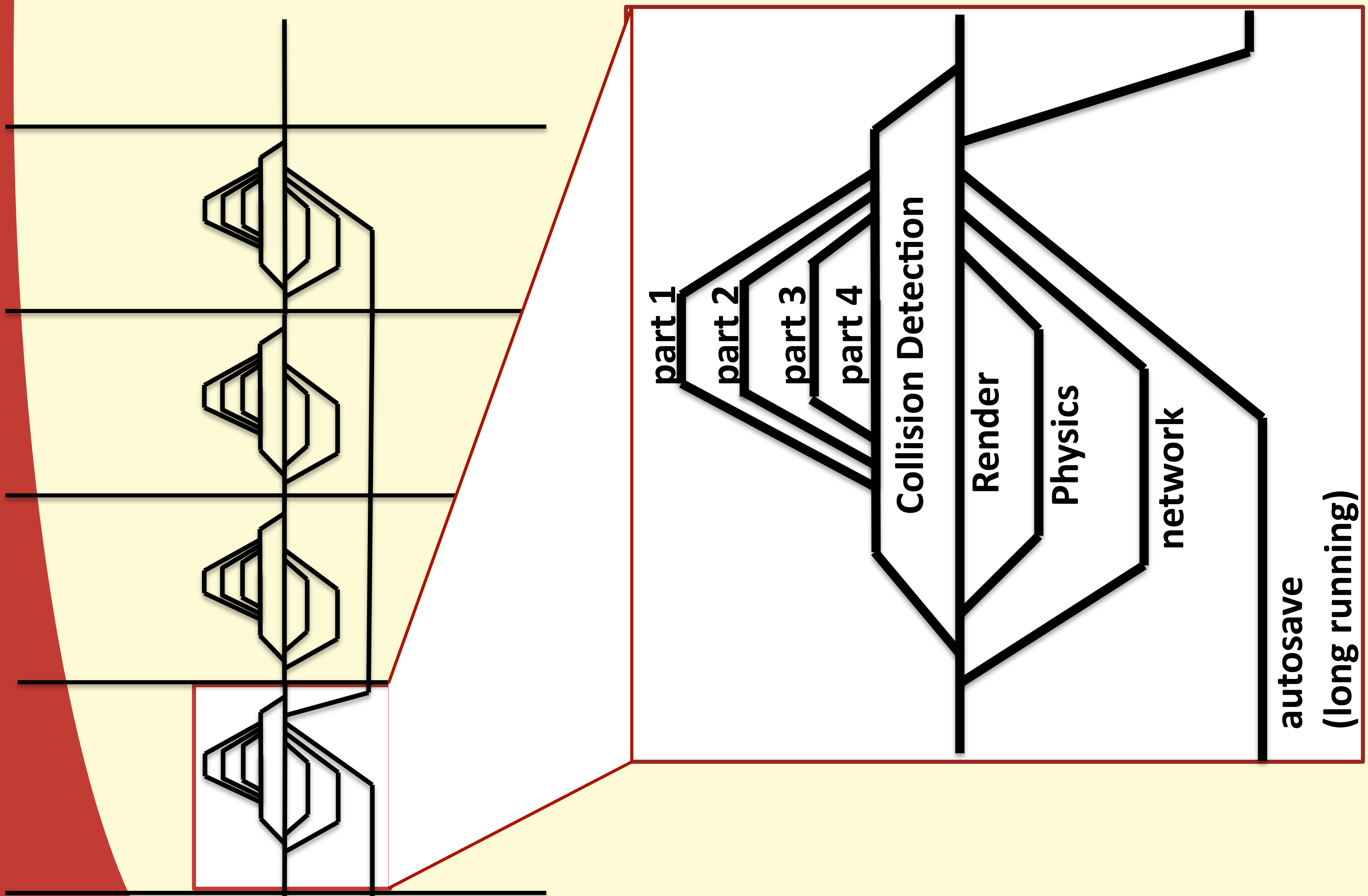
Back to the Game: Can we now parallelize this loop?

```
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++;
}
```

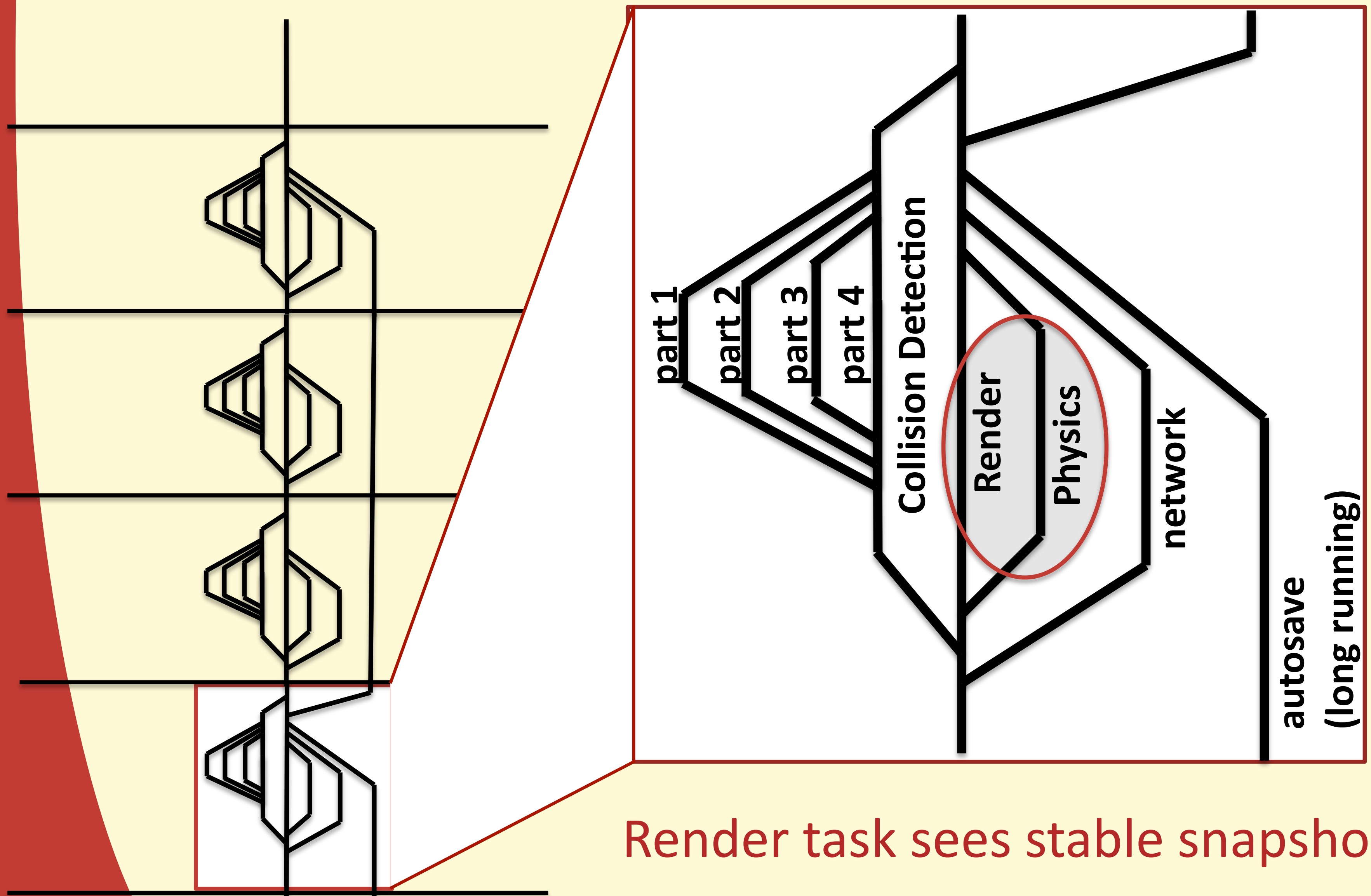
Conflicts on object Coordinates:

- physics.UpdateWorld(): **Reads and writes** all positions
- physics.CollisionCheck(i): **Reads** all, **Writes** some positions
- network.HandleQueuedPackets(): **Writes** some positions
- SaveGame(): **Reads** all positions
- screen.RenderFrameToScreen(): **Reads** all positions

Revision Diagram of Parallelized Game Loop

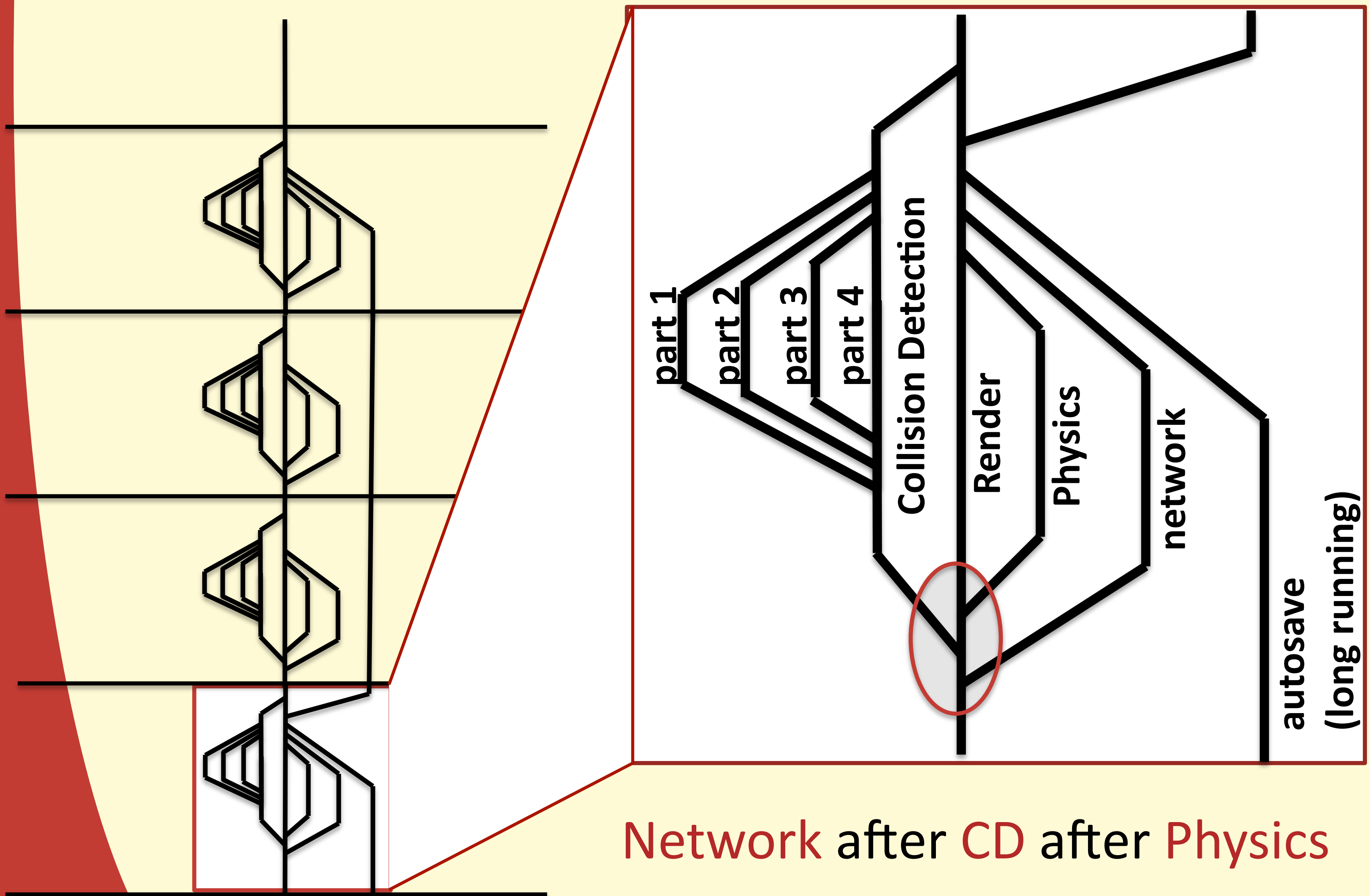


Isolation Eliminates Read-Write Conflicts



Render task sees stable snapshot

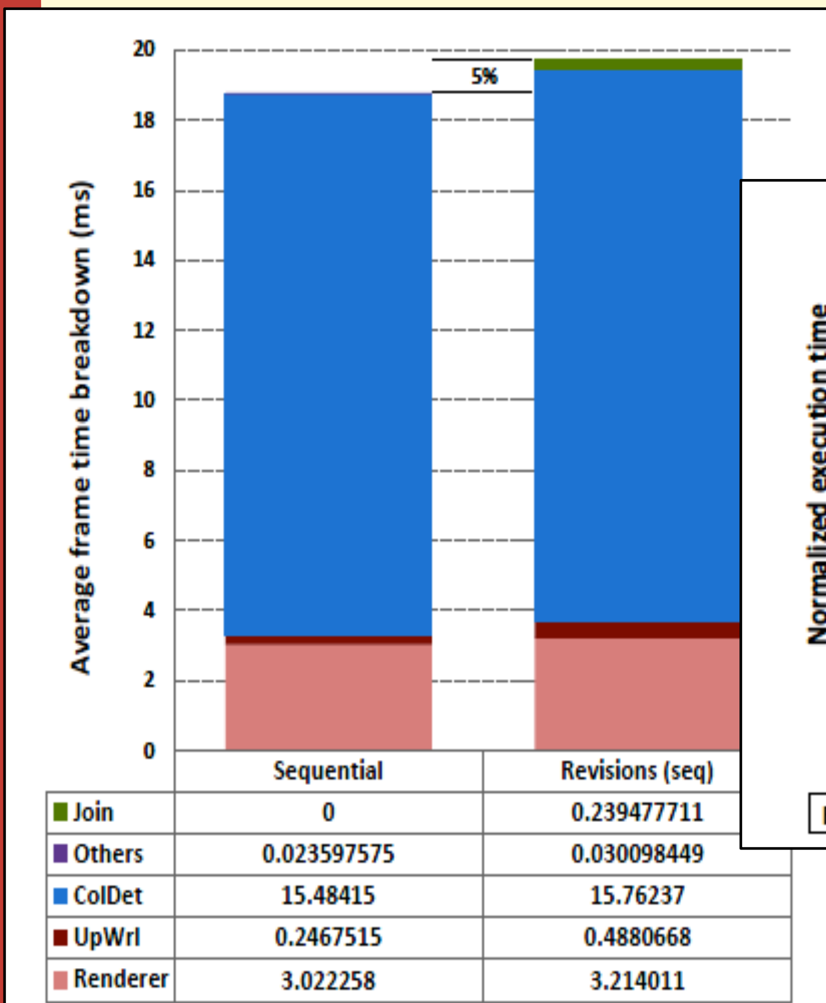
Joins Order Write-Write Conflicts

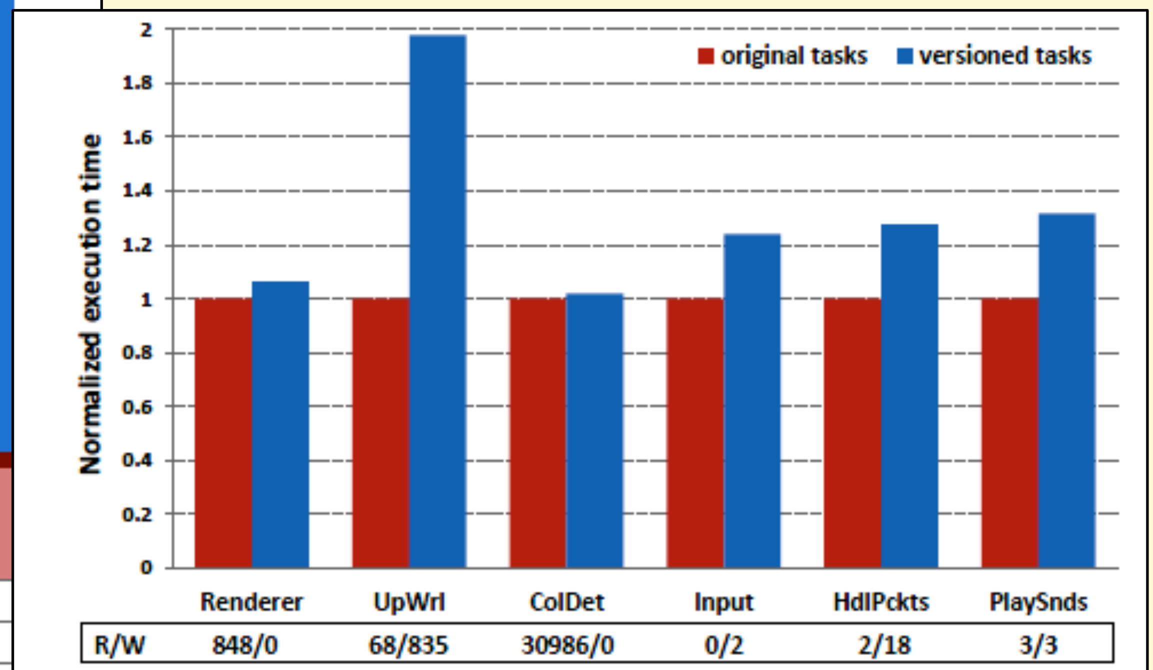


Network after CD after Physics

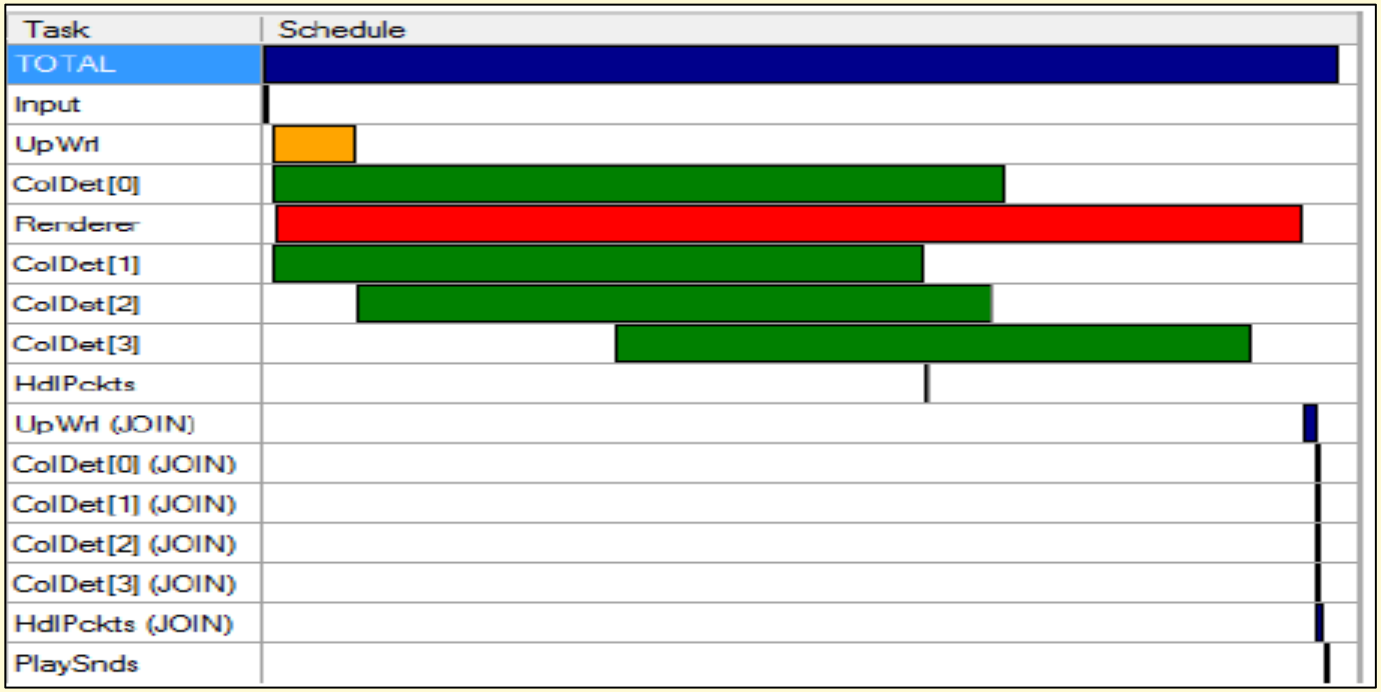
Overhead:

How much does all the copying and the indirection cost?





Results



- Autosave now perfectly unnoticeable in background
- Overall Speed-Up:
  3.03x on four-core
  (almost completely limited by graphics card)

A Software engineering perspective

- Transactional memory:
  - **Code** centric: put “atomic” in the code
  - Granularity:
    - too broad: too many conflicts and no parallel speedup
    - too small: potential races and incorrect code
- Concurrent revisions:
  - **Data** centric: put annotations on the data
  - Granularity: group data that have mutual constraints together, i.e. if $(x + y > 0)$ should hold, then x and y should be versioned together.

Current Implementation: C# library

- For each versioned object, maintain multiple copies
  - Map revision ids to versions
  - `mostly’ **lock-free** array

Revision	Value
1	0
40	2
45	7

- New copies are allocated lazily
  - Don’t copy on fork… **copy on first write** after fork
- Old copies are released on join
  - **No space leak**

Demo

```
class Sample
{
  [Versioned]
  int i = 0;

  public void Run()
  {
    var t1 = CurrentRevision.Fork(() => {
      i += 1;
    });
    var t2 = CurrentRevision.Fork(() => {
      i += 2;
    });
    i+=3;

    CurrentRevision.Join(t1);
    CurrentRevision.Join(t2);
    Console.WriteLine("i = " + i);
  }
}
```

Demo

```
class Sample {
  [Versioned,MergeWith(“merge”)]
  int i = 0;

  public int merge(int main, int join, int org){
    return main+(join-org);
  }

  public void Run() {
    var t1 = CurrentRevision.Fork(() => {
      i += 1;
    });
    var t2 = CurrentRevision.Fork(() => {
      i += 2;
    });
    i+=3;
    CurrentRevision.Join(t1);
    CurrentRevision.Join(t2);
    Console.WriteLine("i = " + i);
  }
}
```

Demo: Sandbox

```
class Sandbox
{
  [Versioned]
  int i = 0;

  public void Run()
  {
    var r = CurrentRevision.Branch("FlakyCode");
    try {
      r.Run(() =>
      {
        i = 1;
        throw new Exception("Oops");
      });
      CurrentRevision.Merge(r);
    }
    catch {
      CurrentRevision.Abandon(r);
    }
    Console.WriteLine("\n i = " + i);
  }
}
```

Fork a revision without forking an associated task/thread

Run code in a certain revision

Merge changes in a revision into the main one

Abandon a revision and don't merge its changes.

Formal Semantics

- Similar to the AME calculus by Abadi *et al.*
- Proof of determinism
- Formal correspondence to the Revision Diagrams
- Proof of the semi-lattice property
- Can be shown to generalize ‘snapshot isolation’

Syntactic Symbols

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

State

By construction, there is no 'global' state: just local state for each revision

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

State is simply a (partial) function from a location to a value

Execution Contexts

$$
\begin{aligned}
\mathcal{E} = \; & \Box \\
\mid \; & \mathcal{E}\, e \mid v\, \mathcal{E} \mid (\mathcal{E}\, ?\, e : e) \\
\mid \; & \mathsf{ref}\, \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid l := \mathcal{E} \\
\mid \; & \mathsf{rjoin}\, \mathcal{E}
\end{aligned}
$$

Operational Semantics

For some revision *r*, with snapshot [illegible] and local modifications [illegible] and an expression context with hole ([illegible]*x*.*e*) *v*

the state is a composition of the root snapshot [illegible] and local modifications [illegible]

$$
\begin{array}{ll}
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\lambda x.e)\ v] \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[[v/x]e] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{true}\ ?\ e_1 : e_2)] \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_1] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{false}\ ?\ e_1 : e_2)] \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_2] \rangle] \\
\\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[l] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[!l] \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[(\sigma{::}\tau)(l)] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[l := v] \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[\mathsf{unit}] \rangle] \\
\\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rfork}\ e] \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[r'] \rangle][r' \mapsto \langle \sigma{::}\tau, \epsilon, e \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \langle \sigma', \tau', v \rangle) & \rightarrow_r s[r \mapsto \langle \sigma, \tau{::}\tau', \mathcal{E}[\mathsf{unit}] \rangle][r' \mapsto \bot] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \bot) & \rightarrow_r \epsilon
\end{array}
$$

On a **join**, the writes of the joinee *r'* take priority over the writes of the current revision: [illegible] :: [illegible]'

On a **fork**, the snapshot of the new revision *r'* is the current state: [illegible] :: [illegible]

Questions?

- daan@microsoft.com
- sburckha@microsoft.com

- Download available soon on CodePlex
- Play right now on the web: http://rise4fun.com/Revisions
- Bing "Concurrent Revisions" http://research.microsoft.com/en-us/projects/revisions/