Australian National University

IBM

Work–Stealing Without The Baggage

Vivek Kumar[1], Daniel Frampton[1, 2], Stephen M Blackburn[1], David Grove[3], Olivier Tardieu[3]

1 The Australian National University
2 Microsoft
3 IBM T.J. Watson Research

"What Andy giveth, Bill taketh away"

"The Free Lunch is Now Over!"

Herb Sutter, Dr. Dobb's Journal, March 2005

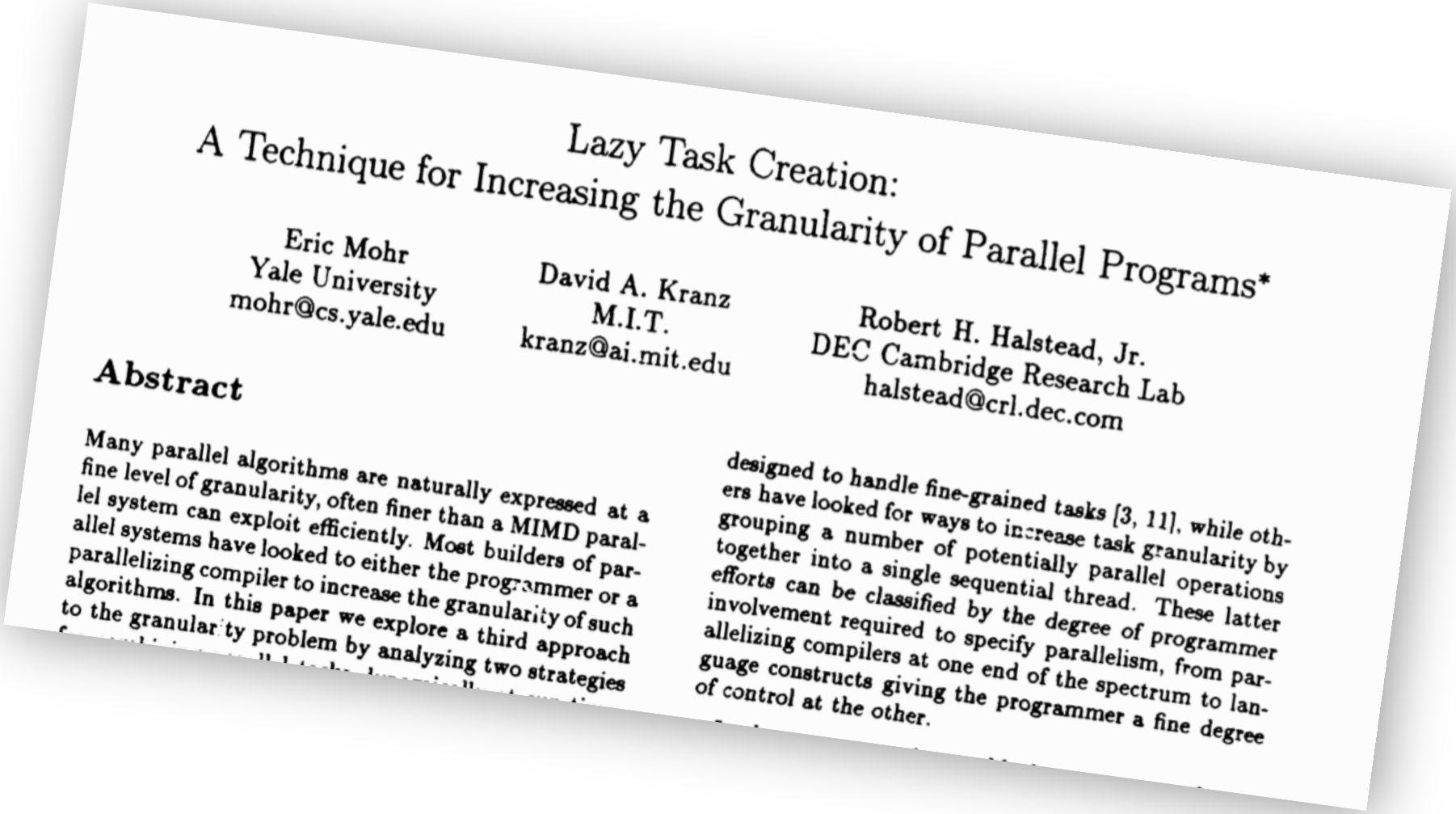
Hardware and Software Today

- Commodity processors with parallel execution abilities
- A fundamental turn toward concurrency in software

The Challenge

- Modern hardware requires s/w parallelism
- Software parallelism difficult to identify, expose
  - Hard coded optimizations may get you there…
- Hard to realize potential of modern processors

Goal: performance ***and*** productivity



Lazy Task Creation:
A Technique for Increasing the Granularity of Parallel Programs*

Eric Mohr
Yale University
mohr@cs.yale.edu

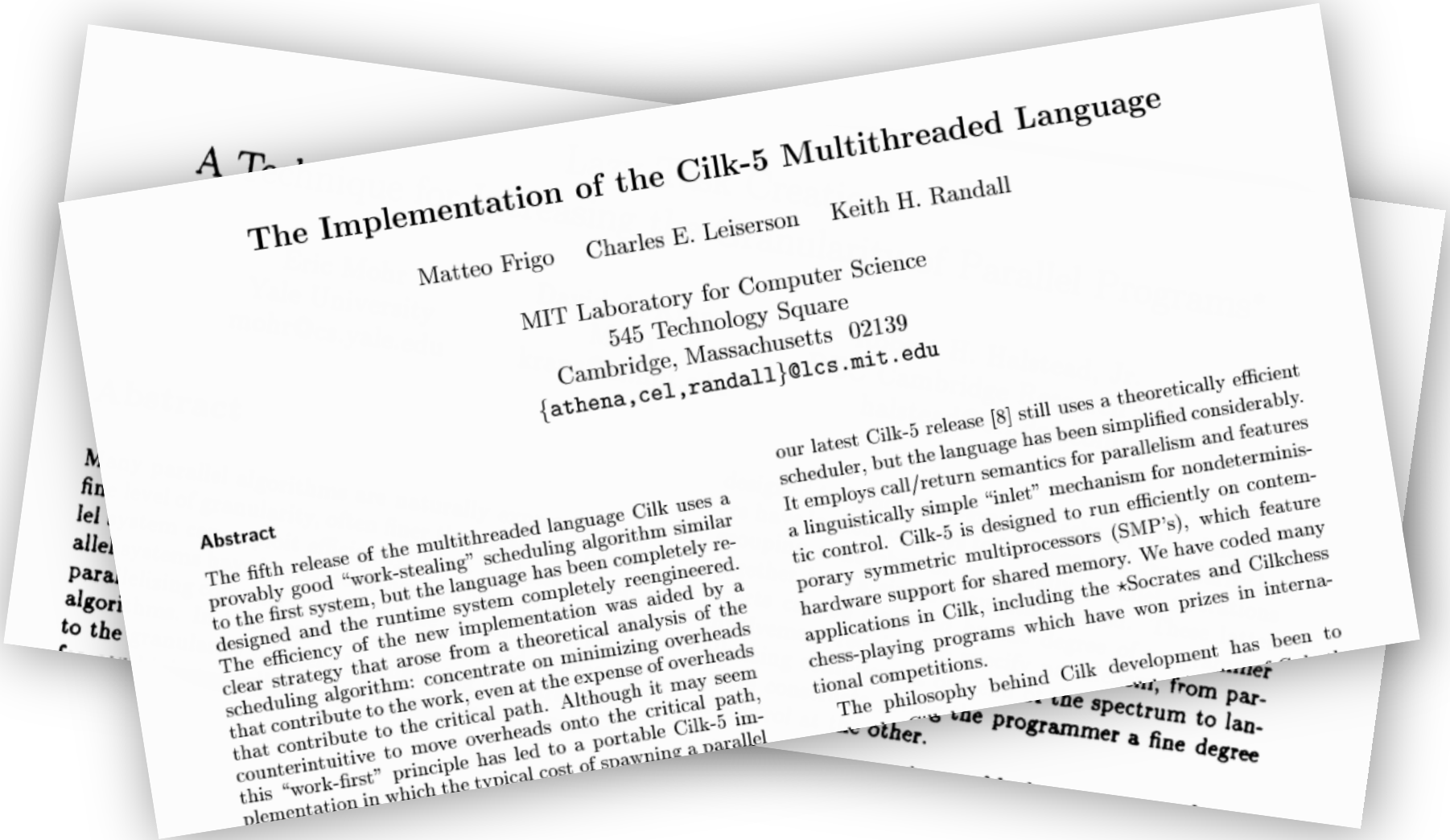
David A. Kranz
M.I.T.
kranz@ai.mit.edu

Robert H. Halstead, Jr.
DEC Cambridge Research Lab
halstead@crl.dec.com

Abstract

Many parallel algorithms are naturally expressed at a fine level of granularity, often finer than a MIMD parallel system can exploit efficiently. Most builders of parallel systems have looked to either the programmer or a parallelizing compiler to increase the granularity of such algorithms. In this paper we explore a third approach to the granularity problem by analyzing two strategies

designed to handle fine-grained tasks [3, 11], while others have looked for ways to increase task granularity by grouping a number of potentially parallel operations together into a single sequential thread. These latter efforts can be classified by the degree of programmer involvement required to specify parallelism, from parallelizing compilers at one end of the spectrum to language constructs giving the programmer a fine degree of control at the other.



The Implementation of the Cilk-5 Multithreaded Language

Matteo Frigo Charles E. Leiserson Keith H. Randall

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139
{athena,cel,randall}@lcs.mit.edu

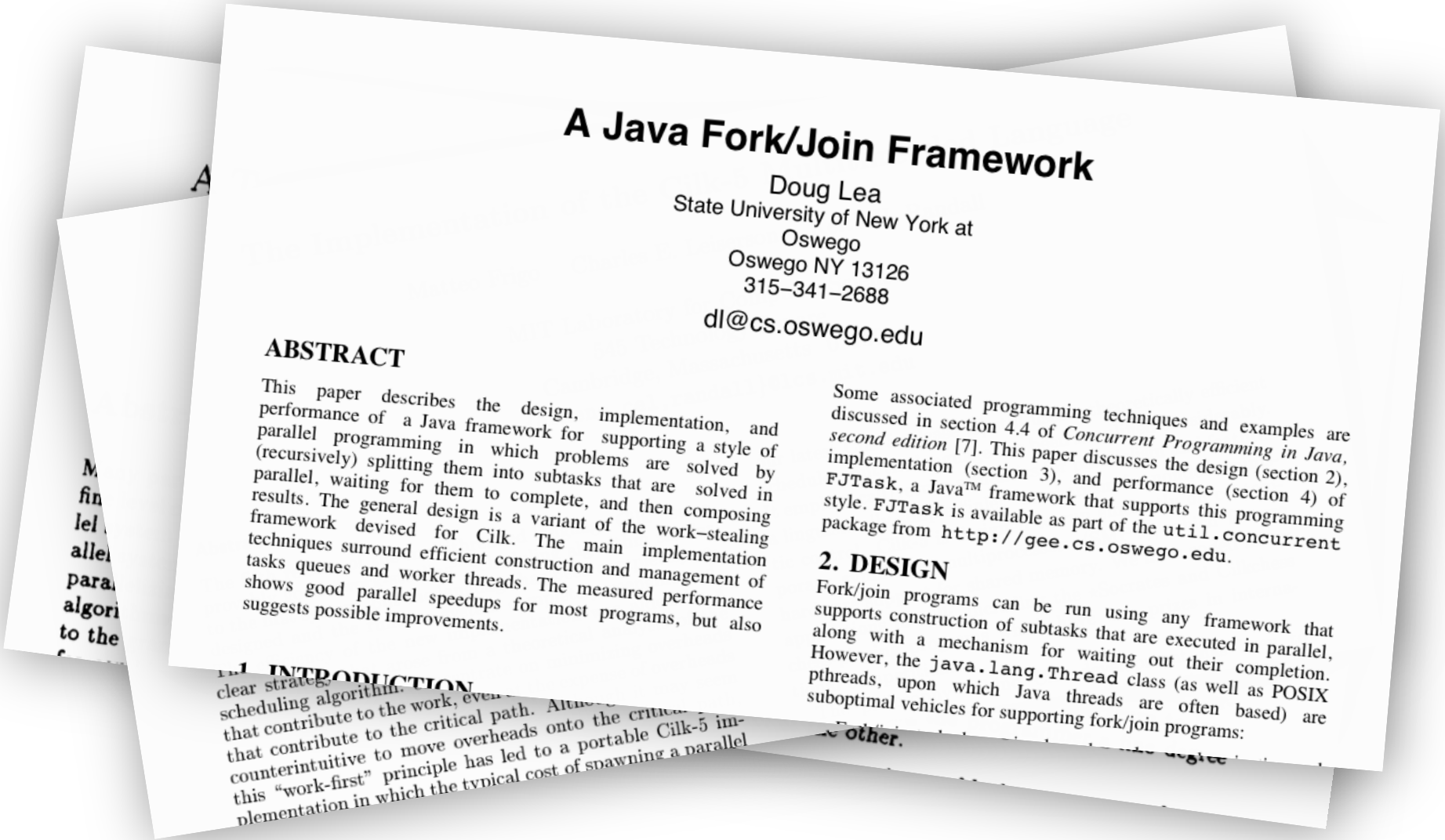
Abstract

The fifth release of the multithreaded language Cilk uses a provably good "work-stealing" scheduling algorithm similar to the first system, but the language has been completely redesigned and the runtime system completely reengineered. The efficiency of the new implementation was aided by a clear strategy that arose from a theoretical analysis of the scheduling algorithm: concentrate on minimizing overheads that contribute to the work, even at the expense of overheads that contribute to the critical path. Although it may seem counterintuitive to move overheads onto the critical path, this "work-first" principle has led to a portable Cilk-5 implementation in which the typical cost of spawning a parallel

our latest Cilk-5 release [8] still uses a theoretically efficient scheduler, but the language has been simplified considerably. It employs call/return semantics for parallelism and features a linguistically simple "inlet" mechanism for nondeterministic control. Cilk-5 is designed to run efficiently on contemporary symmetric multiprocessors (SMP's), which feature hardware support for shared memory. We have coded many applications in Cilk, including the ⋆Socrates and Cilkchess chess-playing programs which have won prizes in international competitions.

The philosophy behind Cilk development has been to

Background



A Java Fork/Join Framework

Doug Lea
State University of New York at Oswego
Oswego NY 13126
315–341–2688
dl@cs.oswego.edu

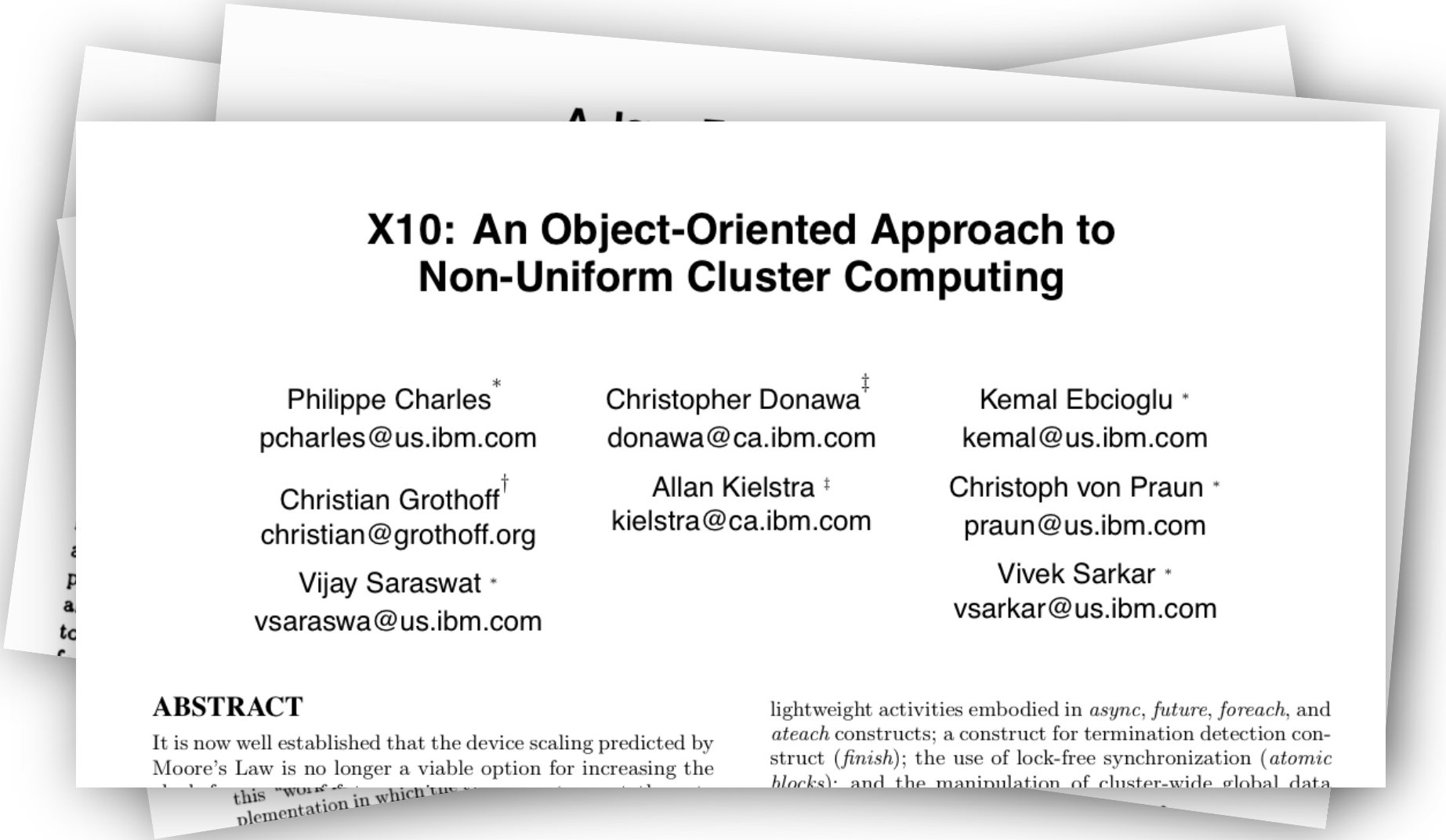
ABSTRACT

This paper describes the design, implementation, and performance of a Java framework for supporting a style of parallel programming in which problems are solved by (recursively) splitting them into subtasks that are solved in parallel, waiting for them to complete, and then composing results. The general design is a variant of the work–stealing framework devised for Cilk. The main implementation techniques surround efficient construction and management of tasks queues and worker threads. The measured performance shows good parallel speedups for most programs, but also suggests possible improvements.

Some associated programming techniques and examples are discussed in section 4.4 of *Concurrent Programming in Java, second edition* [7]. This paper discusses the design (section 2), implementation (section 3), and performance (section 4) of `FJTask`, a Java™ framework that supports this programming style. `FJTask` is available as part of the `util.concurrent` package from `http://gee.cs.oswego.edu`.

2. DESIGN

Fork/join programs can be run using any framework that supports construction of subtasks that are executed in parallel, along with a mechanism for waiting out their completion. However, the `java.lang.Thread` class (as well as POSIX pthreads, upon which Java threads are often based) are suboptimal vehicles for supporting fork/join programs:



X10: An Object-Oriented Approach to Non-Uniform Cluster Computing

Philippe Charles *
pcharles@us.ibm.com

Christopher Donawa ‡
donawa@ca.ibm.com

Kemal Ebcioglu *
kemal@us.ibm.com

Christian Grothoff †
christian@grothoff.org

Allan Kielstra ‡
kielstra@ca.ibm.com

Christoph von Praun *
praun@us.ibm.com

Vijay Saraswat *
vsaraswa@us.ibm.com

Vivek Sarkar *
vsarkar@us.ibm.com

ABSTRACT

It is now well established that the device scaling predicted by Moore's Law is no longer a viable option for increasing the

lightweight activities embodied in *async*, *future*, *foreach*, and *ateach* constructs; a construct for termination detection construct (*finish*); the use of lock-free synchronization (*atomic blocks*); and the manipulation of cluster-wide global data

Background

- Language based features to expose parallelism
  - Dynamic task parallelism
  - Work–stealing scheduler
- A runtime to hide the hardware complexities

Contributions

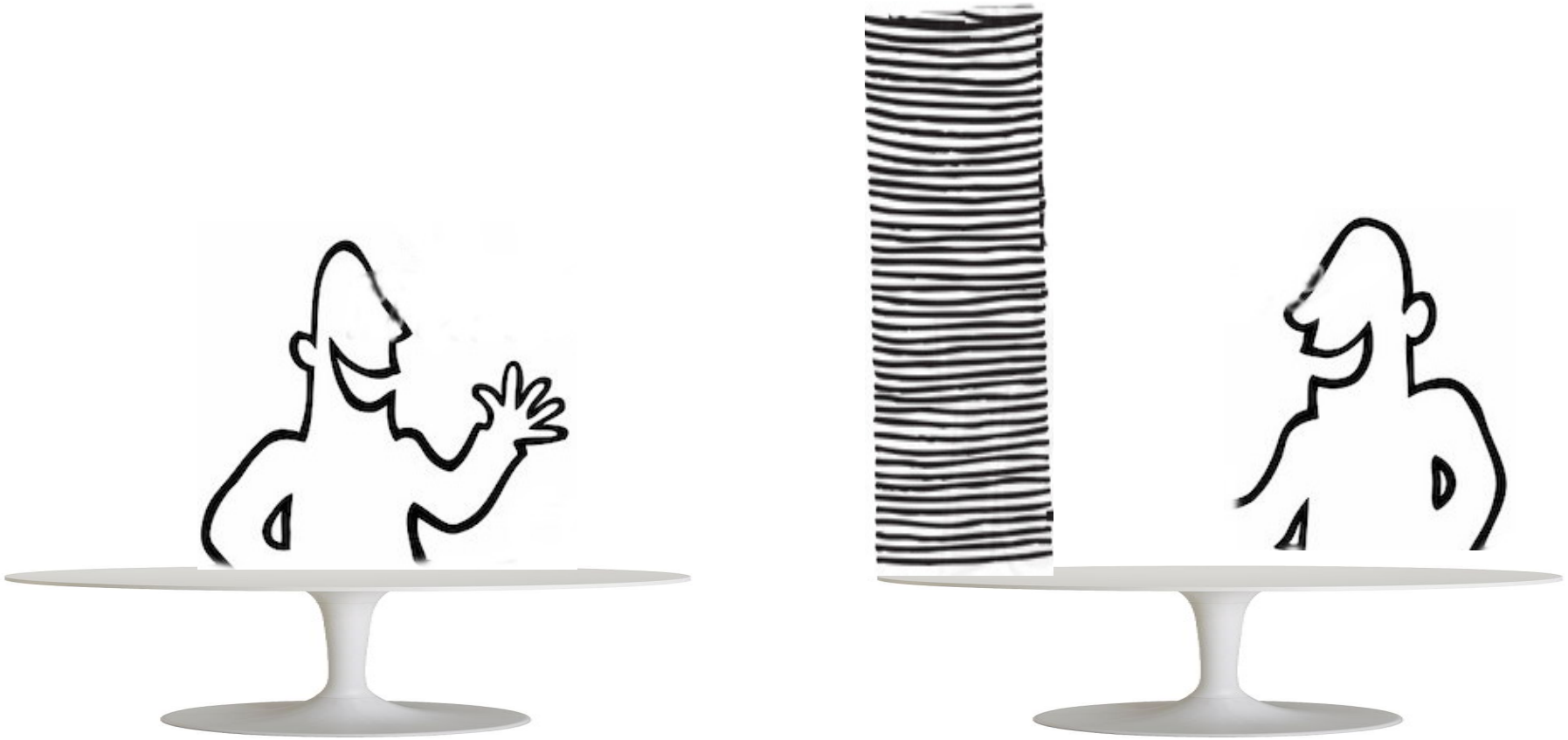
- ✔ In-depth analysis
  - Of sources of overheads in existing work-stealing implementations
- ✔ Two new configurations
  - That leverage mechanisms within modern JVMs
- ✔ Detailed performance study
  - Using standard work-stealing benchmarks
- ✔ Results
  - That shows we can almost completely remove sequential overhead from a work-stealing implementation

Understanding Work–Stealing

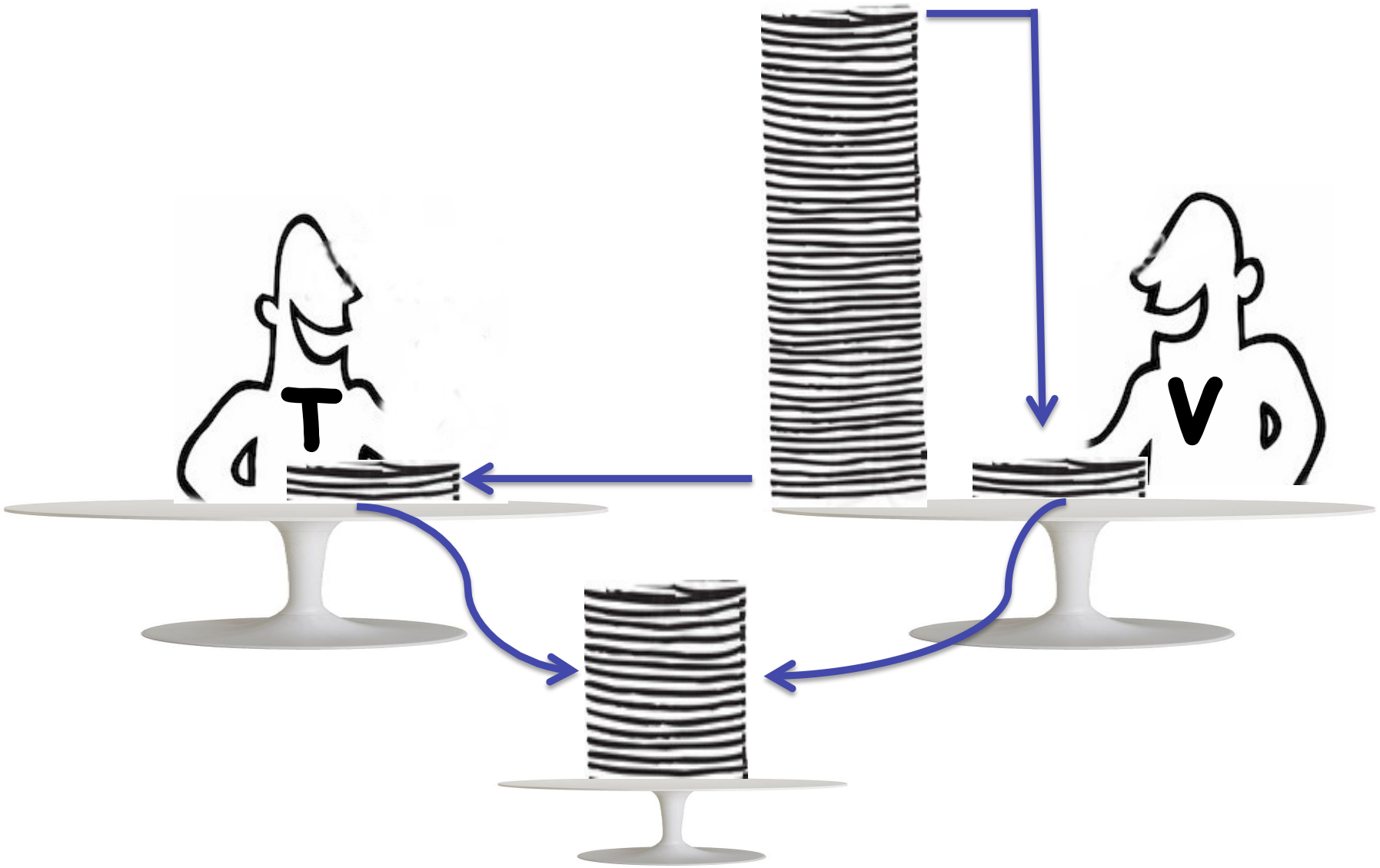
Work–Stealing

Work–Stealing

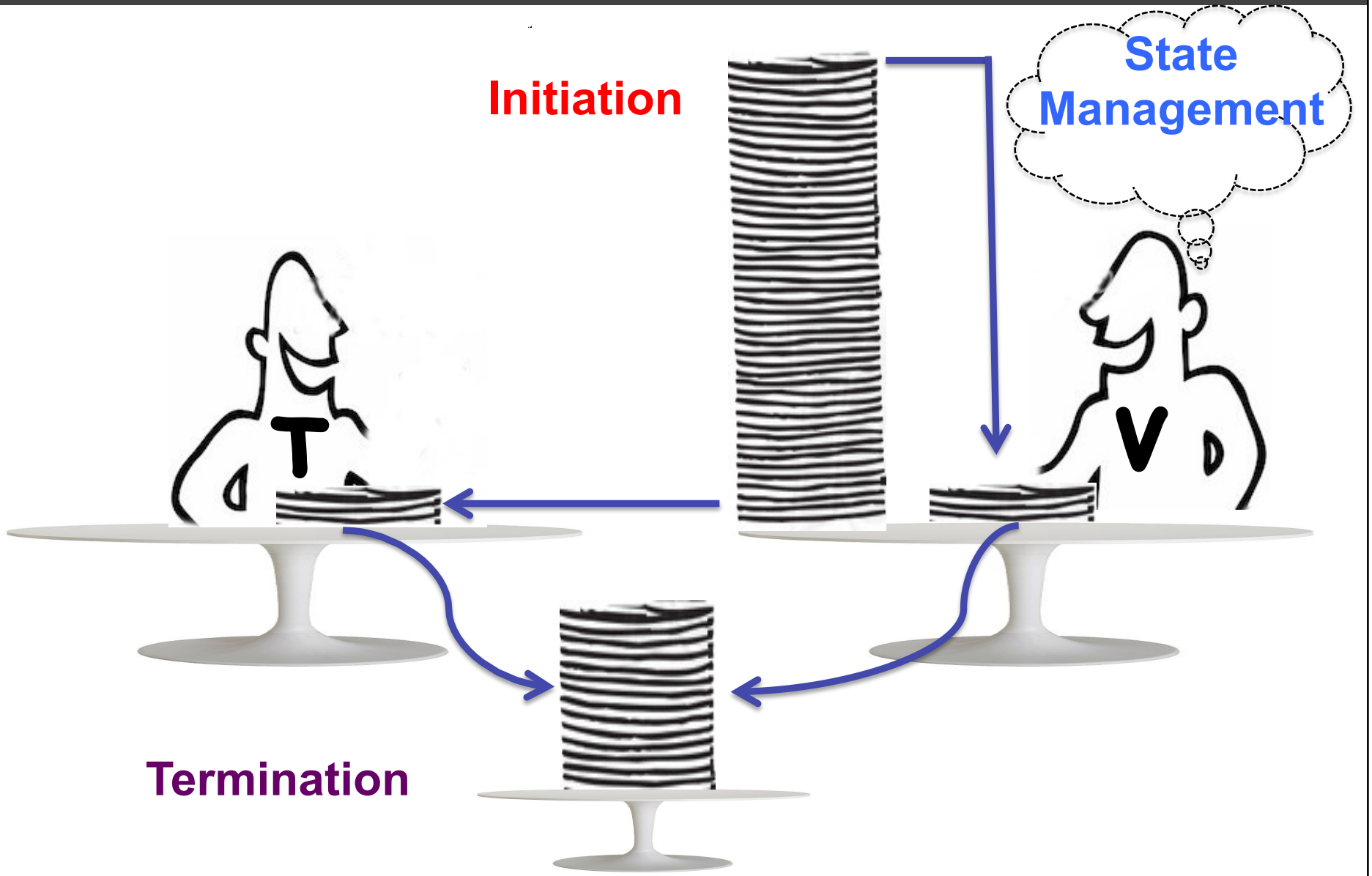
Work–Stealing



Work–Stealing



Work–Stealing

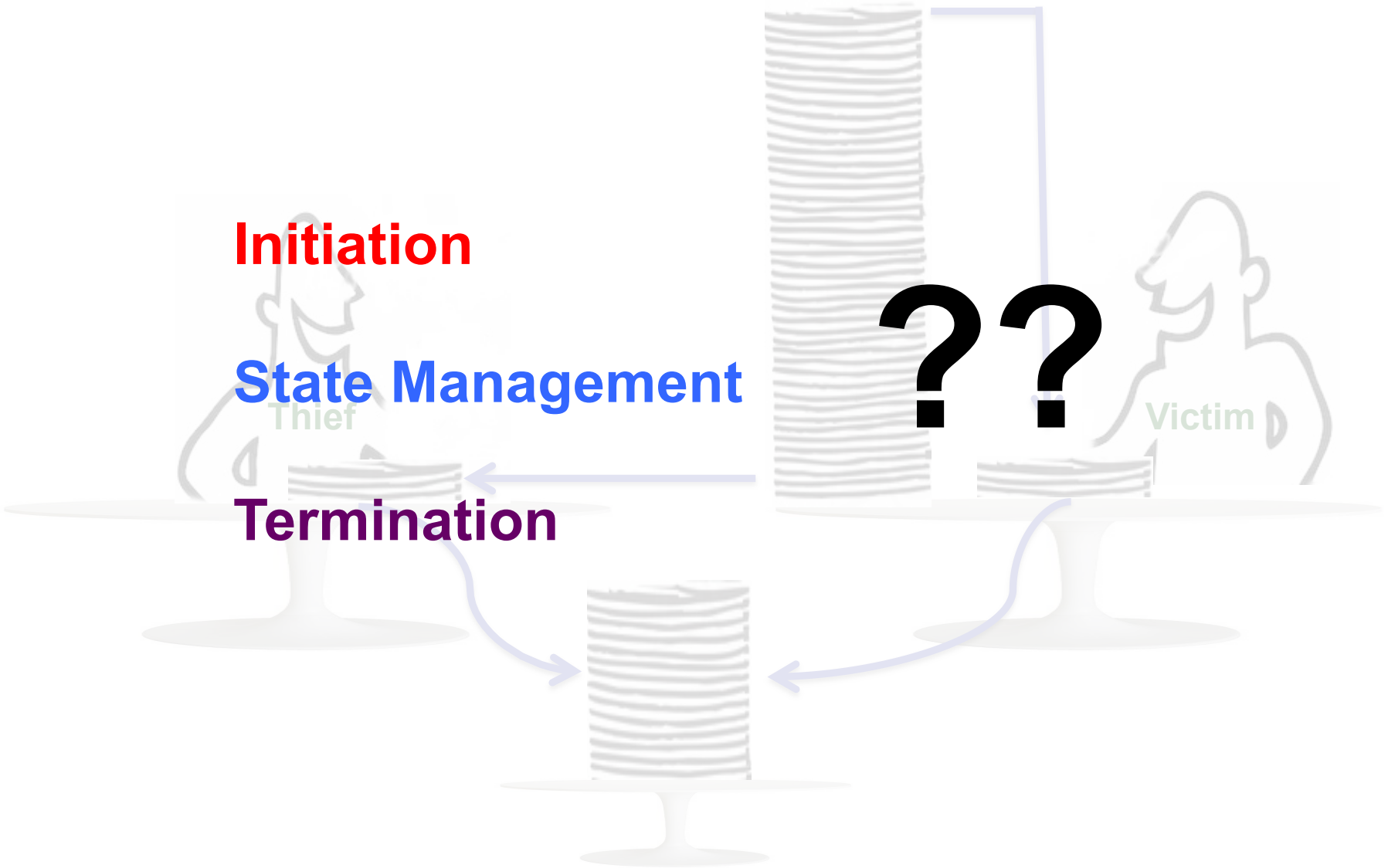


Work–Stealing

Initiation

State Management

Termination

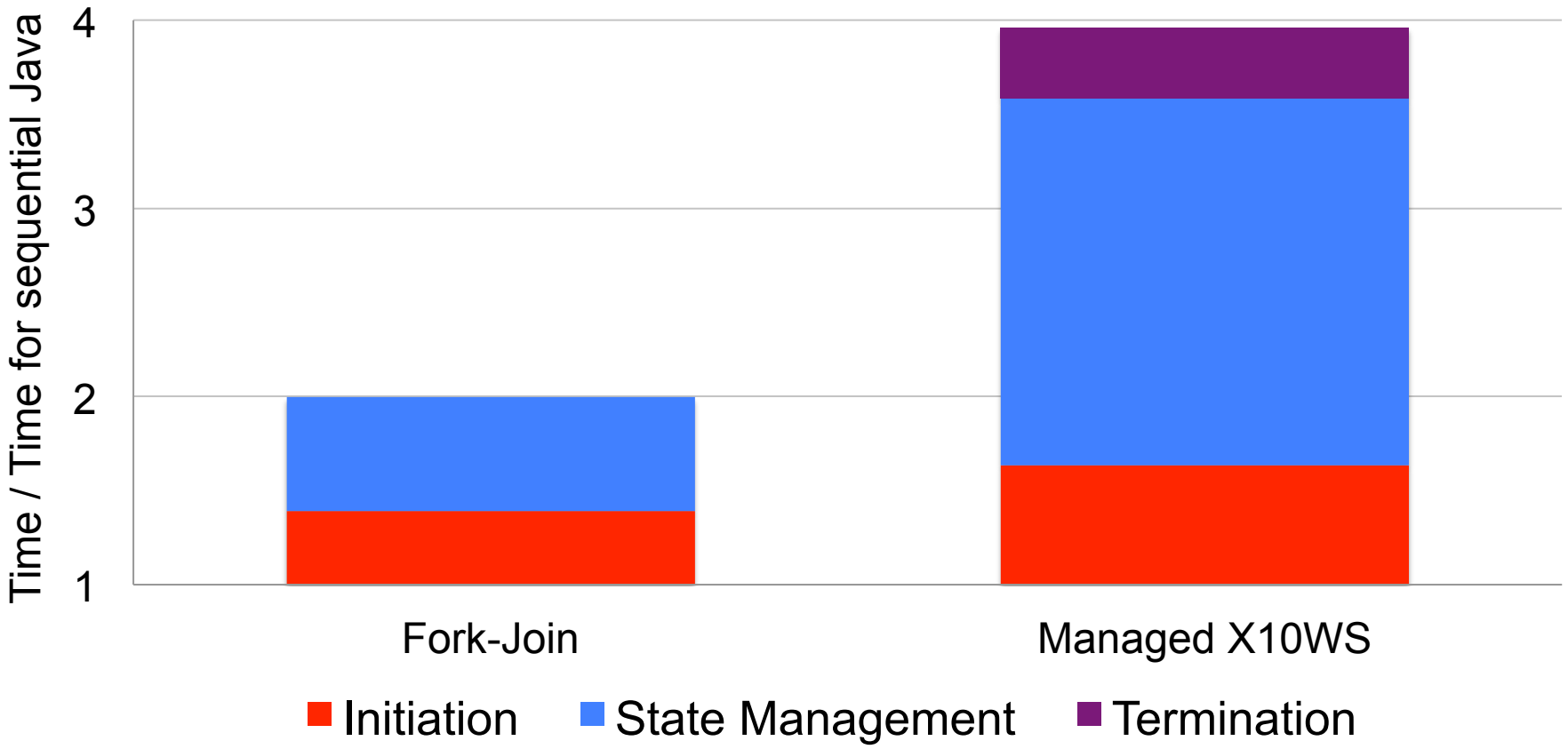


Motivating Analysis

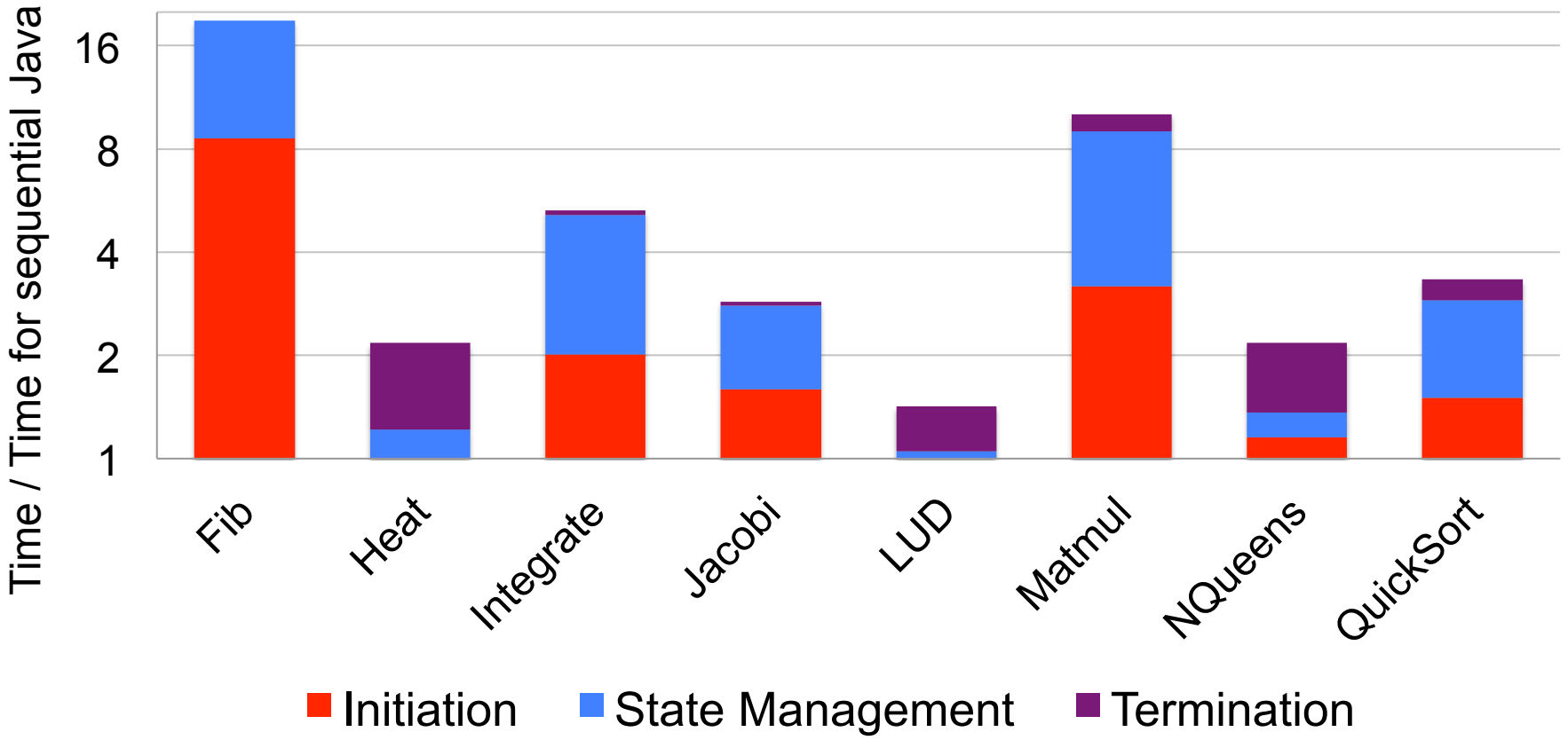
Methodology

- Benchmarks
  - Fibonacci
  - Integrate
  - Jacobi
  - Matmul
  - Nqueens
  - Quicksort
  - LU Decomposition
  - Heat Diffusion
- Hardware Platform
  - 2 Intel Xeon E7530
    - 6 cores each
- Software Platform
  - Jikes RVM (3.1.2)

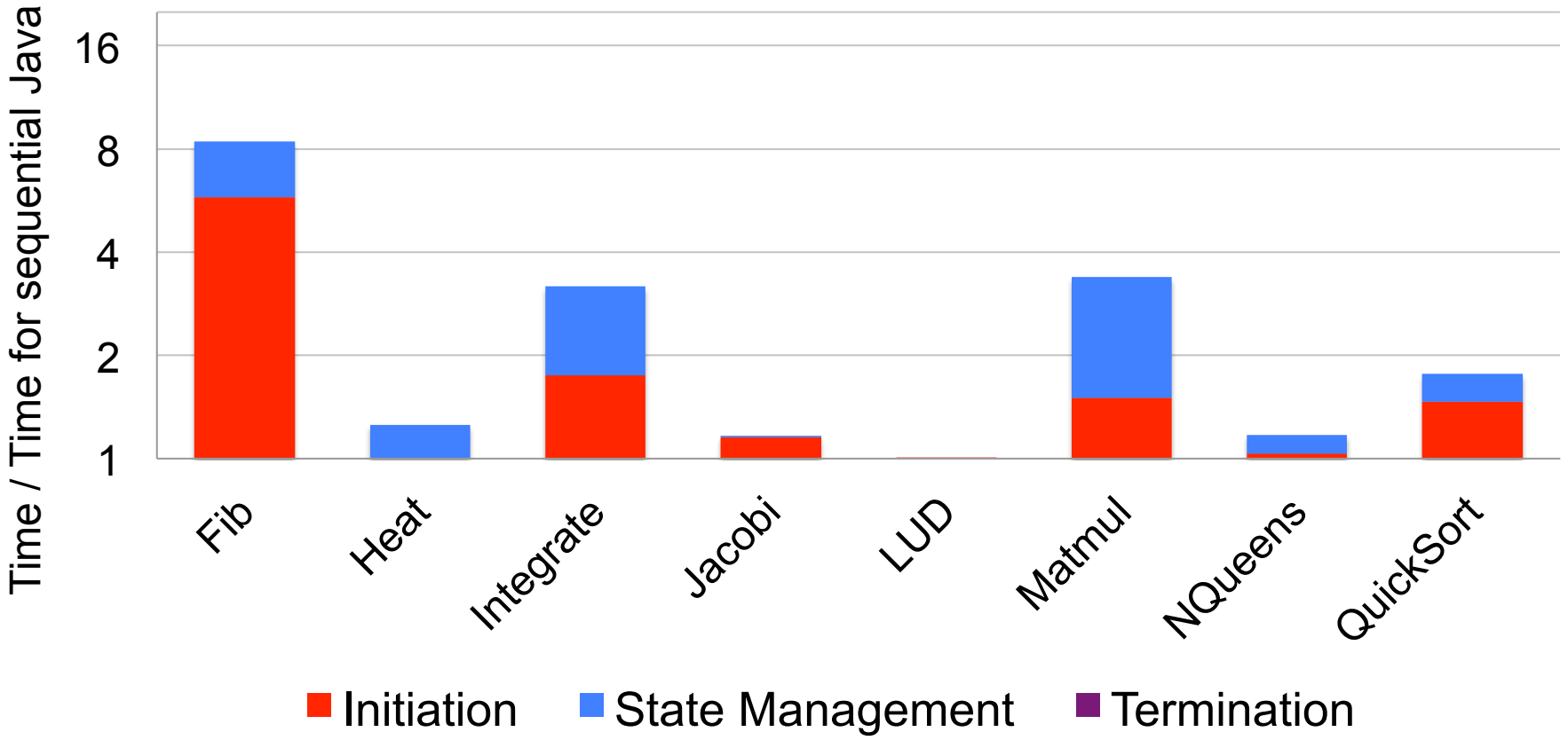
Sequential Overhead



Sequential Overhead: Managed X10WS

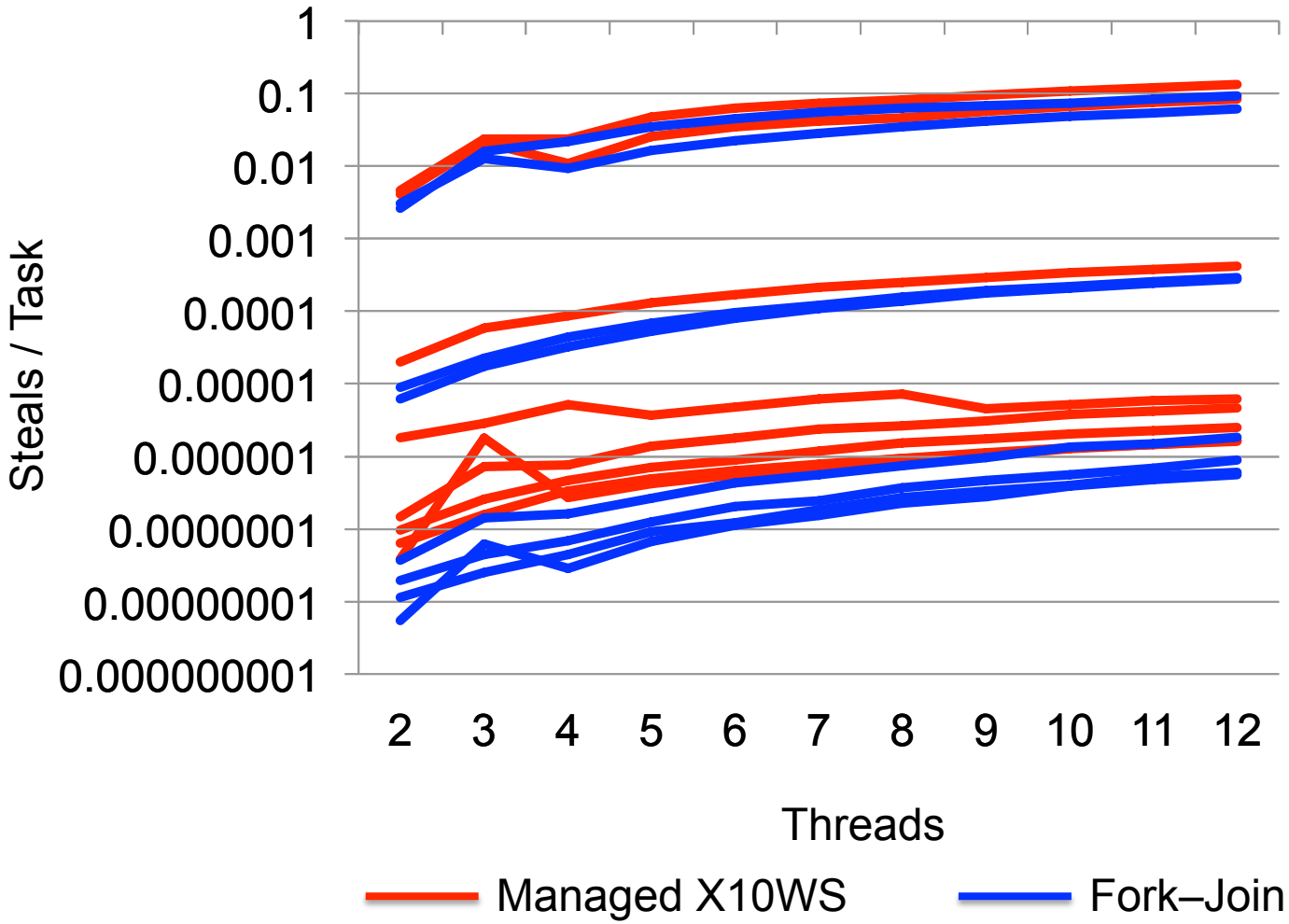


Sequential Overhead: Fork–Join



Motivating Analysis

Steals To Task Ratio



Motivating Analysis

- Sequential overheads in work–stealing
  - Initiation
  - State management
  - Termination
- Low steals to task ratio
  - Most tasks consumed locally by the victim

Insights

- Move the overheads from common case to the rare case
- Re-use existing mechanisms inside modern JVMs

Approach

- Initiation
  - ✔Victim's execution stack used for initiation
- State management
  - ✔Extract state directly from victim's stack & registers
- Termination
  - ✔Victim dynamically switch versions of the code
  - ✔Java exception handling (try–catch blocks)

Implementation

Two Configurations

- X10WS (OffStack)
  - ✔ Victim's execution stack used for initiation
  - ✔ Extract state directly from victim's stack & registers
  - ✔ Victim dynamically switch versions of the code

- X10 (Try–Catch)
  - – All of the above, plus:
  - ✔ Exploit try-catch semantics and leverage fast exception handling in modern JVMs
  - ✔ Compiler support for try-catch code generation

X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

```
X = S1();
```

```
Y = S2();
```

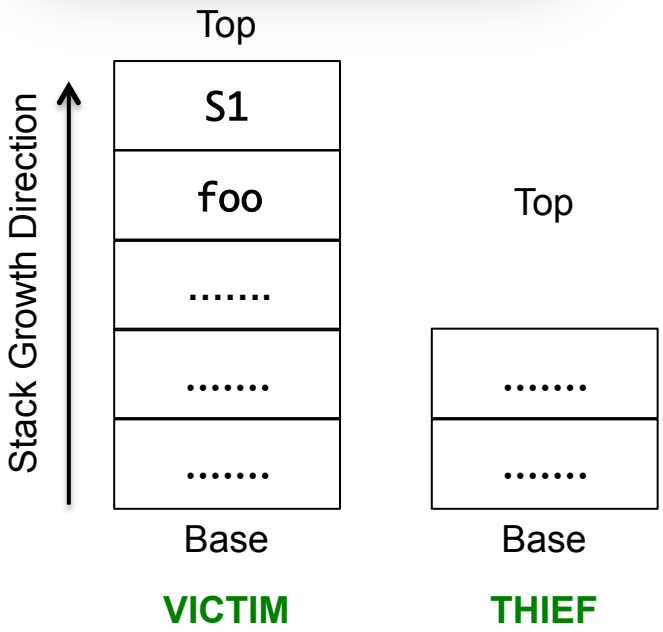
X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();

    }



    } catch (ExceptionEntryThief t) {

    }
    Y = S2();
```

X10 (Try–Catch)

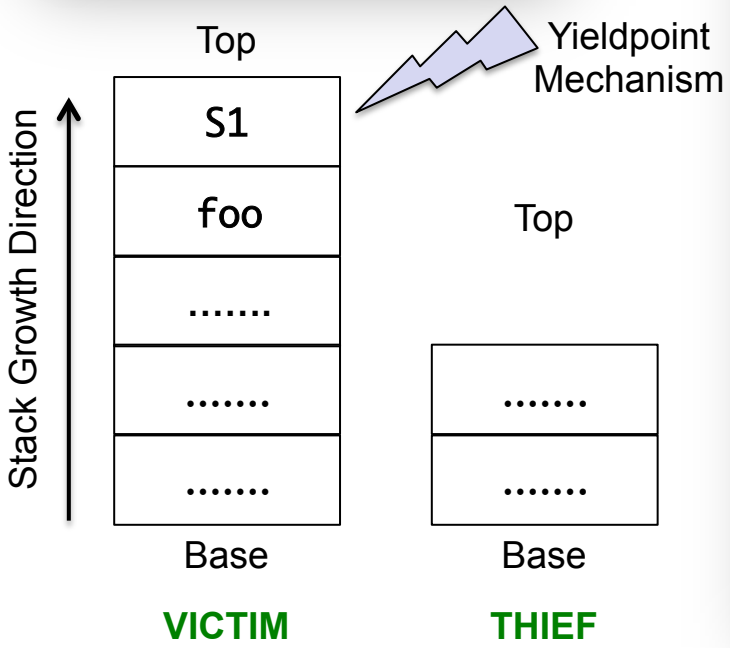
```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```




```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();

    }



    } catch (ExceptionEntryThief t) {

    }
    Y = S2();
```

X10 (Try–Catch)

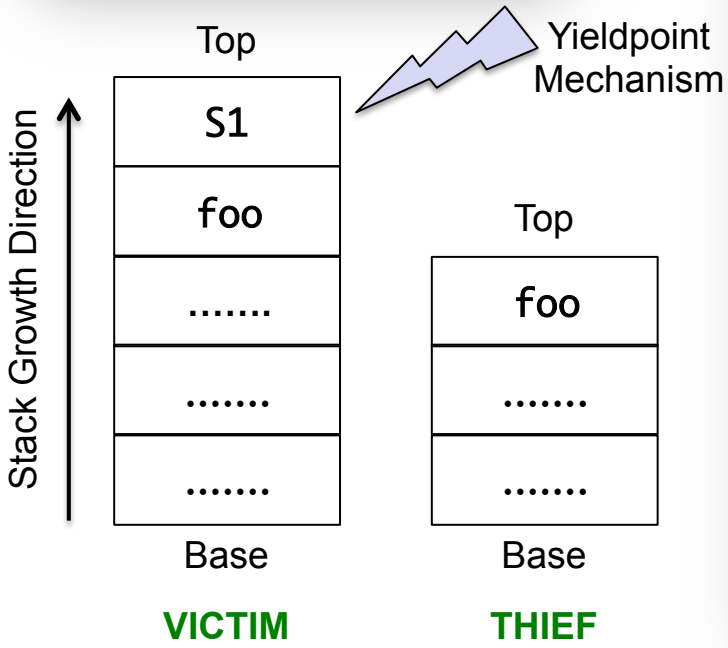
```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```




```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();

    }



    } catch (ExceptionEntryThief t) {

    }
    Y = S2();
```

X10 (Try–Catch)

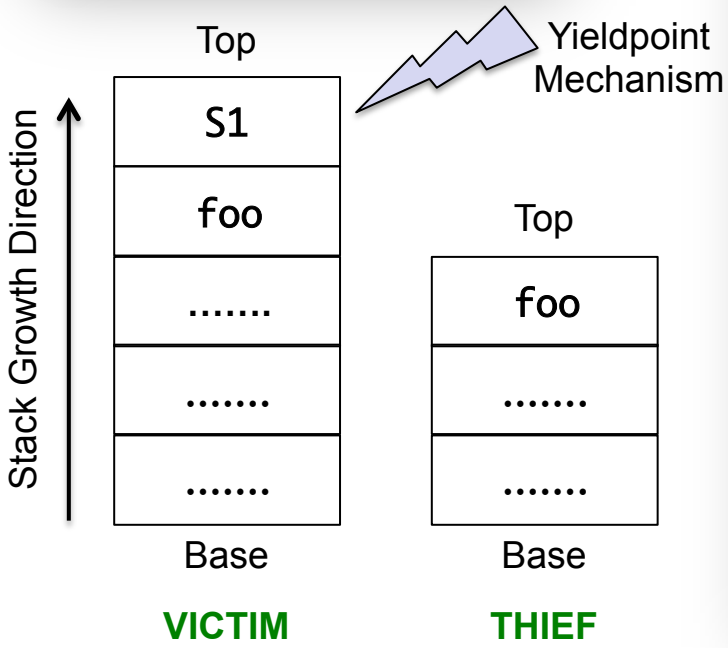
```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```




```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();

    }



    } catch (ExceptionEntryThief t) {

    }
    Y = S2();
```

X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```




```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();

    }



    } catch (ExceptionEntryThief t) {
        // Entrypoint for Thief
    }
    Y = S2();
```

X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();
        // if S2 stolen then throw ExceptionVictim
    }



    } catch (ExceptionEntryThief t) {
        // Entrypoint for Thief
    }
    Y = S2();
    Runtime.finish() // Thief throw ExceptionThief
}  catch (ExceptionThief e) {
    // 1. Runtime stores result Y
    // 2. if S1 not complete, become a Thief

}
```

X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();
        // if S2 stolen then throw ExceptionVictim
    } catch (ExceptionVictim v) {
        // 1. Runtime stores result X
        // 2. if S2 not complete, become a Thief

    } catch (ExceptionEntryThief t) {
        // Entrypoint for Thief
    }
    Y = S2();
    Runtime.finish() // Thief throw ExceptionThief
}  catch (ExceptionThief e) {
    // 1. Runtime stores result Y
    // 2. if S1 not complete, become a Thief

}
```

X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();
        // if S2 stolen then throw ExceptionVictim
    } catch (ExceptionVictim v) {
        // 1. Runtime stores result X
        // 2. if S2 not complete, become a Thief
        // 3. else throw ExceptionFetch
    } catch (ExceptionEntryThief t) {
        // Entrypoint for Thief
    }
    Y = S2();
    Runtime.finish() // Thief throw ExceptionThief
}  catch (ExceptionThief e) {
    // 1. Runtime stores result Y
    // 2. if S1 not complete, become a Thief

}  catch (ExceptionFetch e) {
    // if Victim then get Y from runtime

}
```

X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();
        // if S2 stolen then throw ExceptionVictim
    } catch (ExceptionVictim v) {
        // 1. Runtime stores result X
        // 2. if S2 not complete, become a Thief
        // 3. else throw ExceptionFetch
    } catch (ExceptionEntryThief t) {
        // Entrypoint for Thief
    }
    Y = S2();
    Runtime.finish() // Thief throw ExceptionThief
}  catch (ExceptionThief e) {
    // 1. Runtime stores result Y
    // 2. if S1 not complete, become a Thief
    // 3. else throw ExceptionFetch
}  catch (ExceptionFetch e) {
    // if Victim then get Y from runtime
    // if Thief then get X from runtime
}
```

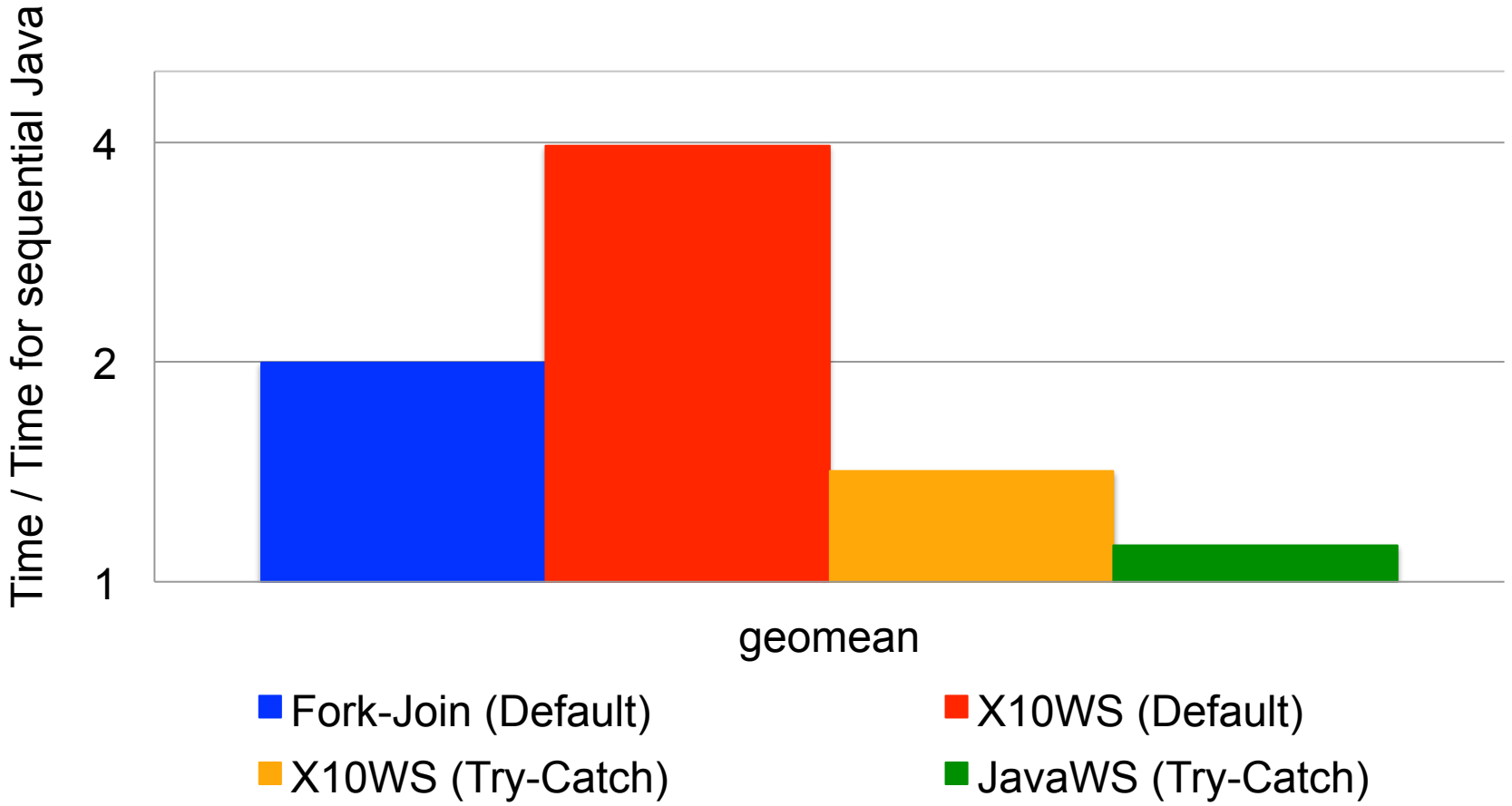
X10 (Try–Catch)

```
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}
```

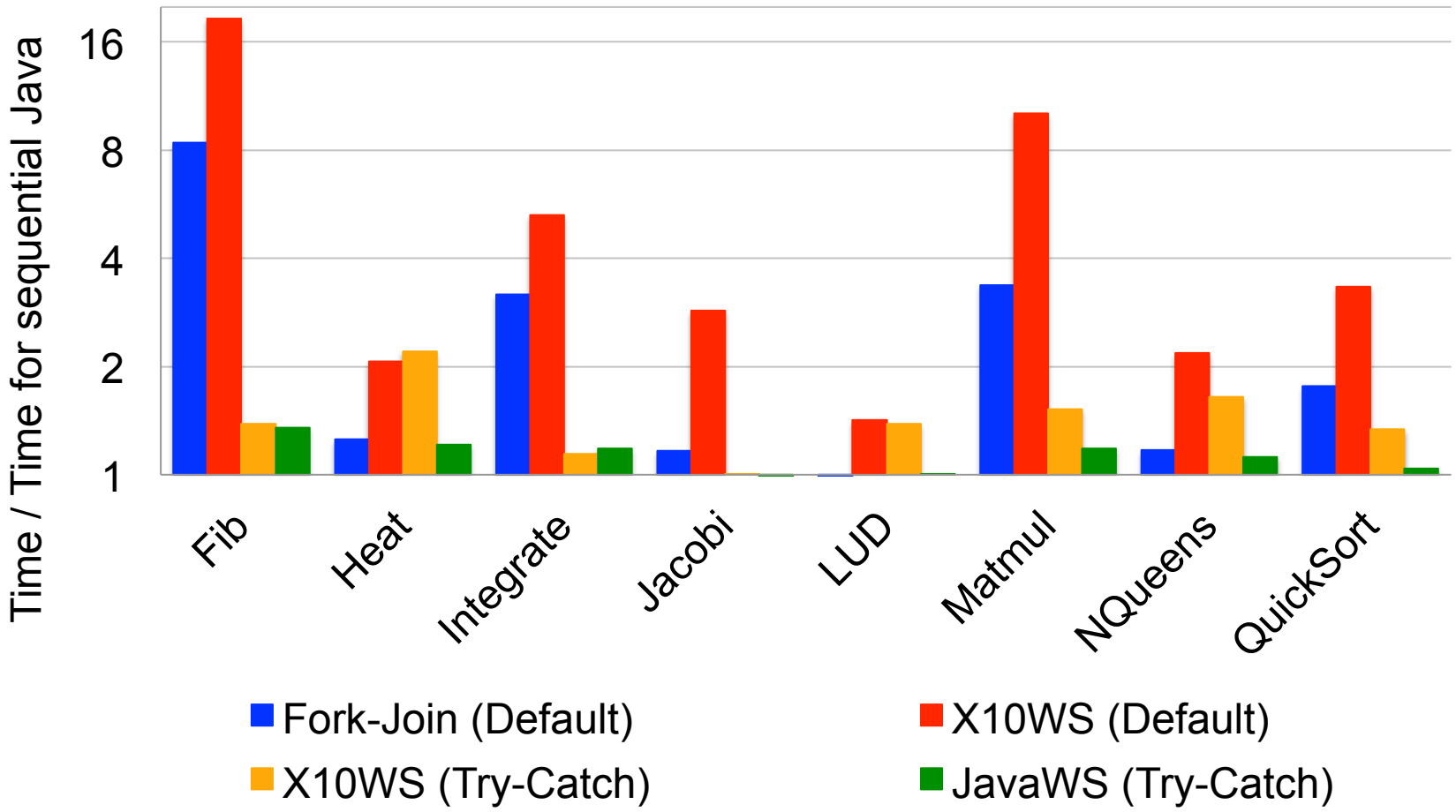
```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();
        // if S2 stolen then throw ExceptionVictim
    } catch (ExceptionVictim v) {
        // 1. Runtime stores result X
        // 2. if S2 not complete, become a Thief
        // 3. else throw ExceptionFetch
    } catch (ExceptionEntryThief t) {
        // Entrypoint for Thief
    }
    Y = S2();
    Runtime.finish() // Thief throw ExceptionThief
}  catch (ExceptionThief e) {
    // 1. Runtime stores result Y
    // 2. if S1 not complete, become a Thief
    // 3. else throw ExceptionFetch
}  catch (ExceptionFetch e) {
    // if Victim then get Y from runtime
    // if Thief then get X from runtime
}
```

Performance Evaluation

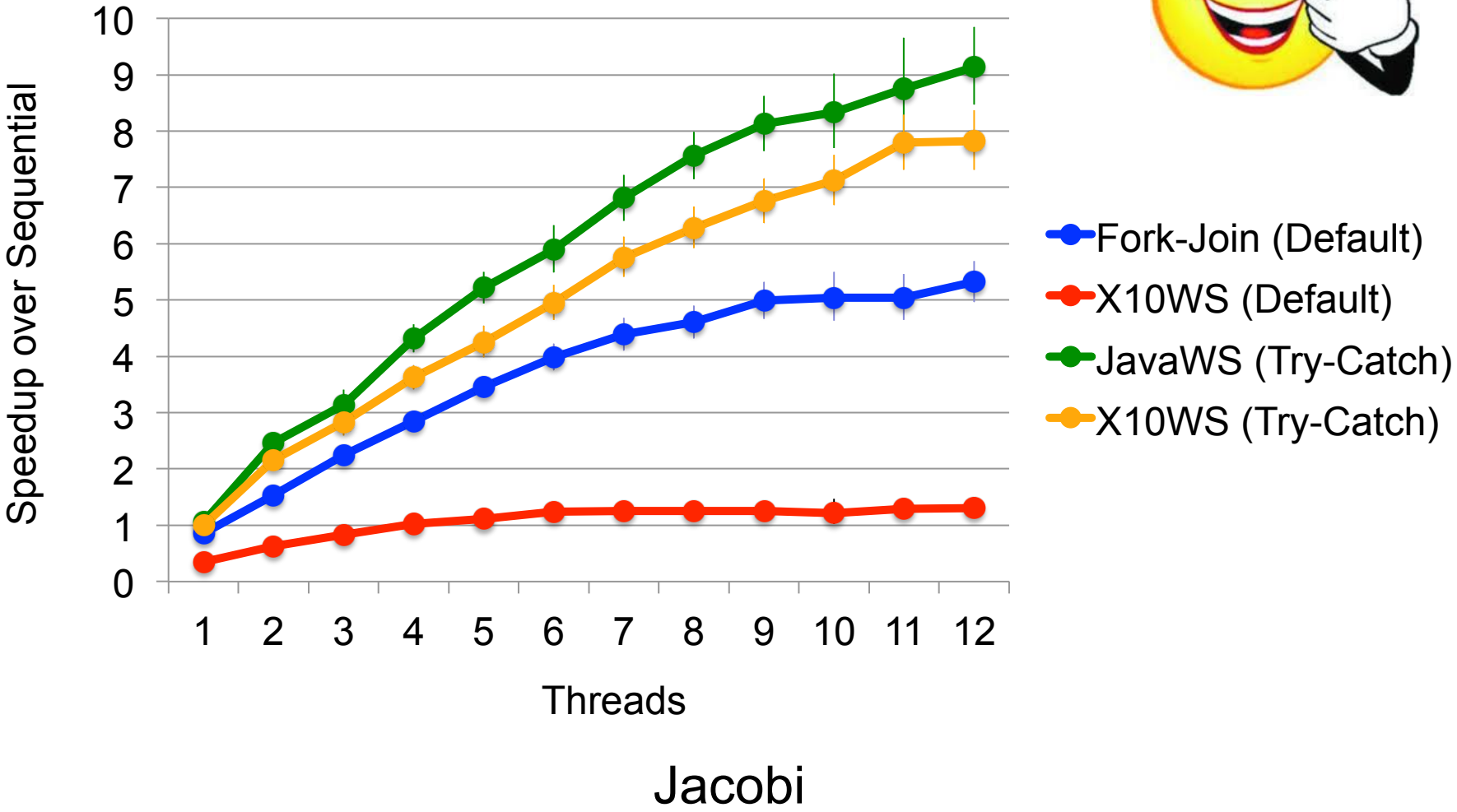
Sequential Overhead



Sequential Overhead

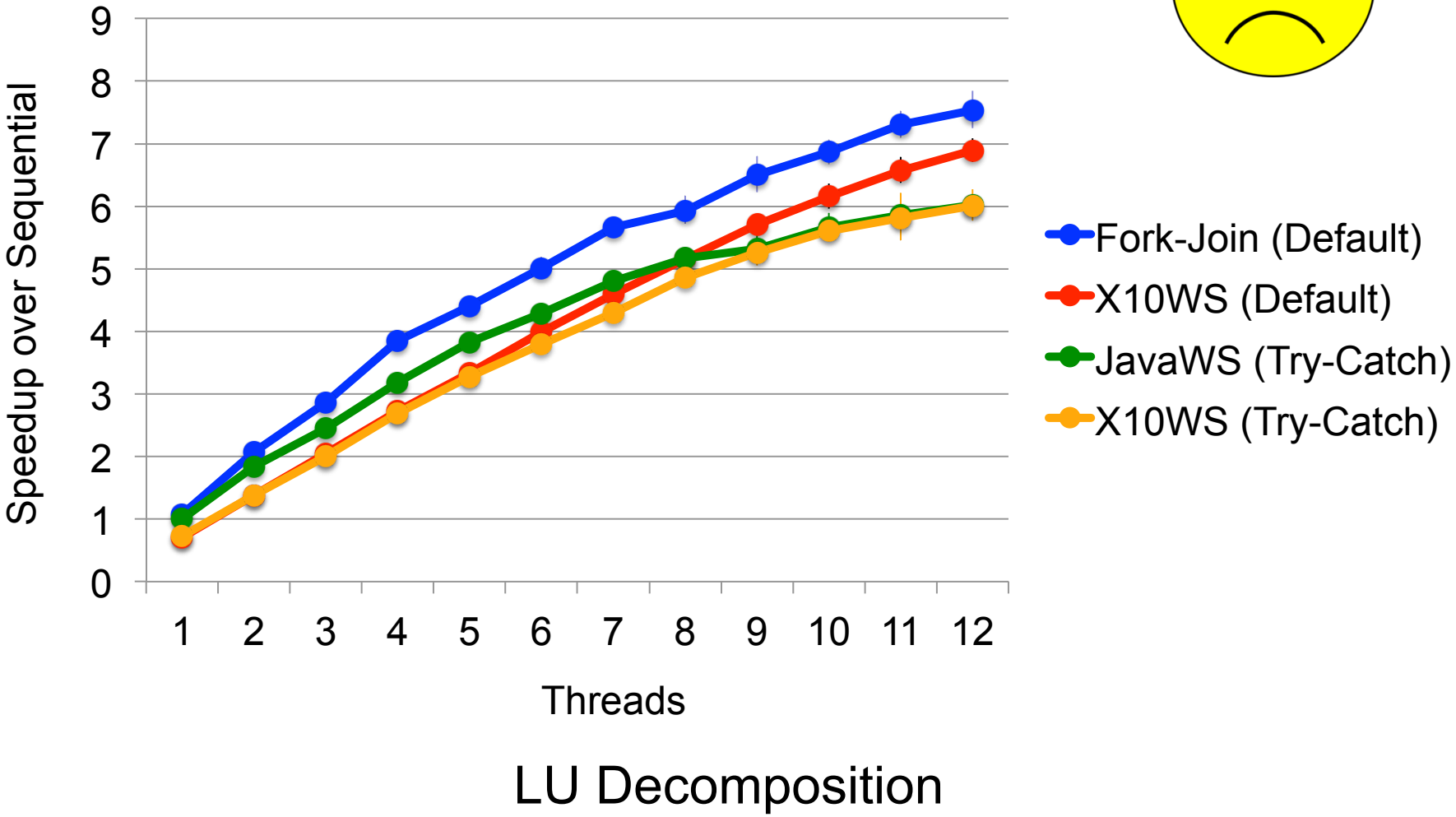


Work–Stealing Performance



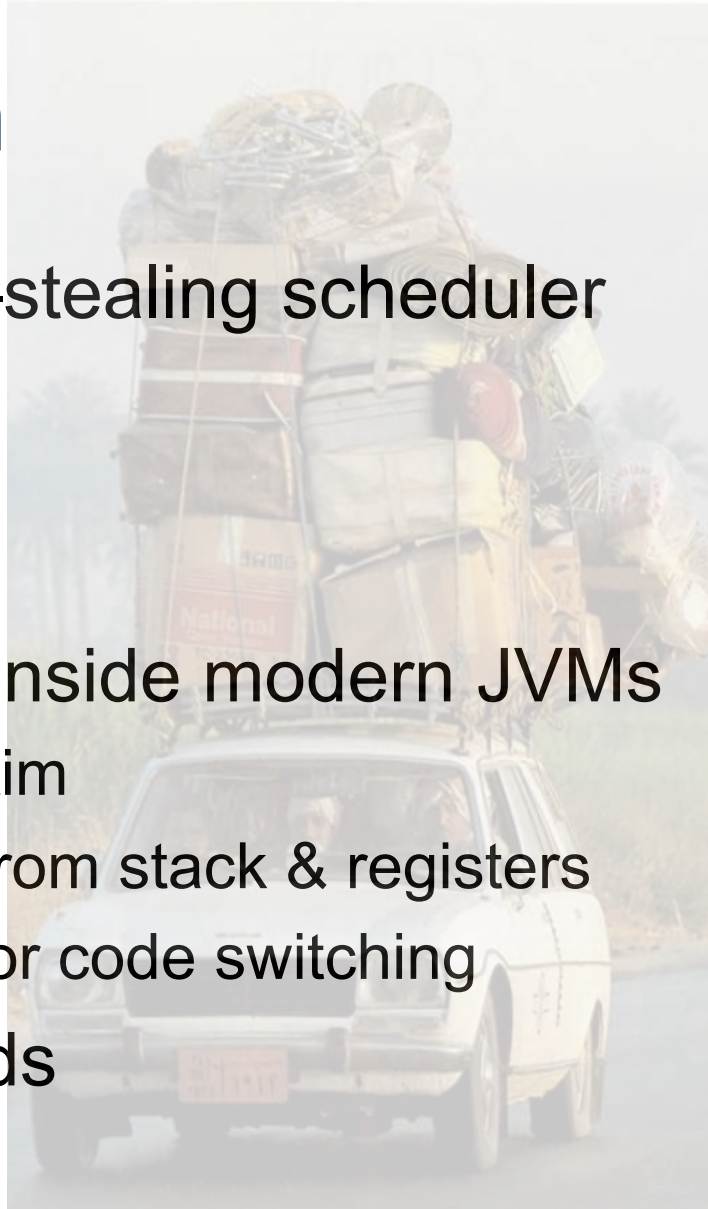
Jacobi

Work–Stealing Performance



LU Decomposition

Summary and Conclusion



- Sequential overheads of work–stealing scheduler
  - Initiation
  - State management
  - Termination
- Reusing existing mechanisms inside modern JVMs
  - **Initiation:** Execution stack of victim
  - **State management:** Extracting from stack & registers
  - **Termination:** Try–catch blocks for code switching
- Eliminated sequential overheads
  - From **400%** to **12%**