Software Transactional Memory "for real"

Introduction

- This talk is about programming multi- or many-core machines

About myself

- Armin Rigo

About myself

- Armin Rigo
- "Language implementation guy"

About myself

- Armin Rigo
- "Language implementation guy"

About myself

- Armin Rigo
- "Language implementation guy"
- PyPy project

About myself

- Armin Rigo
- "Language implementation guy"
- PyPy project
  - Python in Python

About myself

- Armin Rigo
- "Language implementation guy"
- PyPy project
  - Python in Python
  - includes a Just-in-Time Compiler "Generator" for Python and any other dynamic language

Motivation

- A single-core program is getting exponentially slower than a multi-core one

Motivation

- A single-core program is getting exponentially slower than a multi-core one

Motivation

- A single-core program is getting exponentially slower than a multi-core one
- Using several processes exchanging data

Motivation

- A single-core program is getting exponentially slower than a multi-core one
- Using several processes exchanging data
  - works fine in some cases

Motivation

- A single-core program is getting exponentially slower than a multi-core one
- Using several processes exchanging data
  - works fine in some cases
  - but becomes a large mess in others

Motivation

- A single-core program is getting exponentially slower than a multi-core one
- Using several processes exchanging data
  - works fine in some cases
  - but becomes a large mess in others

Motivation

- A single-core program is getting exponentially slower than a multi-core one
- Using several processes exchanging data
  - works fine in some cases
  - but becomes a large mess in others
- Using several threads

Motivation

- A single-core program is getting exponentially slower than a multi-core one
- Using several processes exchanging data
  - works fine in some cases
  - but becomes a large mess in others
- Using several threads
  - this talk!

Common solution

- Organize your program in multiple threads

Common solution

- Organize your program in multiple threads
- Add synchronization when accessing shared, non-read-only data

Synchronization with locks

- Carefully place locks around every access to shared data

Synchronization with locks

- Carefully place locks around every access to shared data

Synchronization with locks

- Carefully place locks around every access to shared data
- How do you know if you missed a place?

Synchronization with locks

- Carefully place locks around every access to shared data
- How do you know if you missed a place?
  - hard to catch by writing tests

Synchronization with locks

- Carefully place locks around every access to shared data
- How do you know if you missed a place?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes

Synchronization with locks

- Carefully place locks around every access to shared data
- How do you know if you missed a place?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes

Synchronization with locks

- Carefully place locks around every access to shared data
- How do you know if you missed a place?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes
- Issues when scaling to a large program

Synchronization with locks

- Carefully place locks around every access to shared data
- How do you know if you missed a place?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes
- Issues when scaling to a large program
  - order of acquisition

Synchronization with locks

- Carefully place locks around every access to shared data
- How do you know if you missed a place?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes
- Issues when scaling to a large program
  - order of acquisition
  - deadlocks

Synchronization with TM

- TM = Transactional Memory

Synchronization with TM

- TM = Transactional Memory

Synchronization with TM

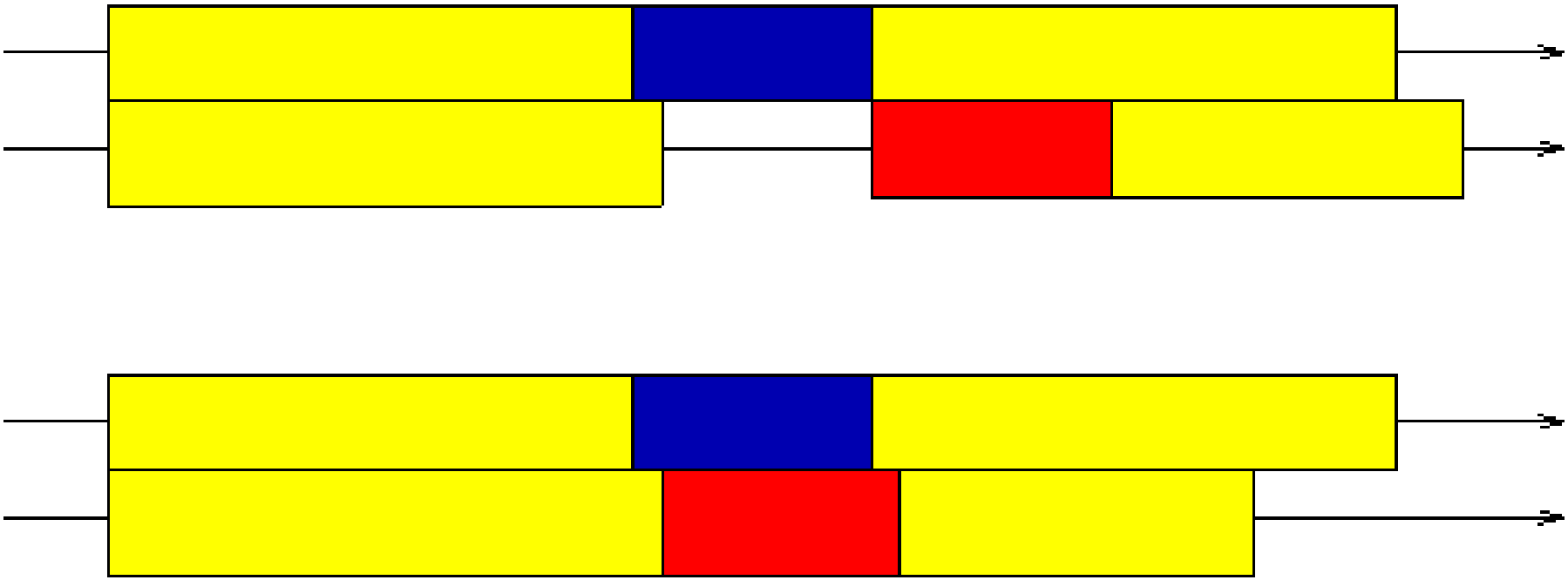
- TM = Transactional Memory

```
-----------------    ---------------------
Locks                Transactional Memory
-----------------    ---------------------
```

```
mylock.acquire();      atomic {
x = list1.pop();         x = list1.pop();
list2.append(x);         list2.append(x);
mylock.release();      }
```

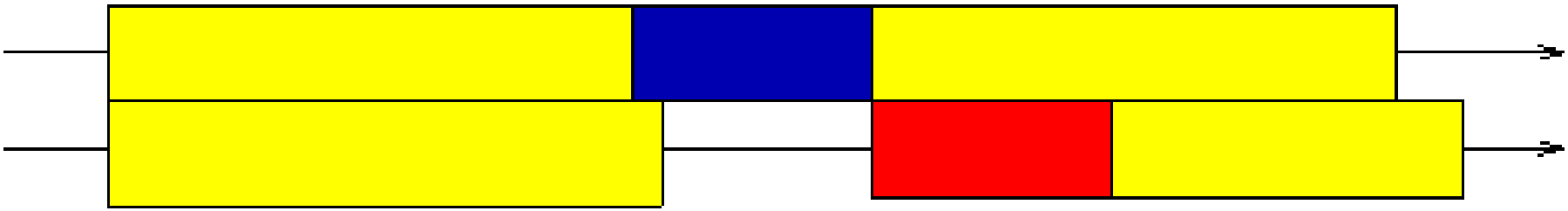
Locks versus TM

- Locks

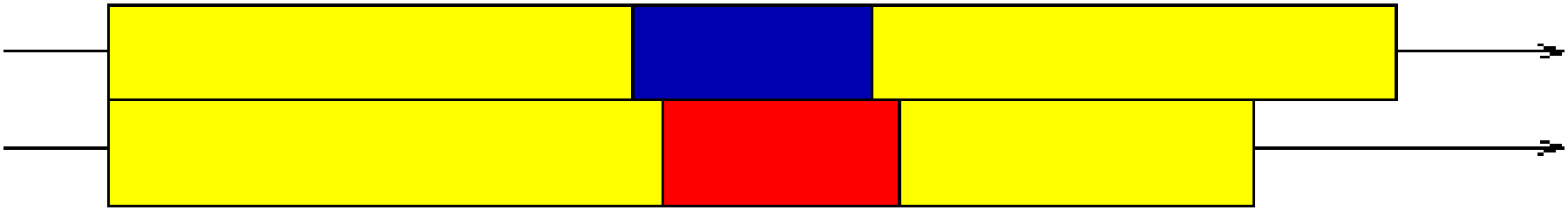


Locks versus TM

- Locks

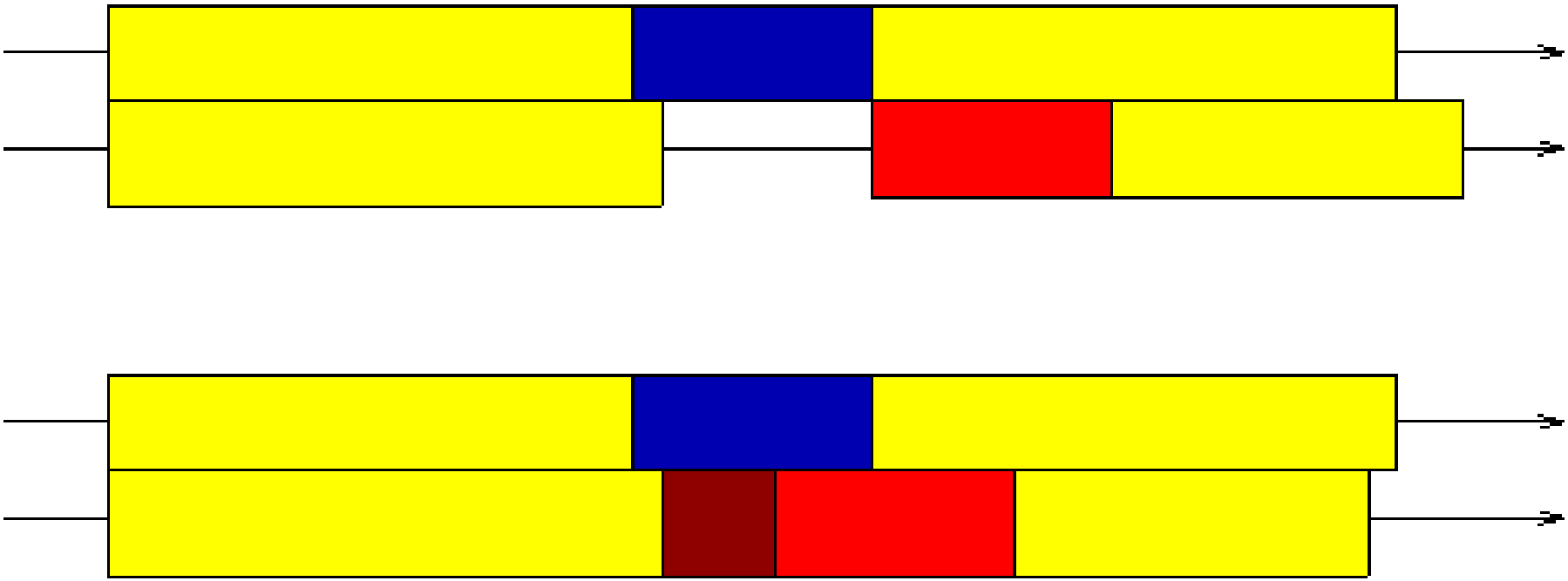


- TM



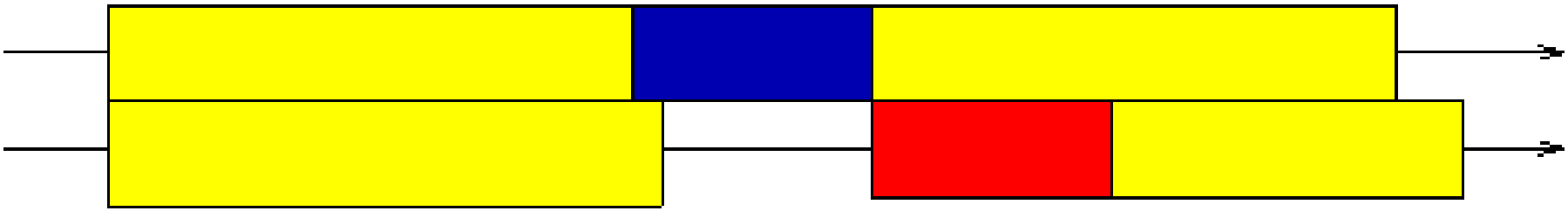
Locks versus TM

- Locks

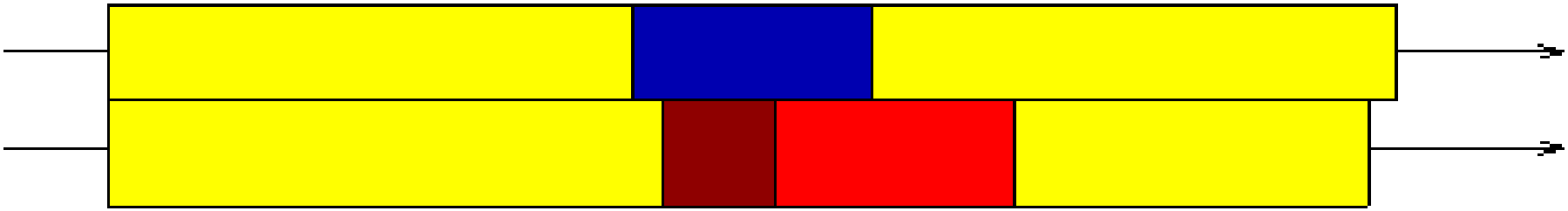


Locks versus TM

- Locks



- TM in case of conflict



Synchronization with TM

- "Optimistic" approach:

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses
  - detect actual conflicts

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses
  - detect actual conflicts
  - if conflict, restart the whole "transaction"

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses
  - detect actual conflicts
  - if conflict, restart the whole "transaction"

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses
  - detect actual conflicts
  - if conflict, restart the whole "transaction"

- Easier to use

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses
  - detect actual conflicts
  - if conflict, restart the whole "transaction"
- Easier to use
  - no need to name locks

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses
  - detect actual conflicts
  - if conflict, restart the whole "transaction"
- Easier to use
  - no need to name locks
  - no deadlocks

Synchronization with TM

- "Optimistic" approach:
  - no lock to protect shared data in memory
  - instead, track all memory accesses
  - detect actual conflicts
  - if conflict, restart the whole "transaction"
- Easier to use
  - no need to name locks
  - no deadlocks
  - "composability"

HTM versus STM

- HTM = Hardware Transactional Memory

HTM versus STM

- HTM = Hardware Transactional Memory
  - Intel Haswell CPU, 2013

HTM versus STM

- HTM = Hardware Transactional Memory
  - Intel Haswell CPU, 2013
  - and others

HTM versus STM

- HTM = Hardware Transactional Memory
  - Intel Haswell CPU, 2013
  - and others

HTM versus STM

- HTM = Hardware Transactional Memory
  - Intel Haswell CPU, 2013
  - and others
- STM = Software Transactional Memory

HTM versus STM

- HTM = Hardware Transactional Memory
  - Intel Haswell CPU, 2013
  - and others
- STM = Software Transactional Memory
  - various approaches

HTM versus STM

- HTM = Hardware Transactional Memory
  - Intel Haswell CPU, 2013
  - and others
- STM = Software Transactional Memory
  - various approaches
  - large overhead (2x-10x), but getting faster

HTM versus STM

- HTM = Hardware Transactional Memory
  - Intel Haswell CPU, 2013
  - and others
- STM = Software Transactional Memory
  - various approaches
  - large overhead (2x-10x), but getting faster
  - experimental in PyPy: read/write barriers, as with GC

The catch

The catch

- You Still Have To Use Threads

The catch

- You Still Have To Use Threads

The catch

- You Still Have To Use Threads
- Threads are hard to debug, non-reproductible

The catch

- You Still Have To Use Threads
- Threads are hard to debug, non-reproductible

The catch

- You Still Have To Use Threads
- Threads are hard to debug, non-reproductible
- Threads are Messy

Issue with threads

- TM does not solve this problem:

Issue with threads

- TM does not solve this problem:
- How do you know if you missed a place to put `atomic` around?

Issue with threads

- TM does not solve this problem:
- How do you know if you missed a place to put `atomic` around?
  - hard to catch by writing tests

Issue with threads

- TM does not solve this problem:
- How do you know if you missed a place to put `atomic` around?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes

Issue with threads

- TM does not solve this problem:
- How do you know if you missed a place to put `atomic` around?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes

Issue with threads

- TM does not solve this problem:
- How do you know if you missed a place to put `atomic` around?
  - hard to catch by writing tests
  - instead you get obscure rare run-time crashes

- What if we put `atomic` everywhere?

Analogy with Garbage Collection

- Explicit Memory Management:

Analogy with Garbage Collection

- Explicit Memory Management:
  - messy, hard to debug rare leaks or corruptions

Analogy with Garbage Collection

- Explicit Memory Management:
  - messy, hard to debug rare leaks or corruptions

Analogy with Garbage Collection

- Explicit Memory Management:
  - messy, hard to debug rare leaks or corruptions
- Automatic GC solves it

Analogy with Garbage Collection

- Explicit Memory Management:
  - messy, hard to debug rare leaks or corruptions
- Automatic GC solves it
  - common languages either have a GC or not

Analogy with Garbage Collection

- Explicit Memory Management:
  - messy, hard to debug rare leaks or corruptions
- Automatic GC solves it
  - common languages either have a GC or not
  - if they have a GC, it controls almost *all* objects

Analogy with Garbage Collection

- Explicit Memory Management:
  - messy, hard to debug rare leaks or corruptions
- Automatic GC solves it
  - common languages either have a GC or not
  - if they have a GC, it controls almost *all* objects
  - not just a small part of them

Proposed solution

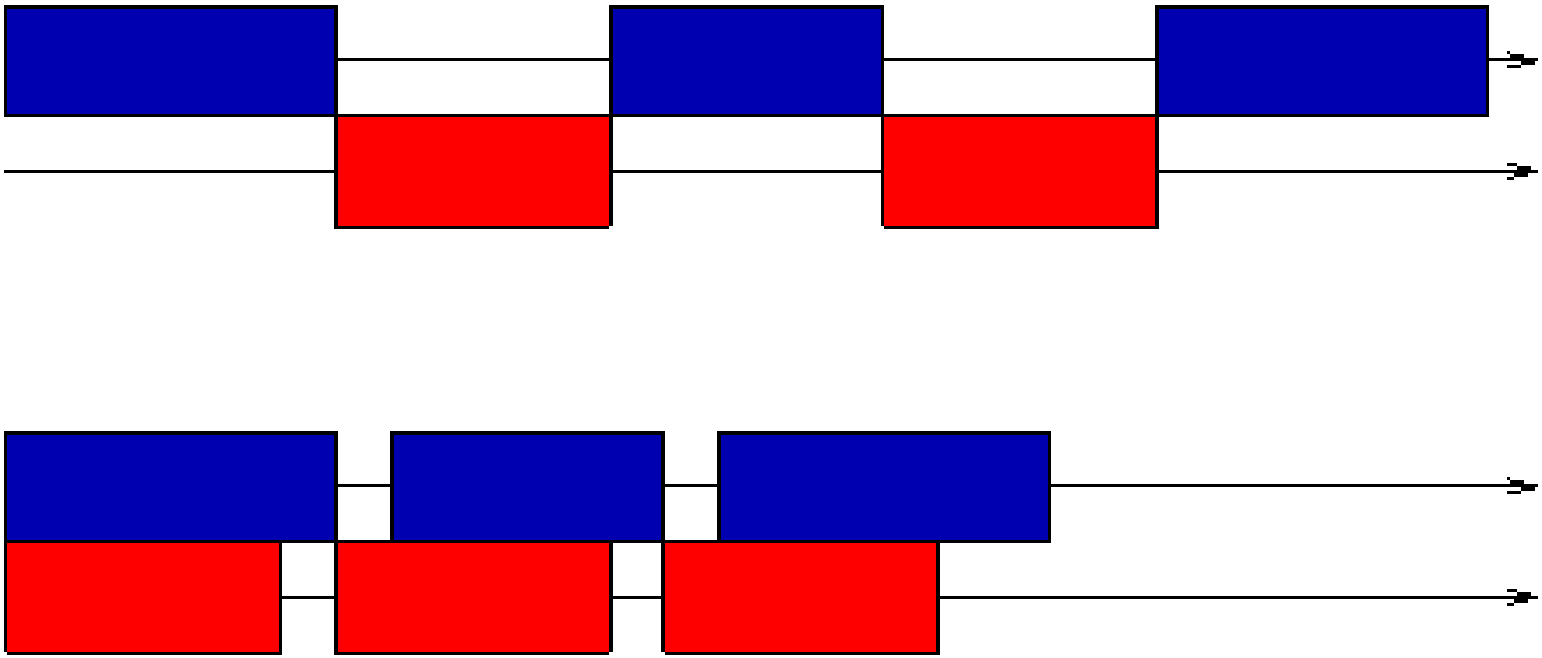
- Put `atomic` everywhere...

Proposed solution

- Put `atomic` everywhere...
- in other words, Run Everything with TM

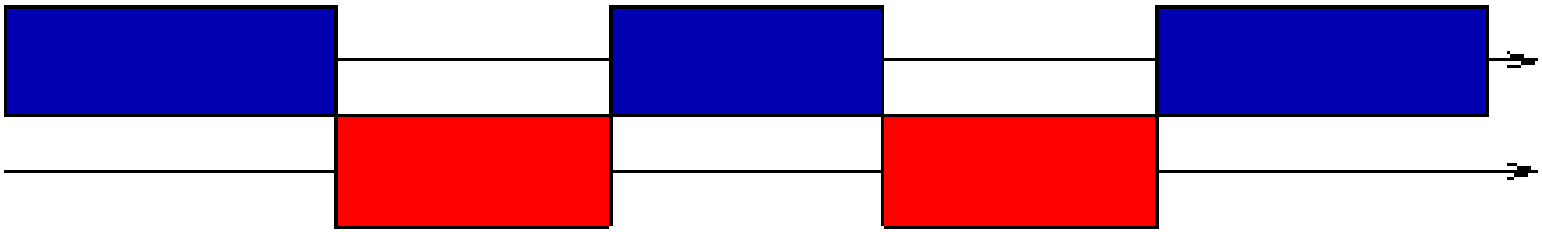
Proposed solution

- Really needs TM. With locks, you'd get this:

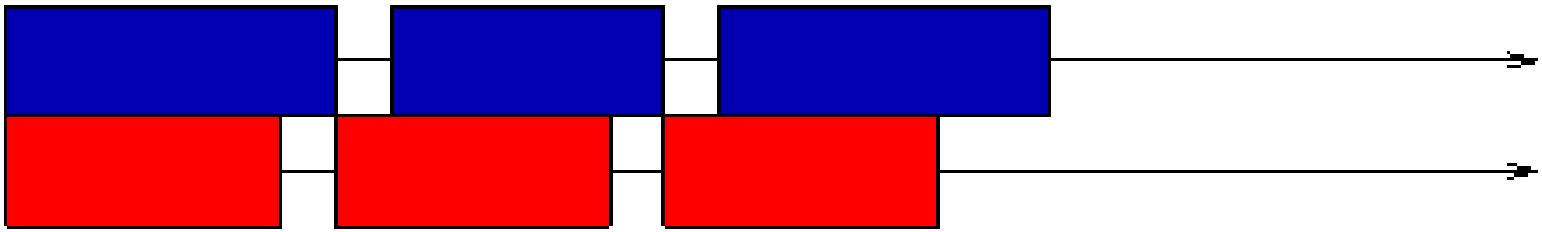


Proposed solution

- Really needs TM. With locks, you'd get this:



- With TM you can get this:



In a few words

- Longer transactions

In a few words

- Longer transactions
- Corresponding to larger parts of the program

In a few words

- Longer transactions
- Corresponding to larger parts of the program
- The underlying multi-threaded model becomes implicit

Typical example

- You want to run `f1()` and `f2()` and `f3()`

Typical example

- You want to run `f1()` and `f2()` and `f3()`

Typical example

- You want to run `f1()` and `f2()` and `f3()`
- Assume they are "mostly independent"

Typical example

- You want to run `f1()` and `f2()` and `f3()`
- Assume they are "mostly independent"
  - i.e. we expect that we can run them in parallel

Typical example

- You want to run `f1()` and `f2()` and `f3()`
- Assume they are "mostly independent"
  - i.e. we expect that we can run them in parallel
  - but we cannot prove it, we just hope that in the common case we can

Typical example

- You want to run `f1()` and `f2()` and `f3()`
- Assume they are "mostly independent"
  - i.e. we expect that we can run them in parallel
  - but we cannot prove it, we just hope that in the common case we can

Typical example

- You want to run `f1()` and `f2()` and `f3()`
- Assume they are "mostly independent"
  - i.e. we expect that we can run them in parallel
  - but we cannot prove it, we just hope that in the common case we can
- In case of conflicts, we don't want random behavior

Typical example

- You want to run `f1()` and `f2()` and `f3()`
- Assume they are "mostly independent"
  - i.e. we expect that we can run them in parallel
  - but we cannot prove it, we just hope that in the common case we can
- In case of conflicts, we don't want random behavior
  - i.e. we don't want thread-like non-determinism and crashes

Pooling and atomic statements

- Solution: use a library that creates a pool of threads

Pooling and atomic statements

- Solution: use a library that creates a pool of threads
- Each thread picks a function from the list and runs it with `atomic`

Results

- The behavior is "as if" we had run `f1()`, `f2()` and `f3()` sequentially

Results

- The behavior is "as if" we had run `f1()`, `f2()` and `f3()` sequentially
- The programmer chooses if he wants this fixed order, or if any order is fine

Results

- The behavior is "as if" we had run `f1()`, `f2()` and `f3()` sequentially
- The programmer chooses if he wants this fixed order, or if any order is fine
- Threads are hidden from the programmer

More generally

- This was an example only

More generally

- This was an example only
- **TM gives various new ways to hide threads under a nice interface**

Not the Ultimate Solution

- Much easier for the programmer to get reproducible results

Not the Ultimate Solution

- Much easier for the programmer to get reproducible results
- But maybe too many conflicts

Not the Ultimate Solution

- Much easier for the programmer to get reproducible results
- But maybe too many conflicts

Not the Ultimate Solution

- Much easier for the programmer to get reproducible results
- But maybe too many conflicts
- "The right side" of the problem

Not the Ultimate Solution

- Much easier for the programmer to get reproducible results
- But maybe too many conflicts

- "The right side" of the problem
  - start with a working program, and improve performance

Not the Ultimate Solution

- Much easier for the programmer to get reproducible results
- But maybe too many conflicts

- "The right side" of the problem
  - start with a working program, and improve performance
  - as opposed to: with locks, start with a fast program, and debug crashes

Not the Ultimate Solution

- Much easier for the programmer to get reproducible results
- But maybe too many conflicts

- "The right side" of the problem
  - start with a working program, and improve performance
  - as opposed to: with locks, start with a fast program, and debug crashes
  - we will need new debugging tools

PyPy-STM

- PyPy-STM: a version of PyPy with Software Transactional Memory

PyPy-STM

- PyPy-STM: a version of PyPy with Software Transactional Memory
  - in-progress, but basically working

PyPy-STM

- PyPy-STM: a version of PyPy with Software Transactional Memory
  - in-progress, but basically working
  - solves the "GIL issue" but more importantly adds `atomic`

PyPy-STM

- PyPy-STM: a version of PyPy with Software Transactional Memory
  - in-progress, but basically working
  - solves the "GIL issue" but more importantly adds `atomic`
- `http://pypy.org/`

PyPy-STM

- PyPy-STM: a version of PyPy with Software Transactional Memory
  - in-progress, but basically working
  - solves the "GIL issue" but more importantly adds `atomic`
- `http://pypy.org/`

PyPy-STM

- PyPy-STM: a version of PyPy with Software Transactional Memory
  - in-progress, but basically working
  - solves the "GIL issue" but more importantly adds `atomic`
- `http://pypy.org/`
- Thank you!