Friendly Barriers: Efficient Work-Stealing With Return Barriers

Vivek Kumar[1], Stephen M Blackburn[1], David Grove[2]

1 The Australian National University

2 IBM T.J. Watson Research

The "New" Era of Computing

- Commodity multi-core processors
  - HPC ➔ servers ➔ laptops ➔ mobile devices
- Software parallelism no longer optional
- Wide adoption of managed languages

Research Opportunities Abound ☺

Our Research Question

How can we apply the

capabilities of managed language runtimes

to enable applications with task-based parallelism

to effectively exploit current and future hardware?

Talk Outline

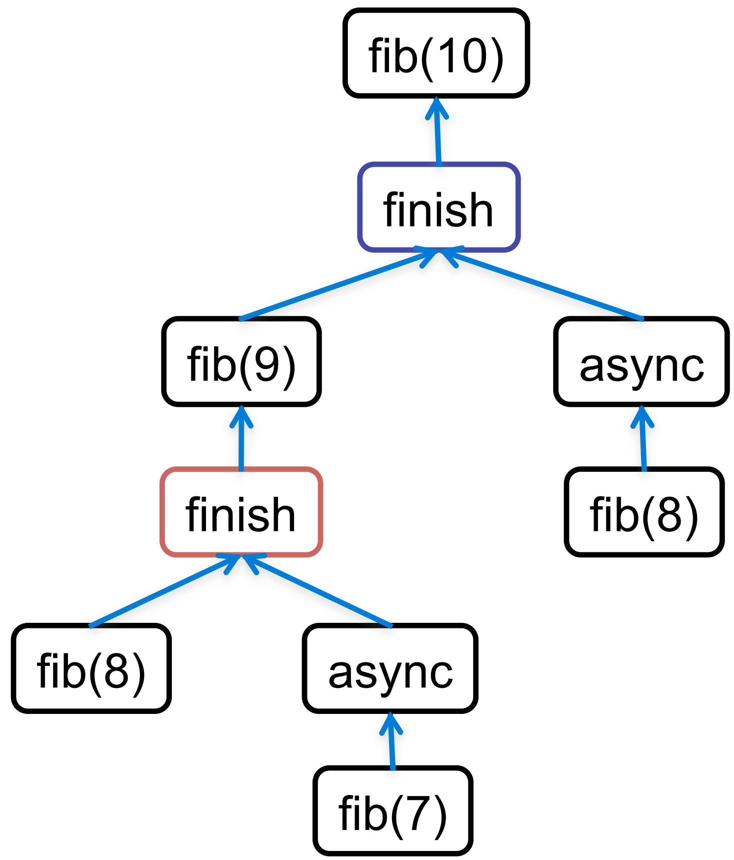
- Background on X10 and Work-Stealing
- Our Base System
  - Try-Catch Work-Stealing [OOPSLA 2012]
- Friendly Barriers [VEE 2014]
  - Motivating analysis
  - How we apply return barriers
  - Performance results
- Conclusions

X10 Summary

- X10 is
  - a programming language
  - an open-source tool chain
    - compiles X10 to C++ or Java
- X10 tackles programming at *scale*
  - scale out: run across many distributed nodes
  - scale up: exploit multi-core and accelerators
  - double goal: *productivity* and *performance*

Task Parallelism in X10

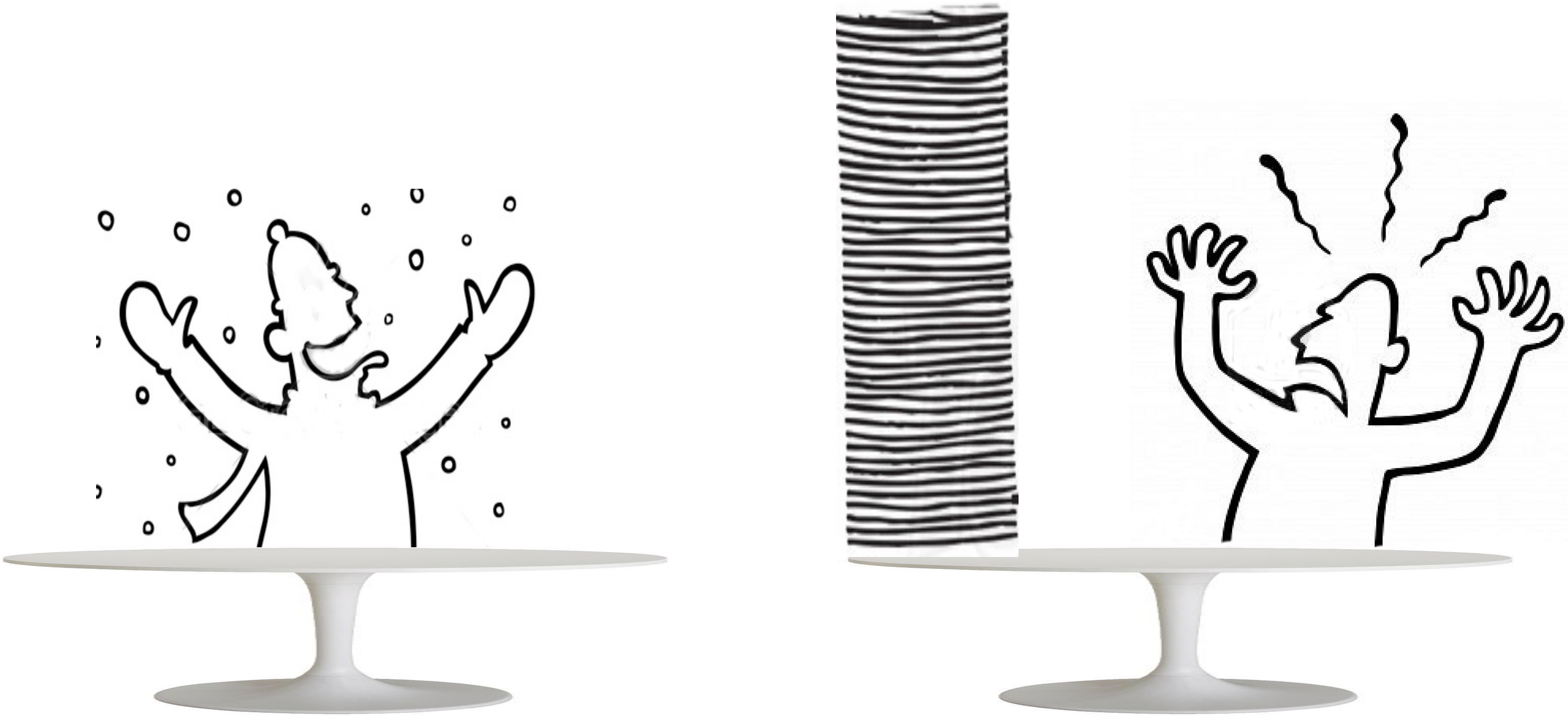
```
static def fib(n:Long):Long {
    val t1, t2:Long;
    if (n < 2) return 1;
    finish {
        async t1 = fib(n-1);
        t2 = fib(n-2);
    }
    return t1 + t2;
}
```



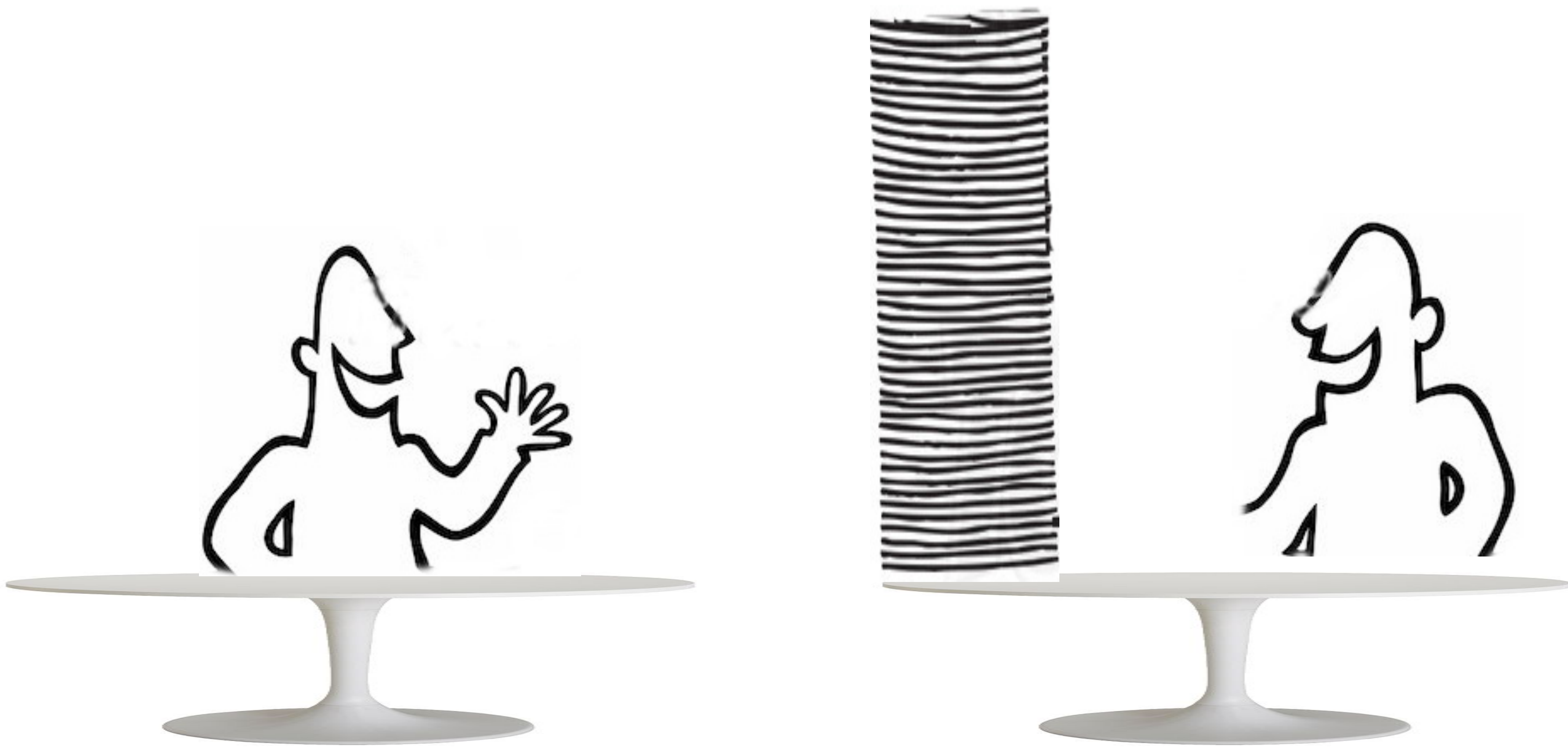
Understanding Work–Stealing

Work–Stealing

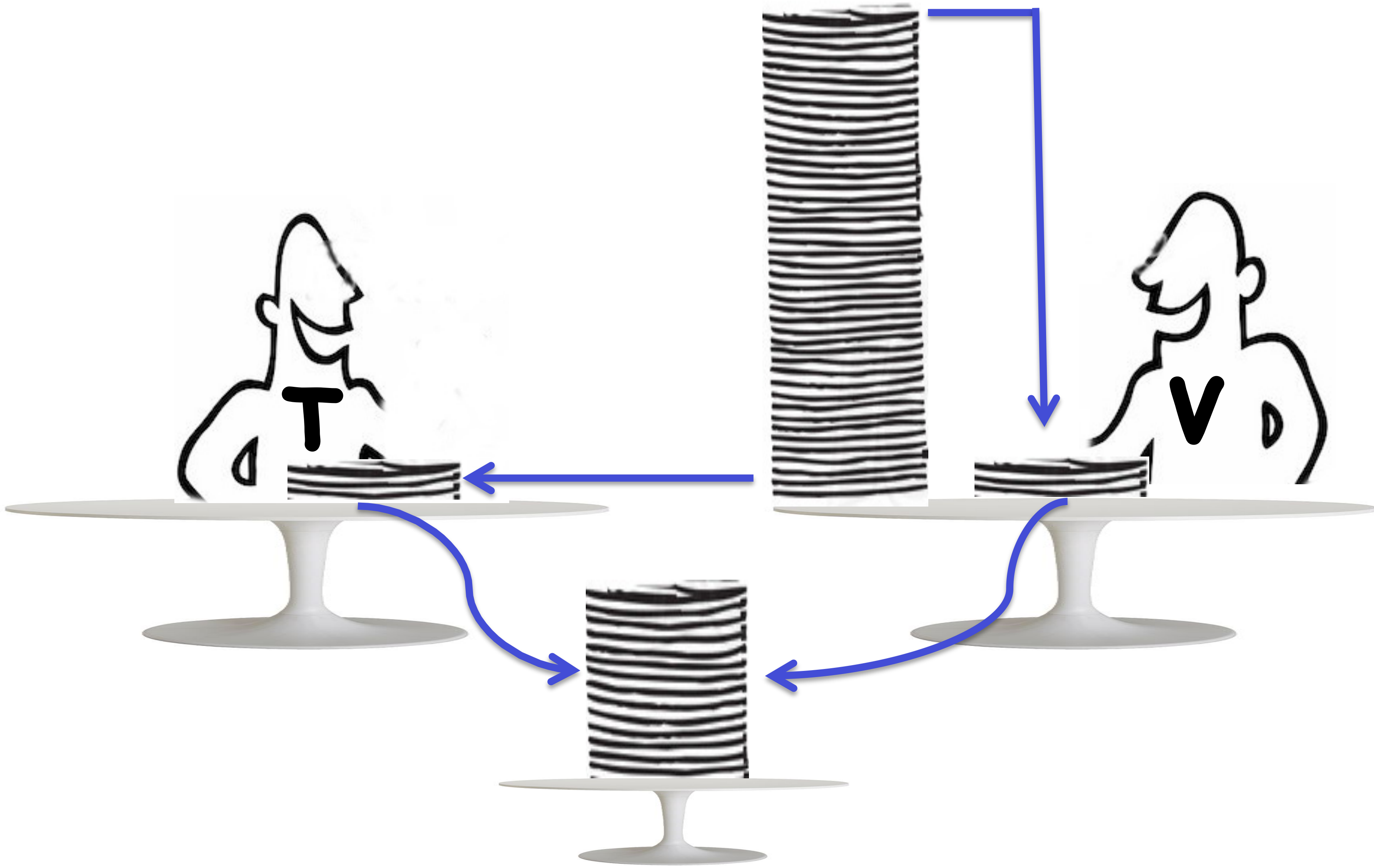
Work–Stealing



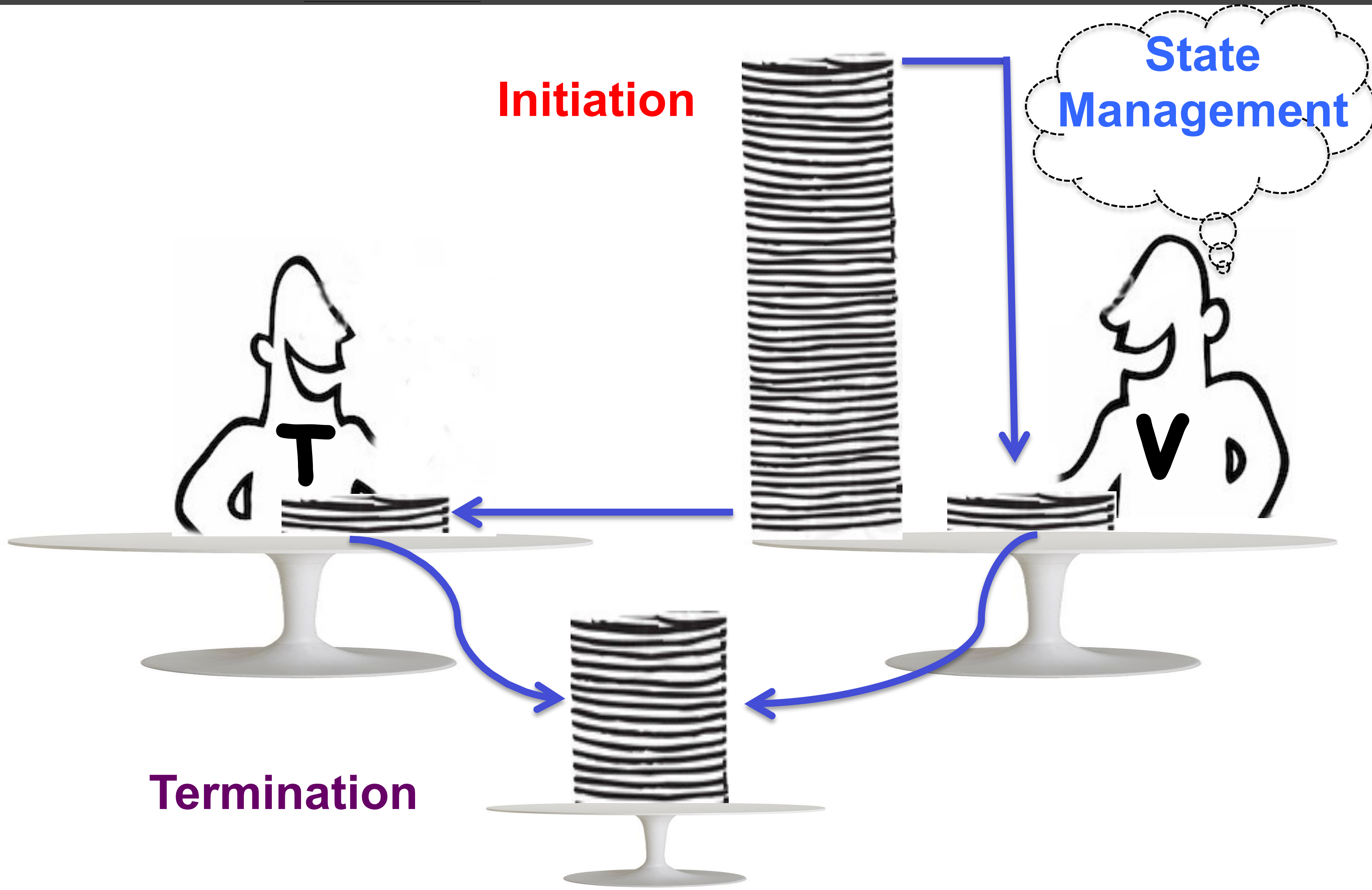
Work–Stealing



Work–Stealing



Work–Stealing



Work-Stealing Schedulers

- Common features
  - a pool of worker threads
  - per-worker deque of pending tasks
  - worker pushes and pops tasks from its deque
  - idle worker steals tasks from another worker's deque
- Widely used
  - Cilk, Java Fork/Join, TBB, X10, Habenero, …

Work-Stealing Without the Baggage

OOPSLA 2012

- JavaWS (Try-Catch)
  - Reduced sequential overheads of work-stealing from 4.1x to 15%
  - Our baseline system
    - DefaultWS

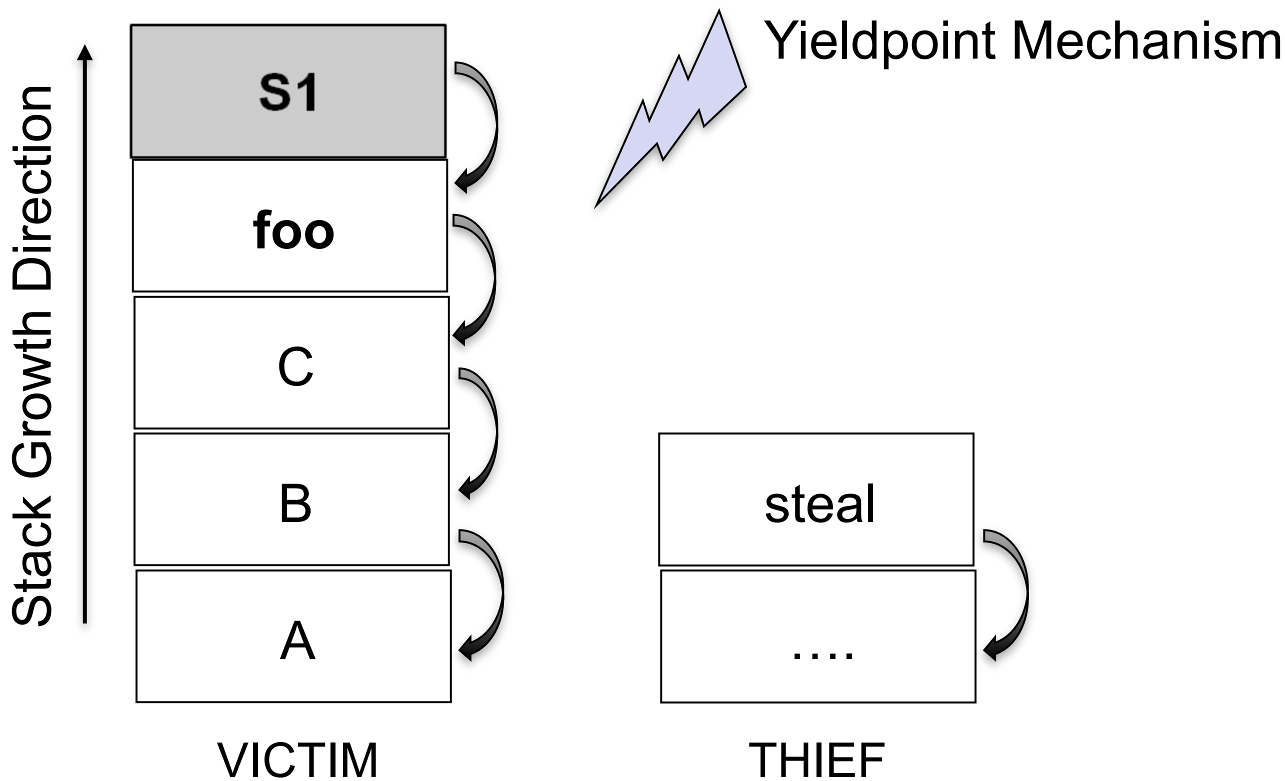
Australian National University

IBM

Our Prior Work

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

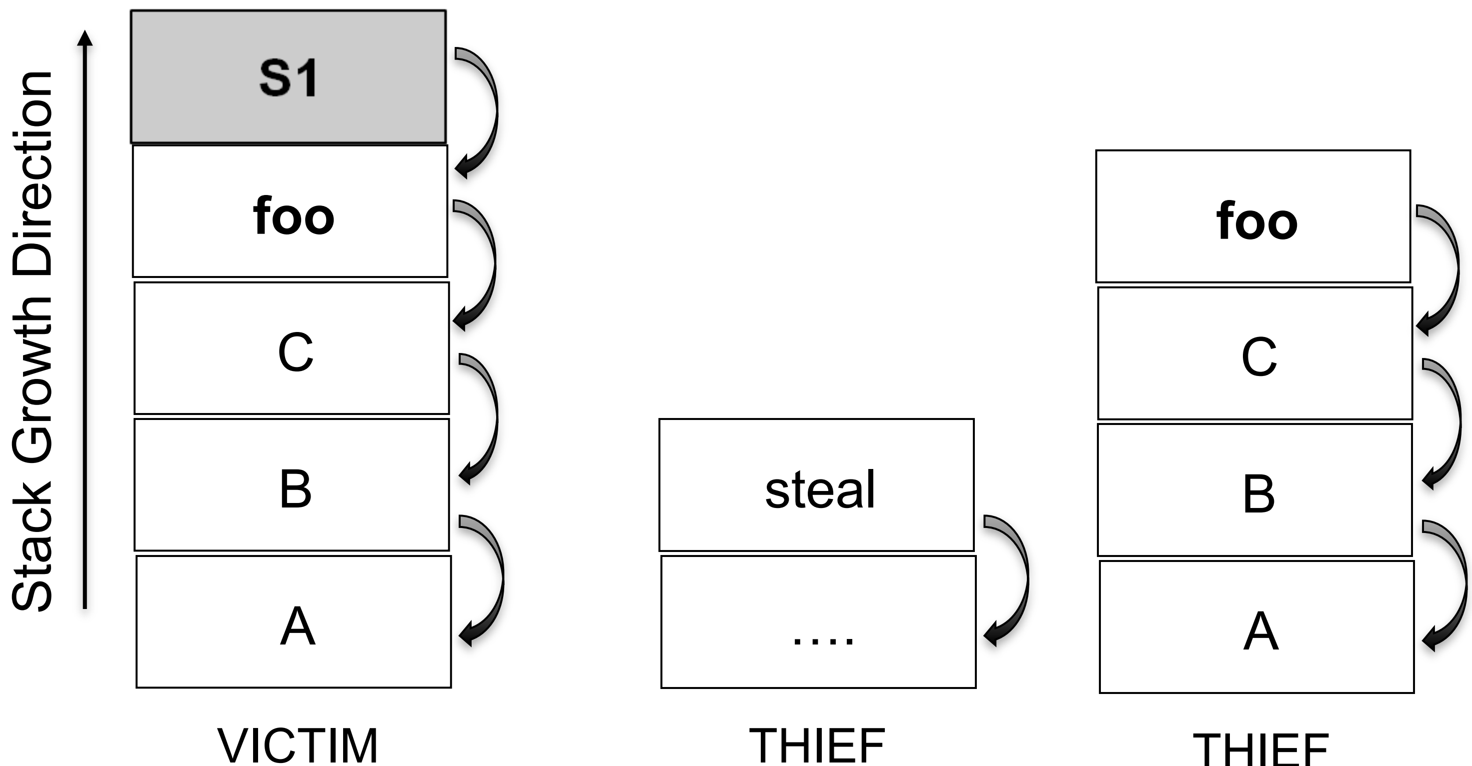
- Yieldpoint mechanism
- On-stack replacement
- Java try/catch exceptions
- Dynamic code patching



Our Prior Work

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

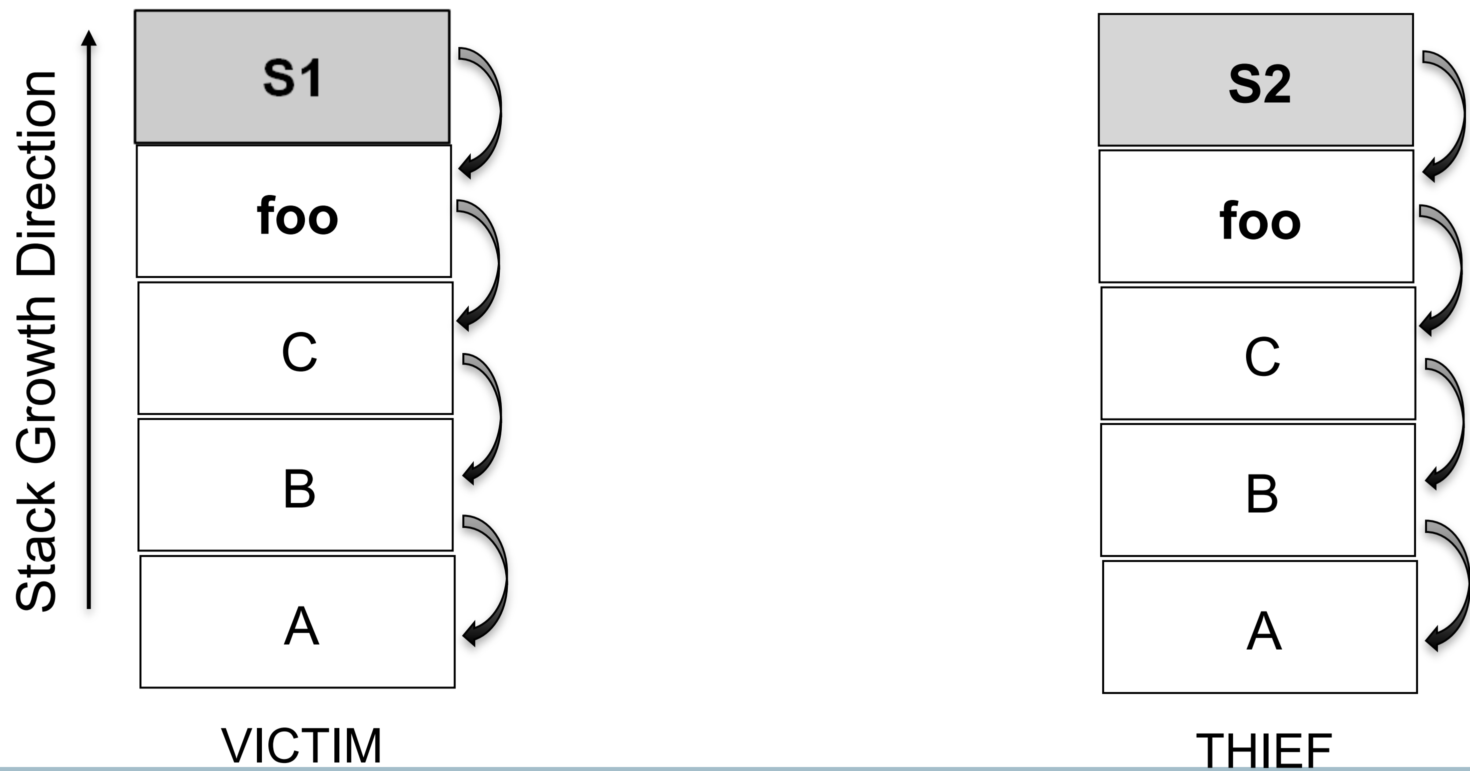
- Yieldpoint mechanism
- On-stack replacement
- Java try/catch exceptions
- Dynamic code patching



Our Prior Work

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

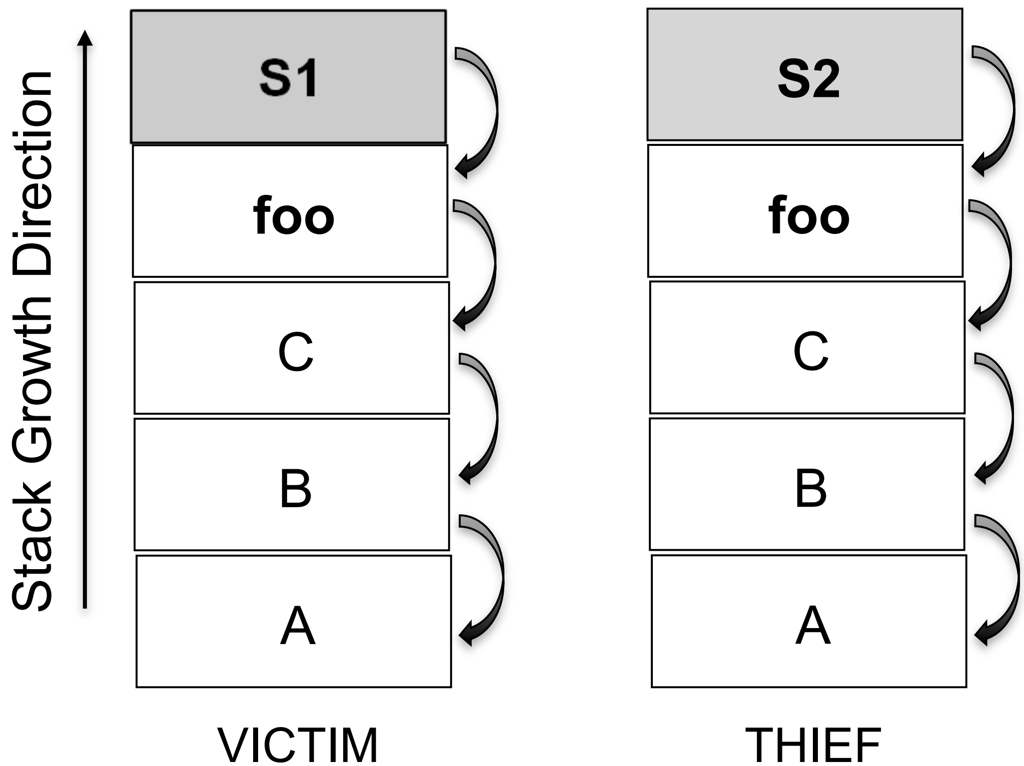
- Yieldpoint mechanism
- On-stack replacement
- Java try/catch exceptions
- Dynamic code patching



Our Prior Work

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

- Yieldpoint mechanism
- On-stack replacement
- Java try/catch exceptions
- Dynamic code patching

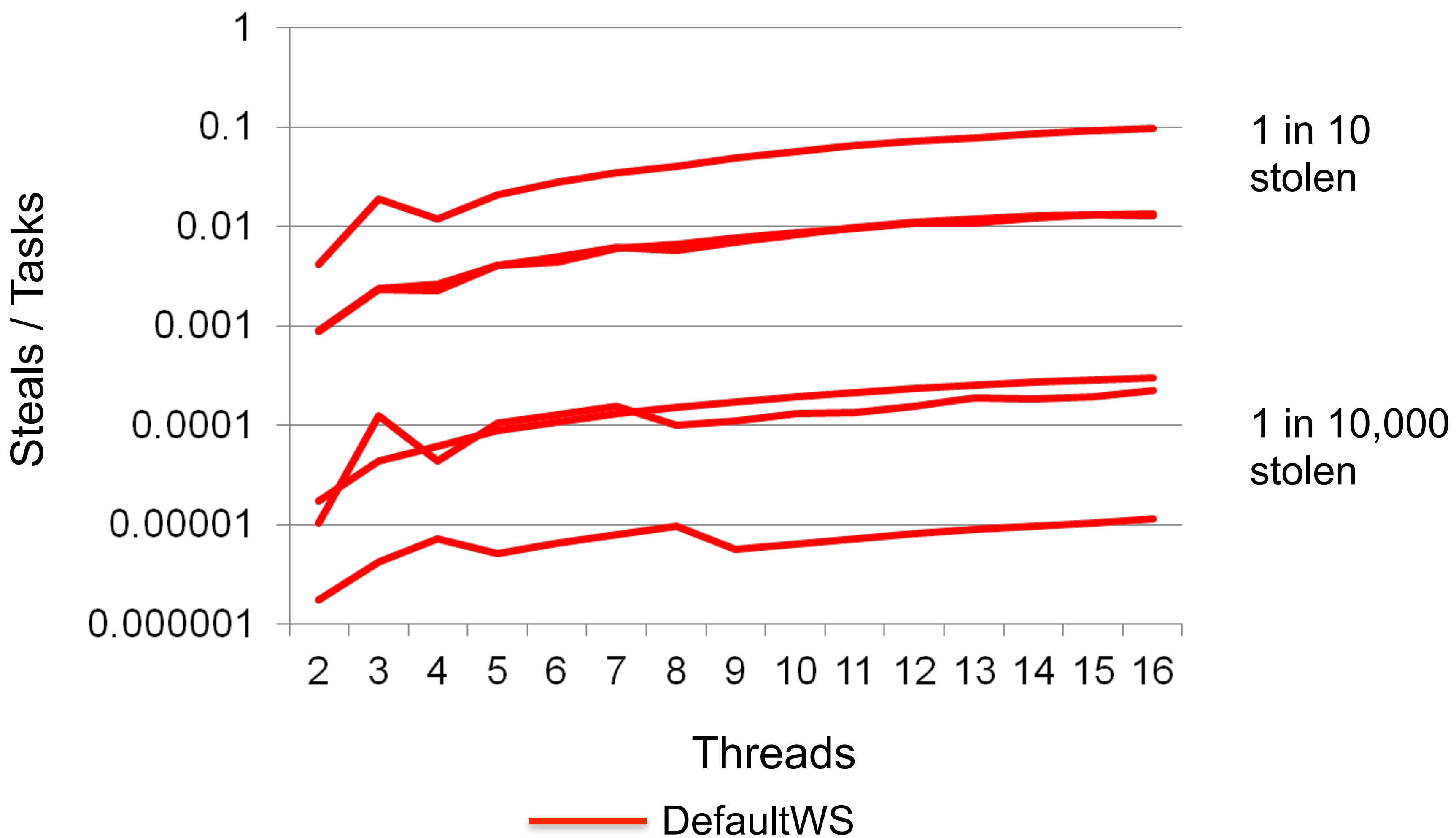


Motivating Analysis

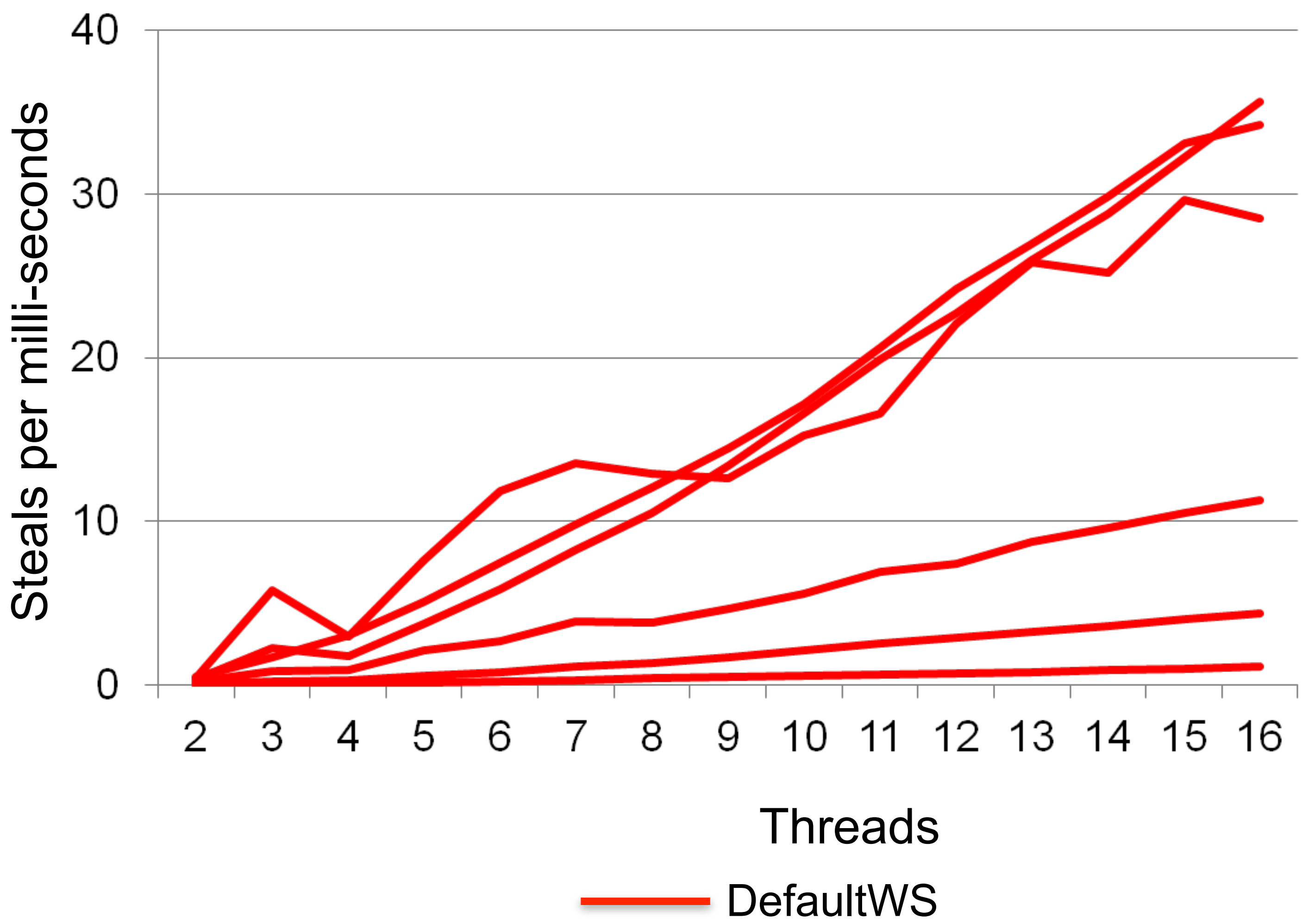
Methodology

- Benchmarks
  - Jacobi
  - FFT
  - CilkSort
  - Barnes-Hut
  - UTS
  - LU Decomposition (LUD)
- Hardware platform
  - 2 Intel Xeon E5-2450
    - 8 cores each
- Software platform
  - Jikes RVM (3.1.3)

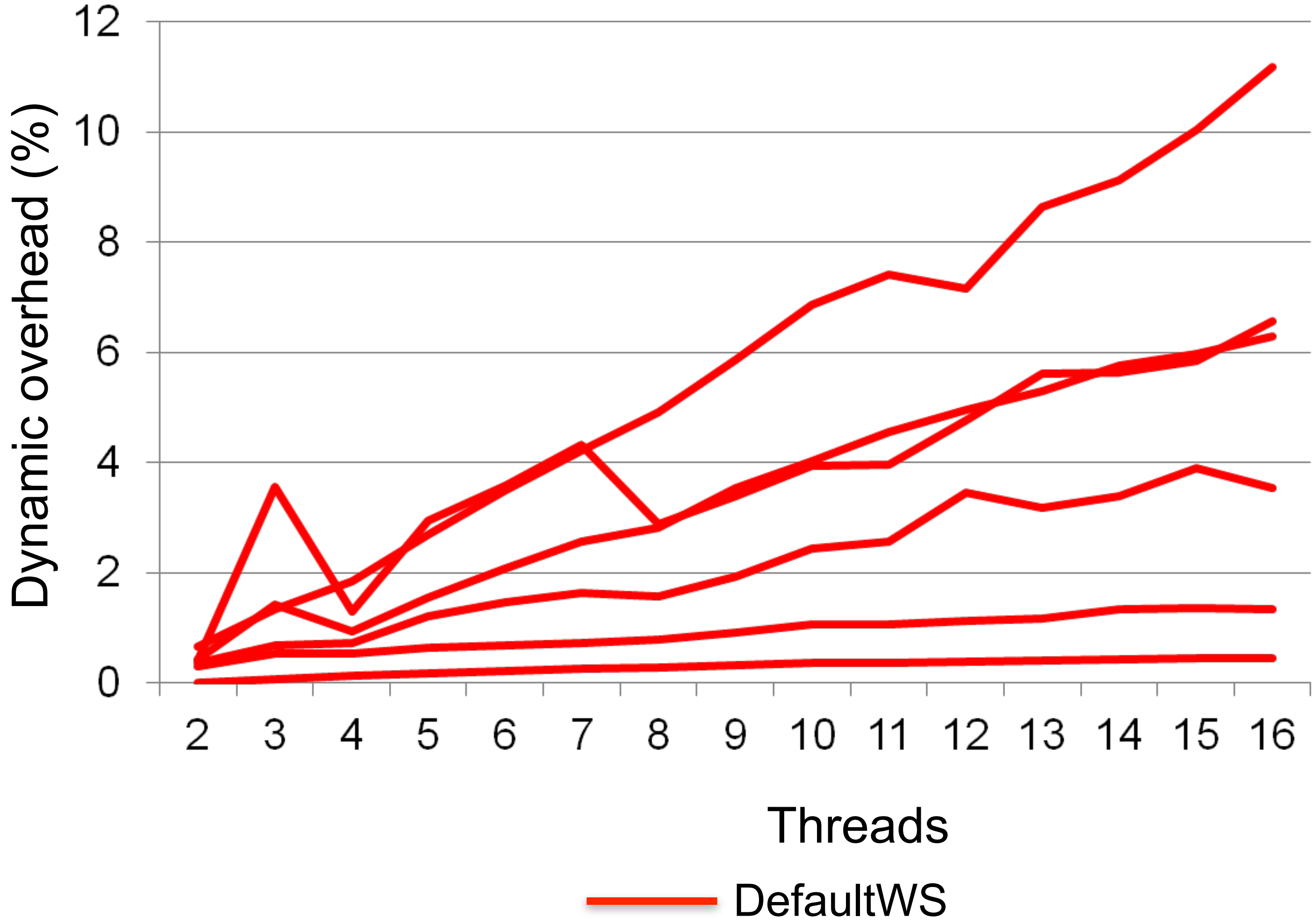
Steals To Task Ratio



Steal Rate


40
30
20
10
0
Steals per milli-seconds
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Threads
DefaultWS

Dynamic Overhead (Victim Stalled)



Insights

- Forcing victim to wait inside yieldpoint at every steal attempt is inefficient
- Re-use existing mechanisms inside modern managed runtime to reduce victim wait time

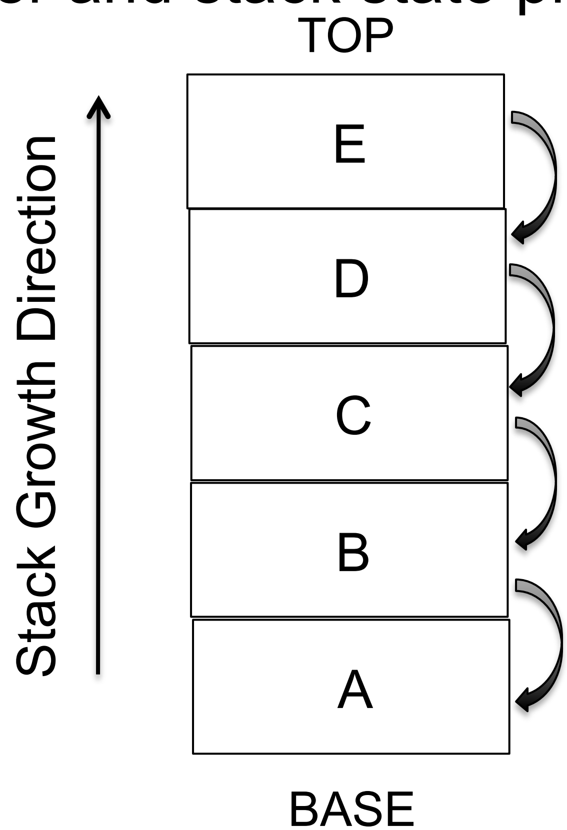
Approach

- Use return barrier to “protect” the victim from thief
  - ✓ Victim oblivious to steal from thief
  - ✓ Cost of barrier only when victim unwind past the barrier
  - ✓ When above the barrier, victim sees no cost
  - ✓ More concurrency between thief and its victim

Implementation

Return Barrier

- Allows runtime to intercept a common event
- Hijack a return and bridge to some other method
- Register and stack state preserved

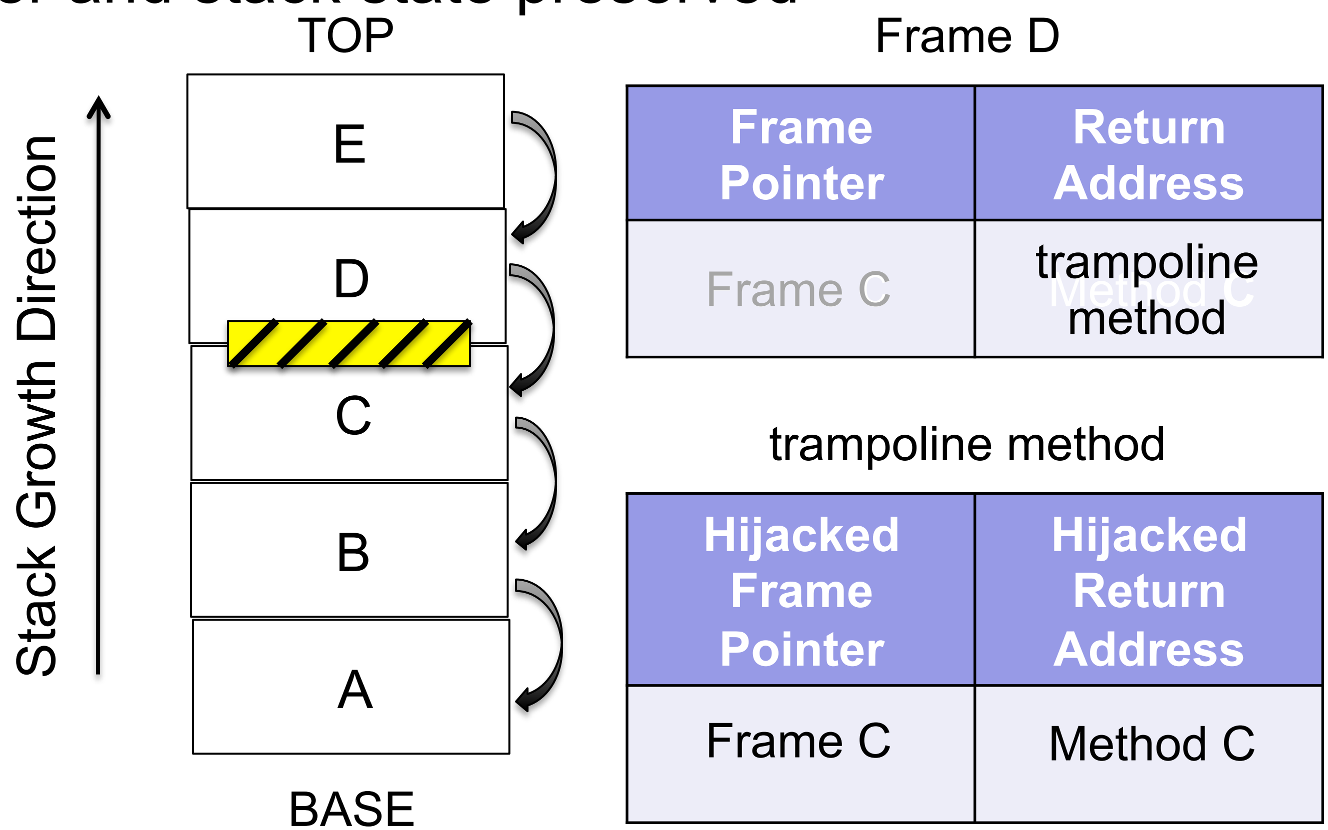


Frame D

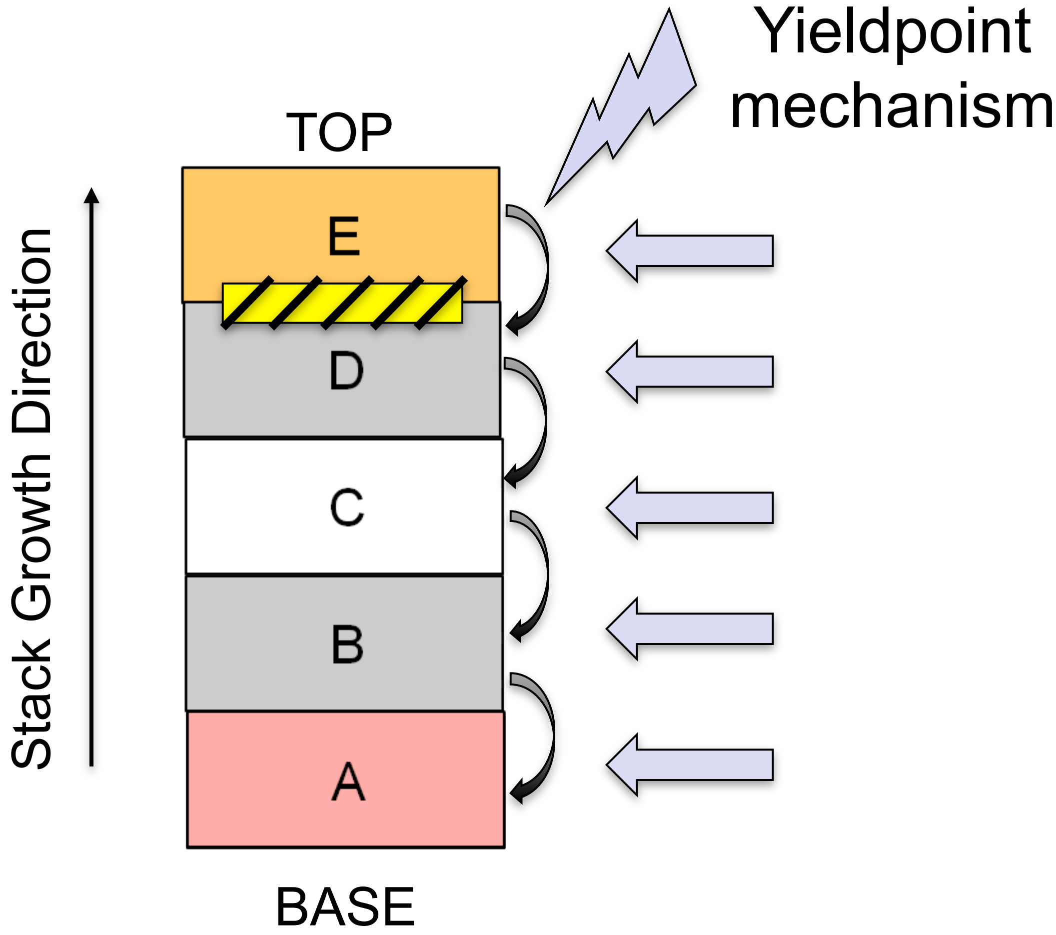
Frame Pointer	Return Address
Frame C	Method C

Return Barrier

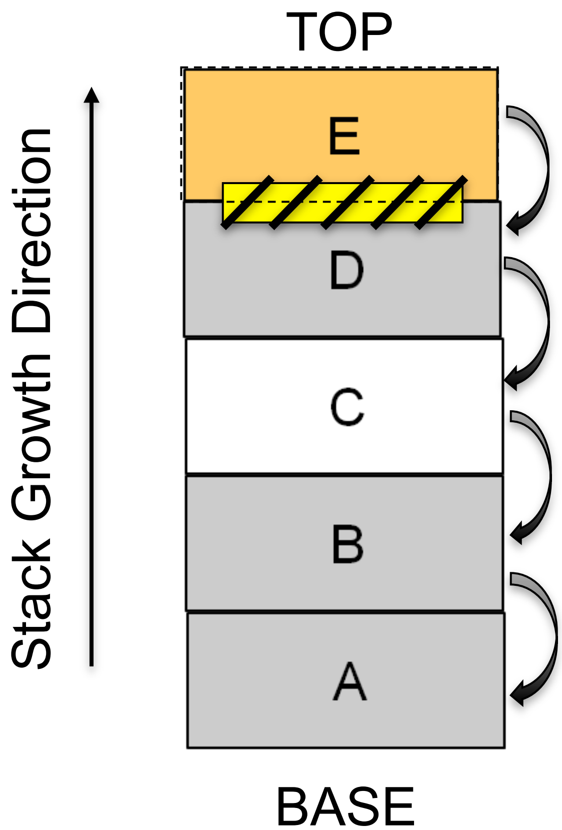
- Allows runtime to intercept a common event
- Hijack a return and bridge to some other method
- Register and stack state preserved



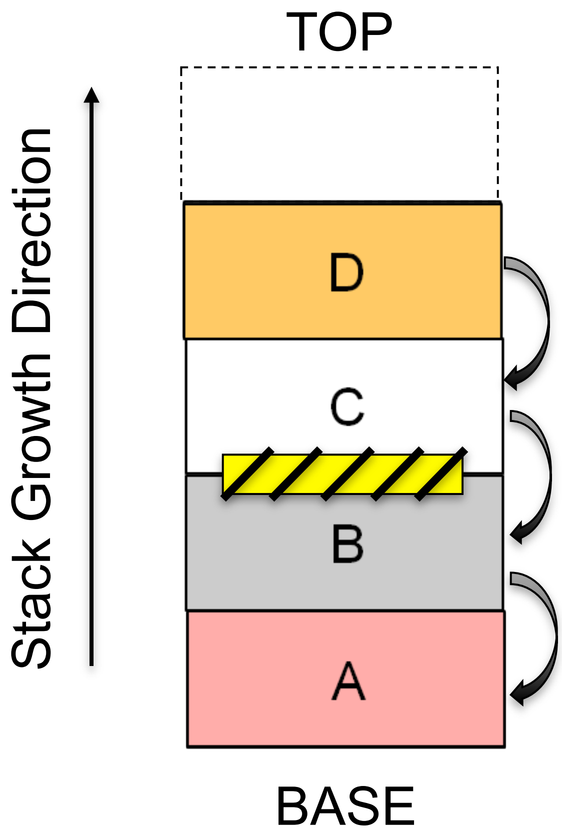
Thief Installs Return Barrier



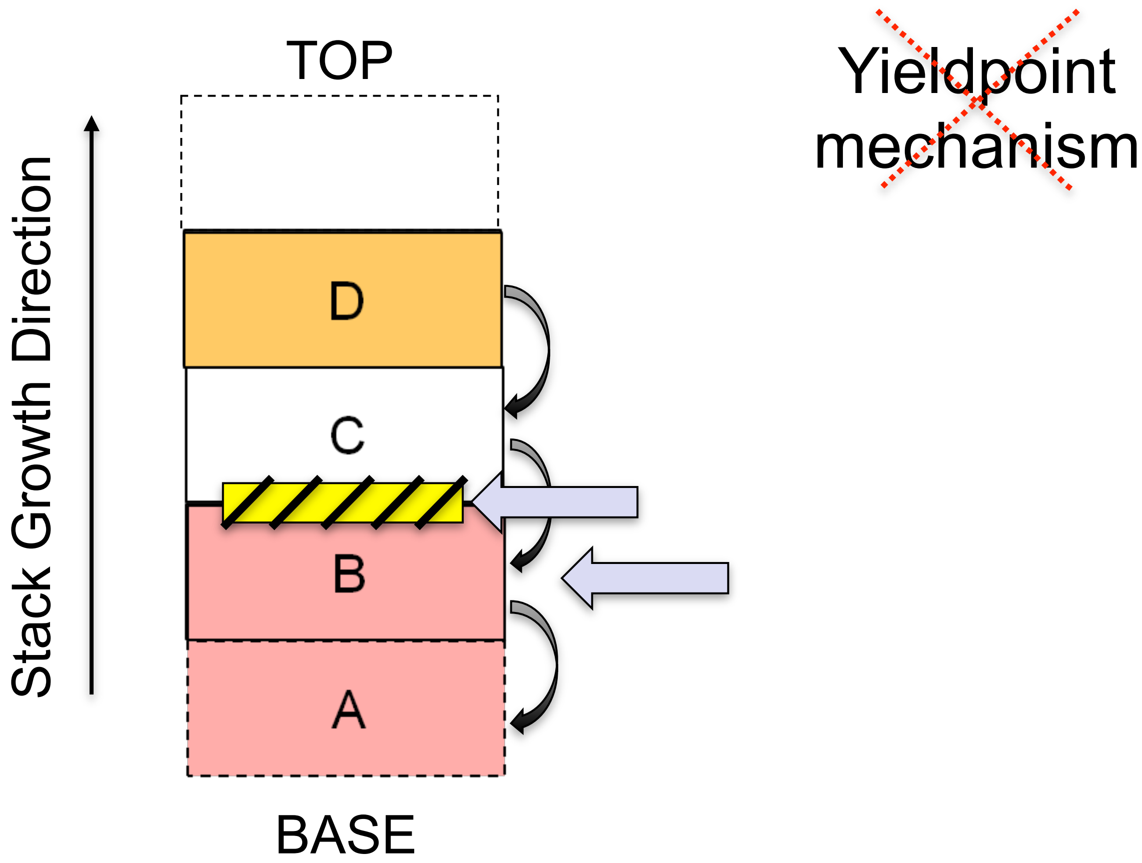
Victim Moves The Return Barrier



Victim Moves The Return Barrier



Robbing A Victim With Return Barrier

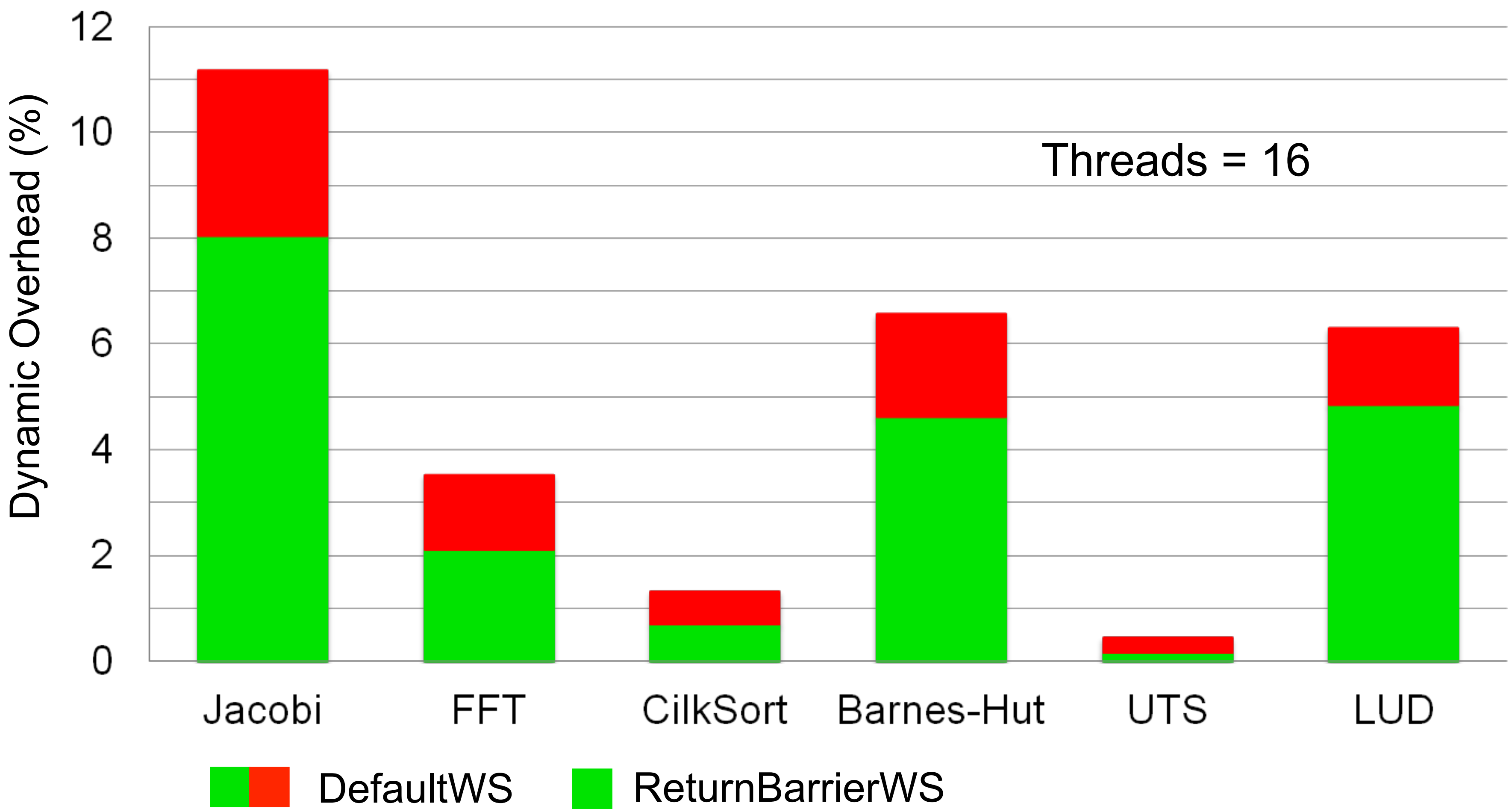


Australian National University

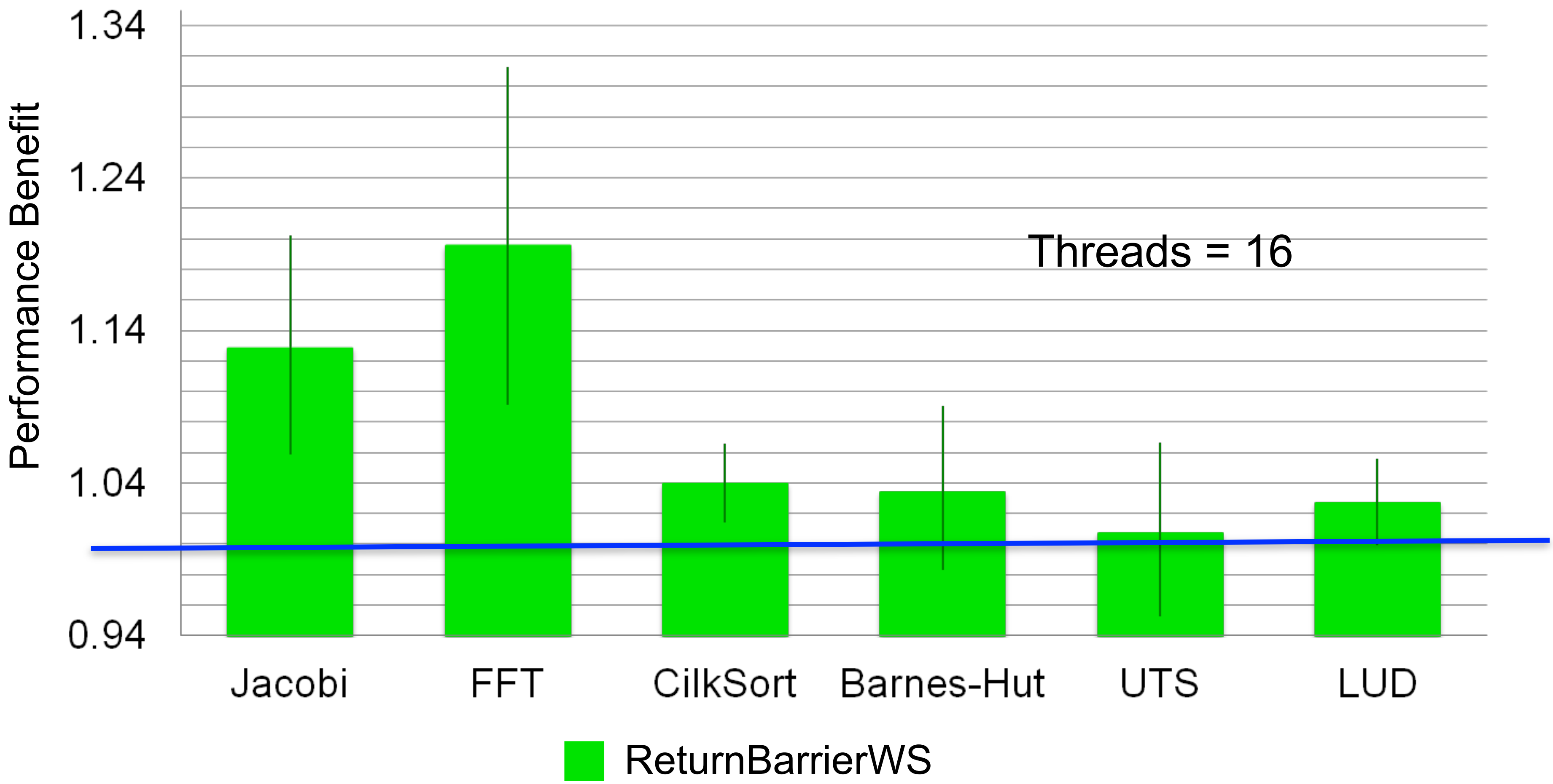
IBM

Performance Evaluation

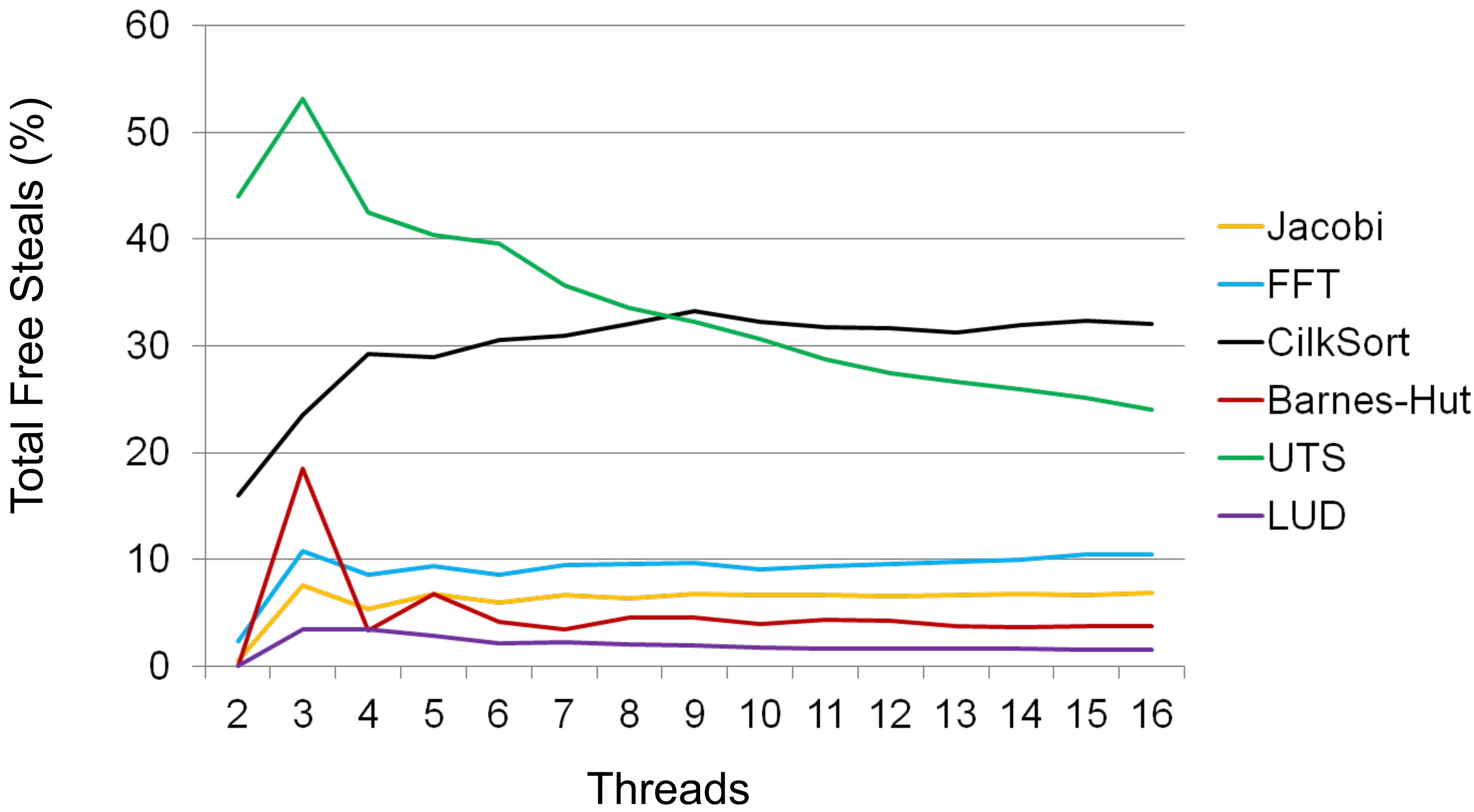
Dynamic Overhead



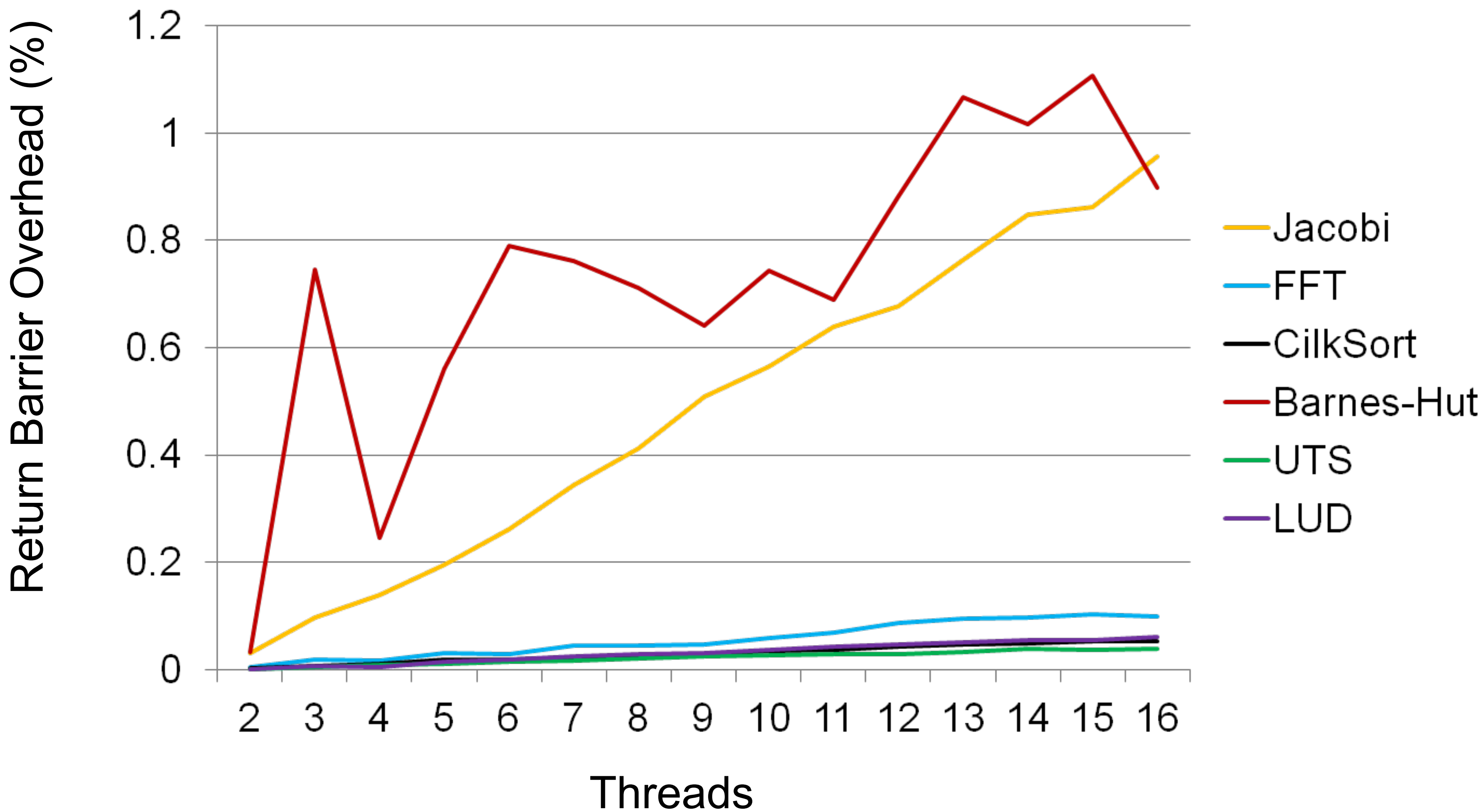
Performance Benefit Relative to DefaultWS



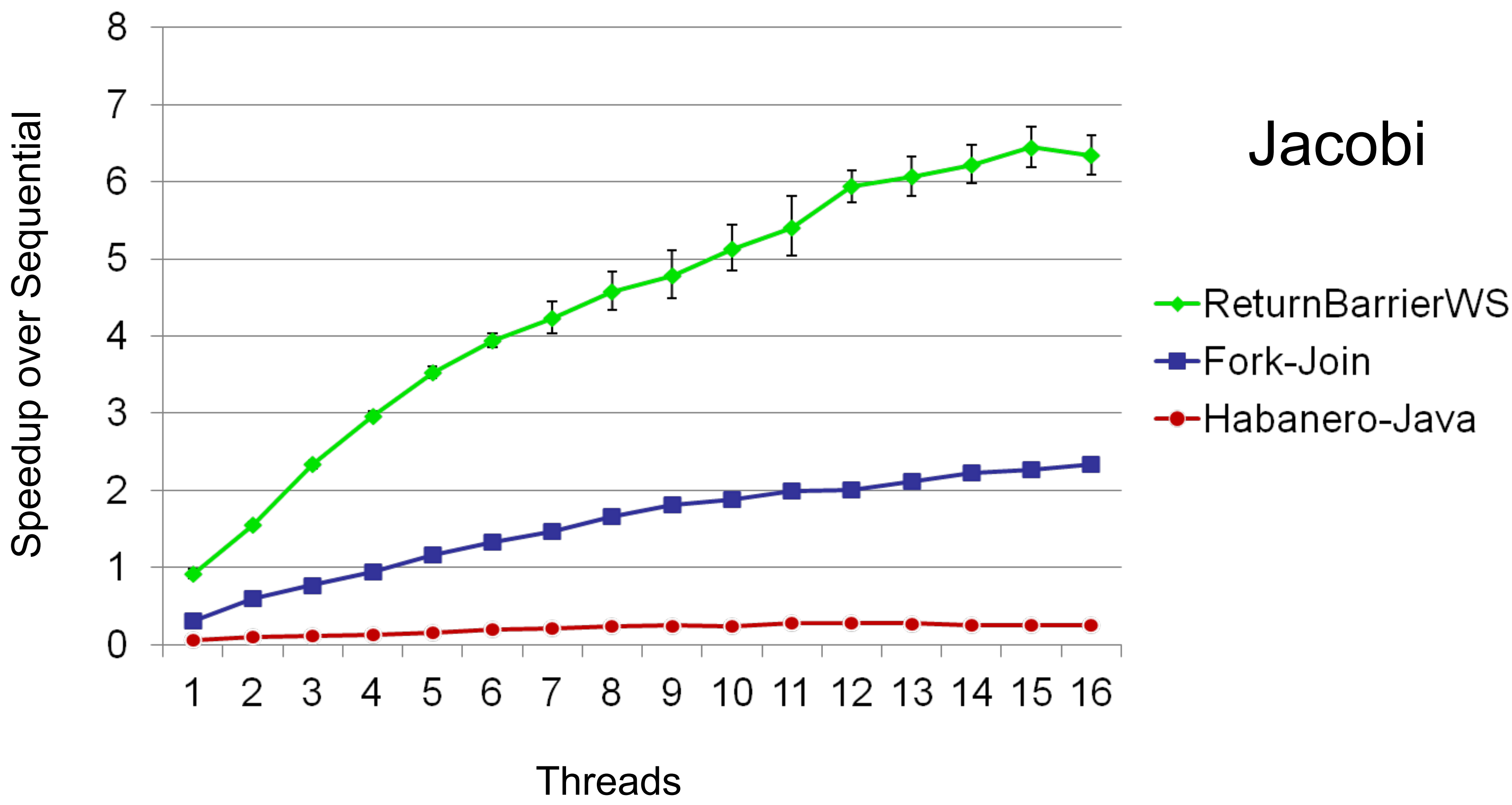
Free Steals From Return Barrier



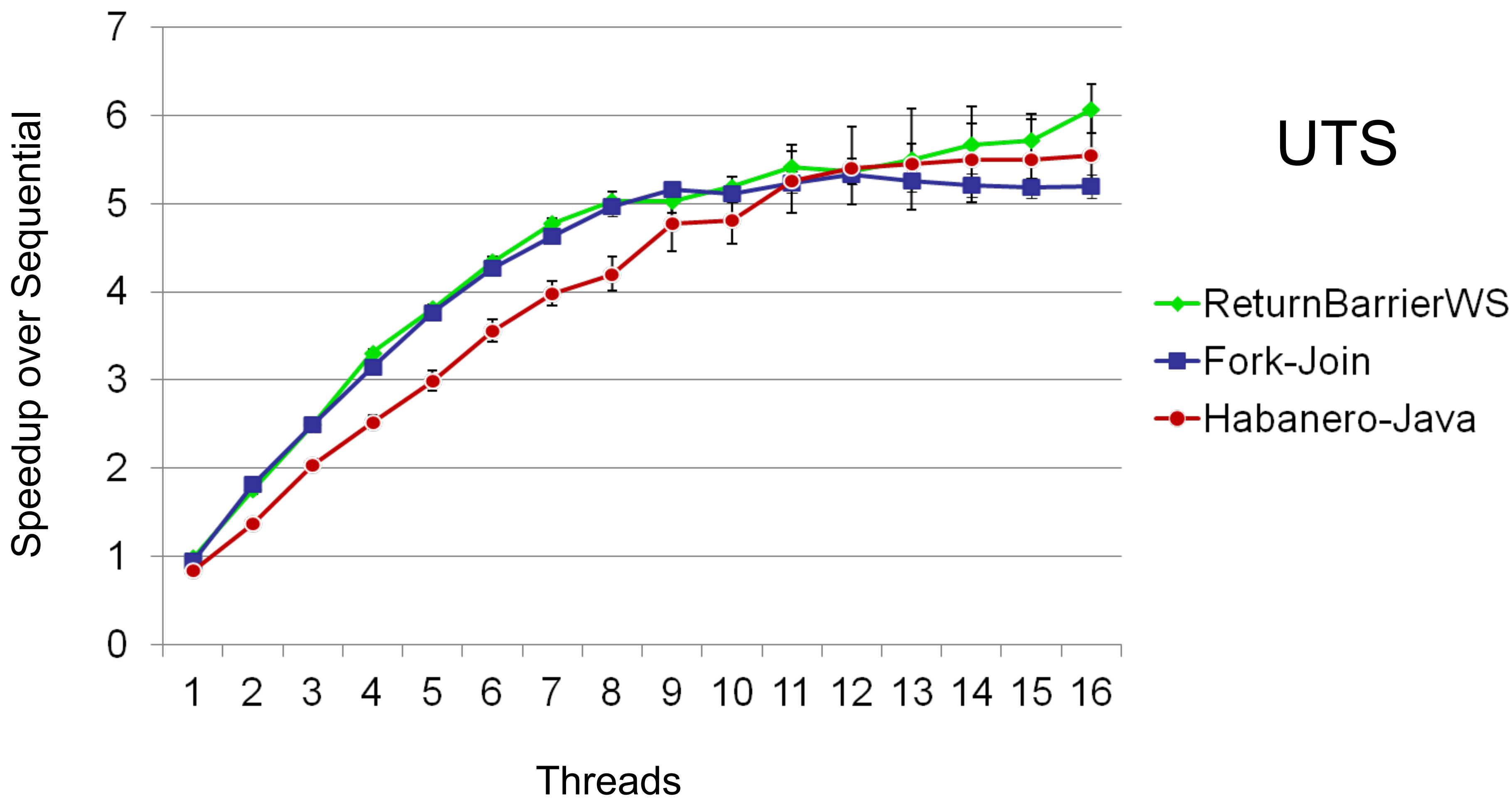
Overhead of Executing Return Barrier



Comparative Performance



Comparative Performance



Summary and Conclusion

- Big Picture: Laziness pays off
  - DefaultWS extremely efficient/effective
- Tackling dynamic overheads
  - grows as parallelism increases
  - grows as steal rate increases
- Return barrier mechanism *protects* victim from thief
  - Victim oblivious to thief's activities
- Return barrier *halves* dynamic overhead
- Performance benefit (vs DefaultWS) of up to 20%