Australian National University

High Performance Runtime for Next Generation Parallel Programming Languages

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

1 The Australian National University

2 IBM T.J. Watson Research

3 Microsoft

Hardware and Software Today

"Hey, Joe. Would you eat another apple? I need to add one more core to this computer."

The Challenge

- Productivity
- Performance
- Portability

Options ?

- Productivity
  - Language based features to expose parallelism – X10, Cilk, Habanero-Java etc
- Performance
  - Work–stealing scheduling
- Portability
  - Managed runtime to hide the hardware complexities

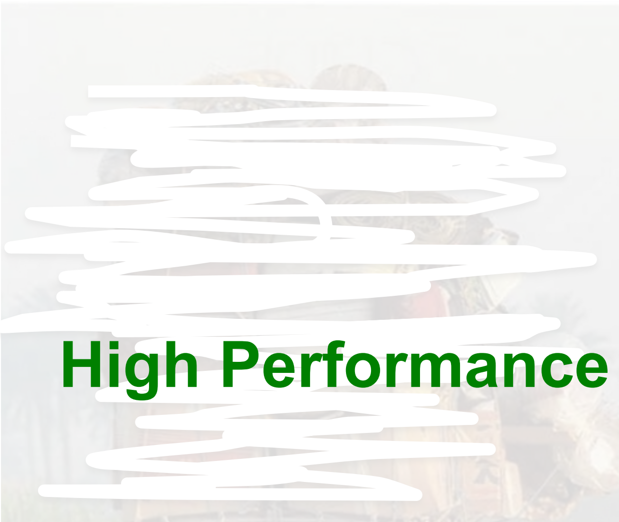
Thesis Statement

High performance languages are using managed platforms for productivity and portability, but performance is inadequate. By exploiting and extending the underlying mechanisms of managed runtimes, implementation of these languages will be able to deliver scalability and performance at the levels necessary for widespread uptake.

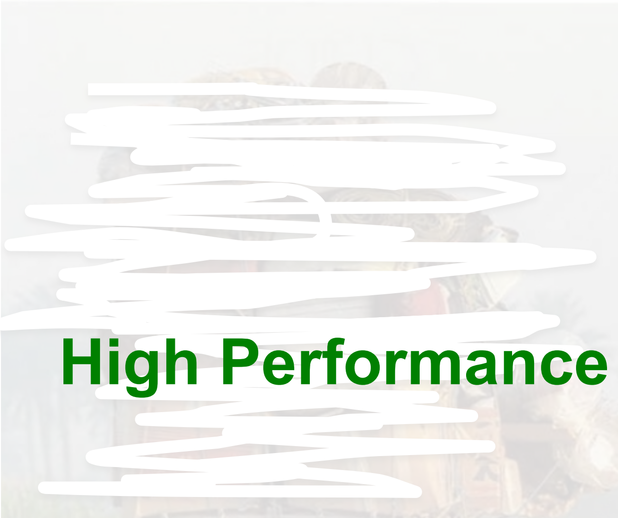
Contributions

Contributions

Contributions

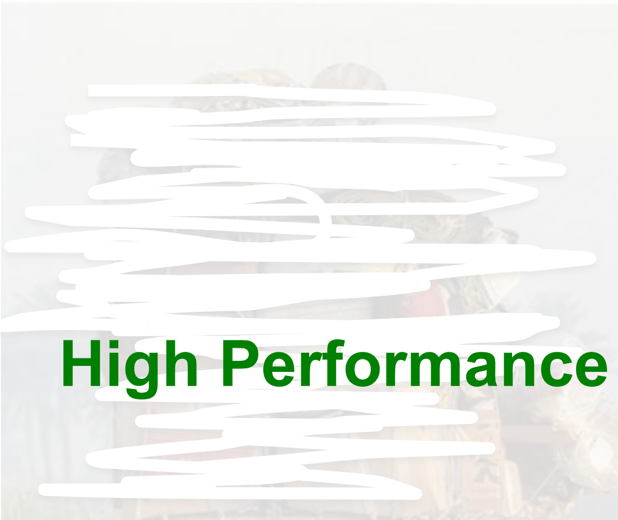


Contributions



High Performance **High Productivity**

Contributions



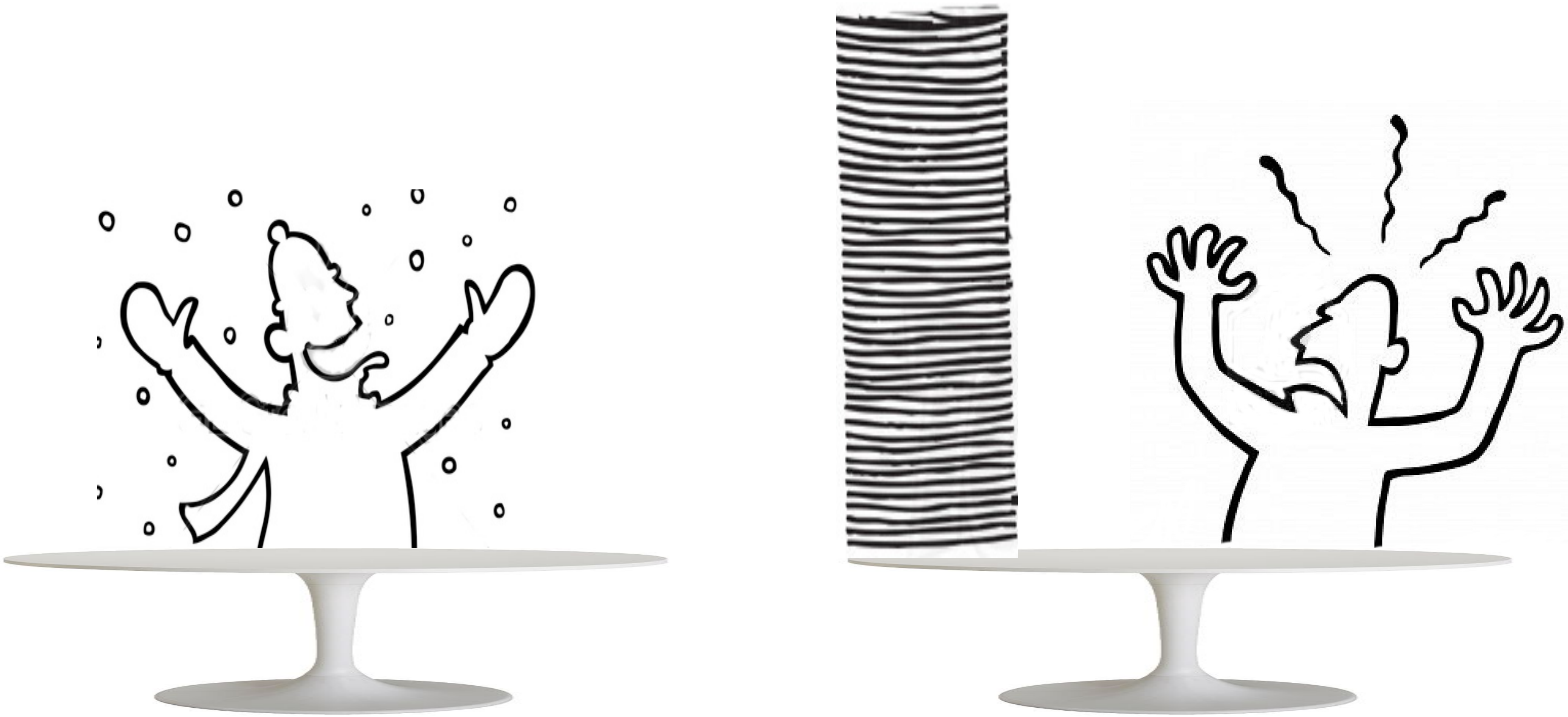
High Performance

High Productivity

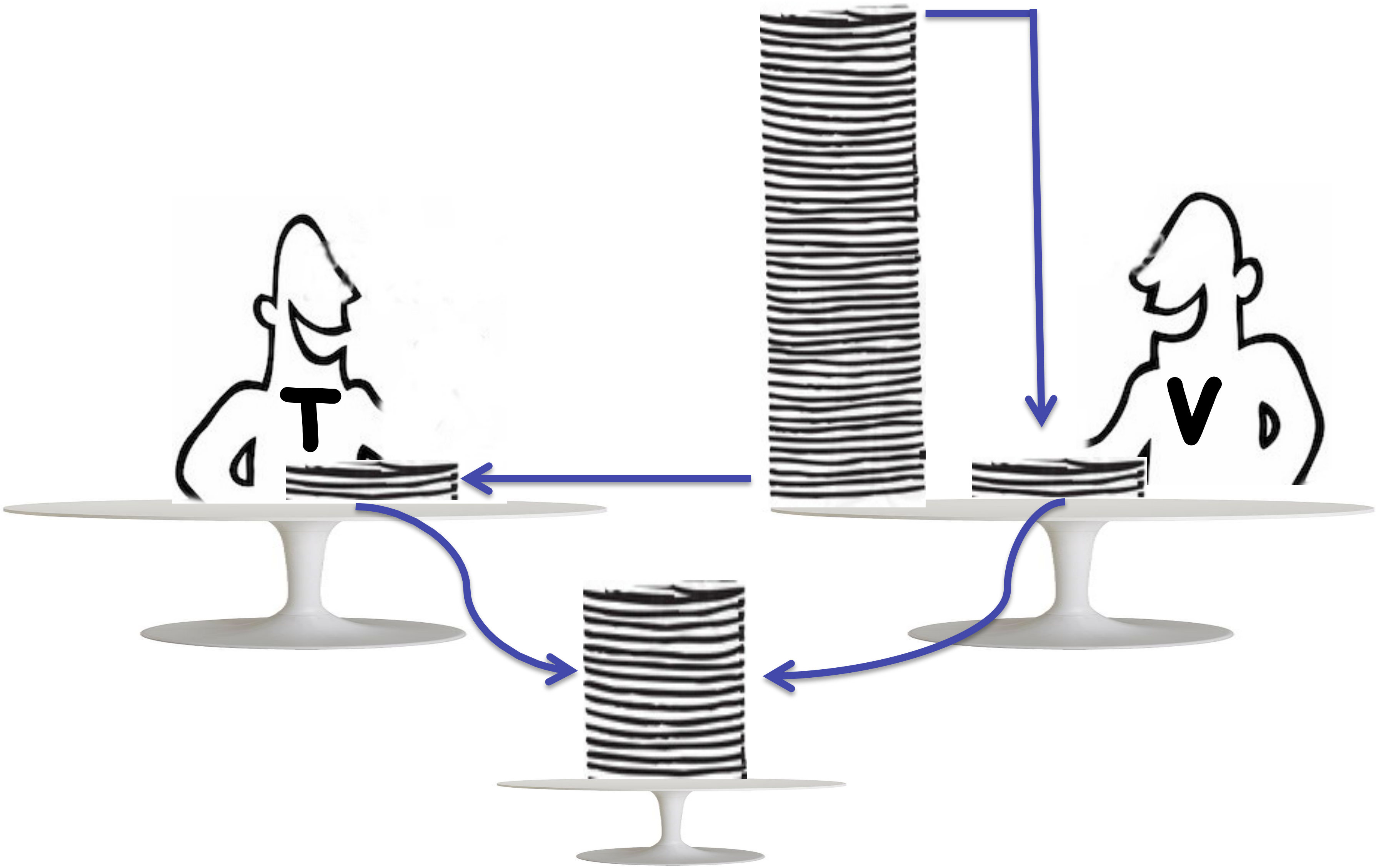
Highly Competitive

Understanding Work–Stealing

Understanding Work–Stealing



Understanding Work–Stealing

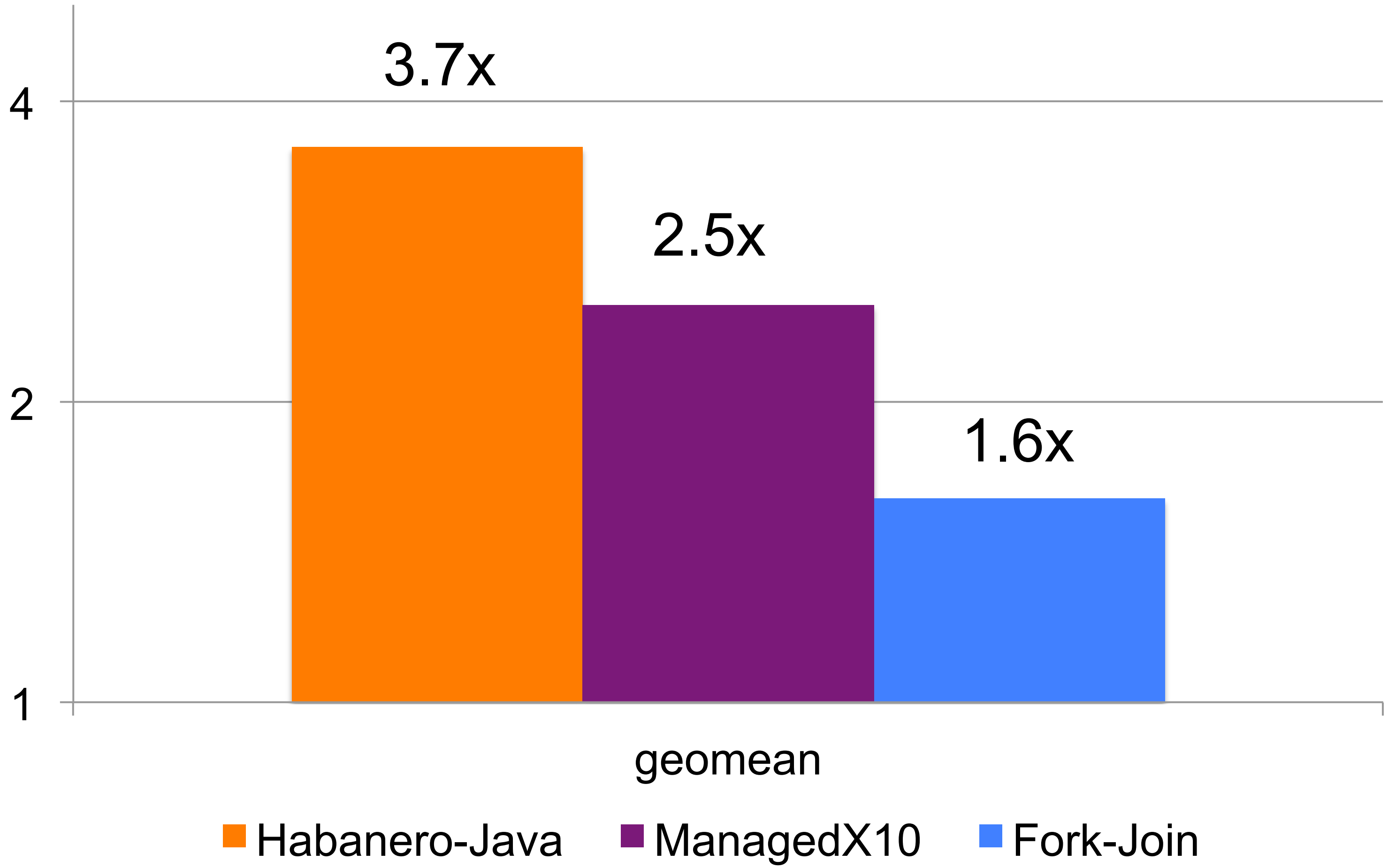


Methodology

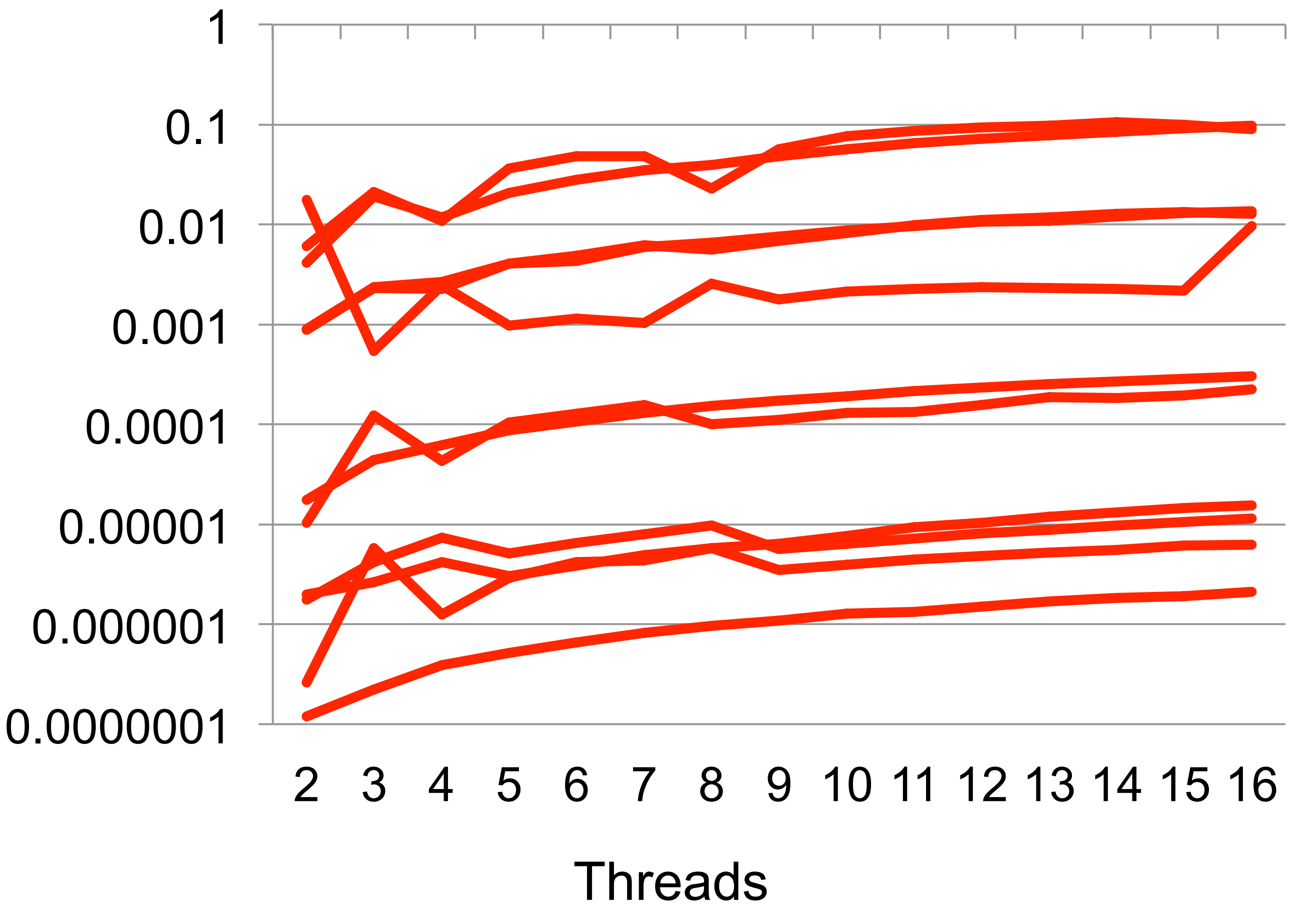
- Hardware Platform
  - 2x8 cores Intel Xeon E5-2450
- Software Platform
  - Jikes RVM (3.1.3)
- Benchmarks
  - UTS, BarnessHut, FFT, Jacobi, LUDecomposition, JGF_SeriesTest, HeatDiffusion, PointCorrelation, NQueens, Matmul, CilkSort and Fibonacci
    - To evaluate performance
  - JMetal (sourceforge project with 327 Java files)
    - To evaluate the productivity of our system

Big…… But How Big ??

Sequential Overhead



Steal to Task Ratio



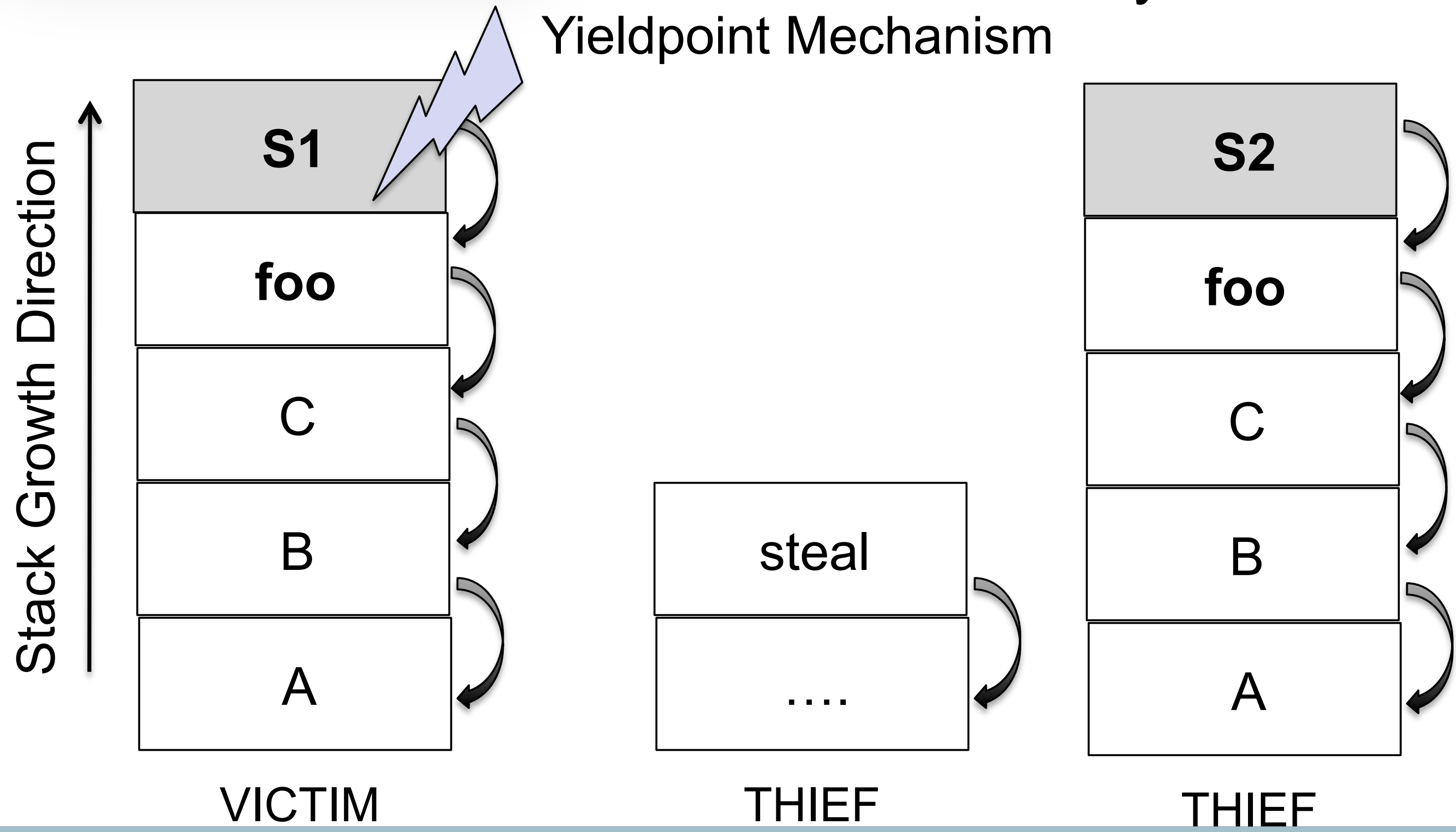
Insights

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

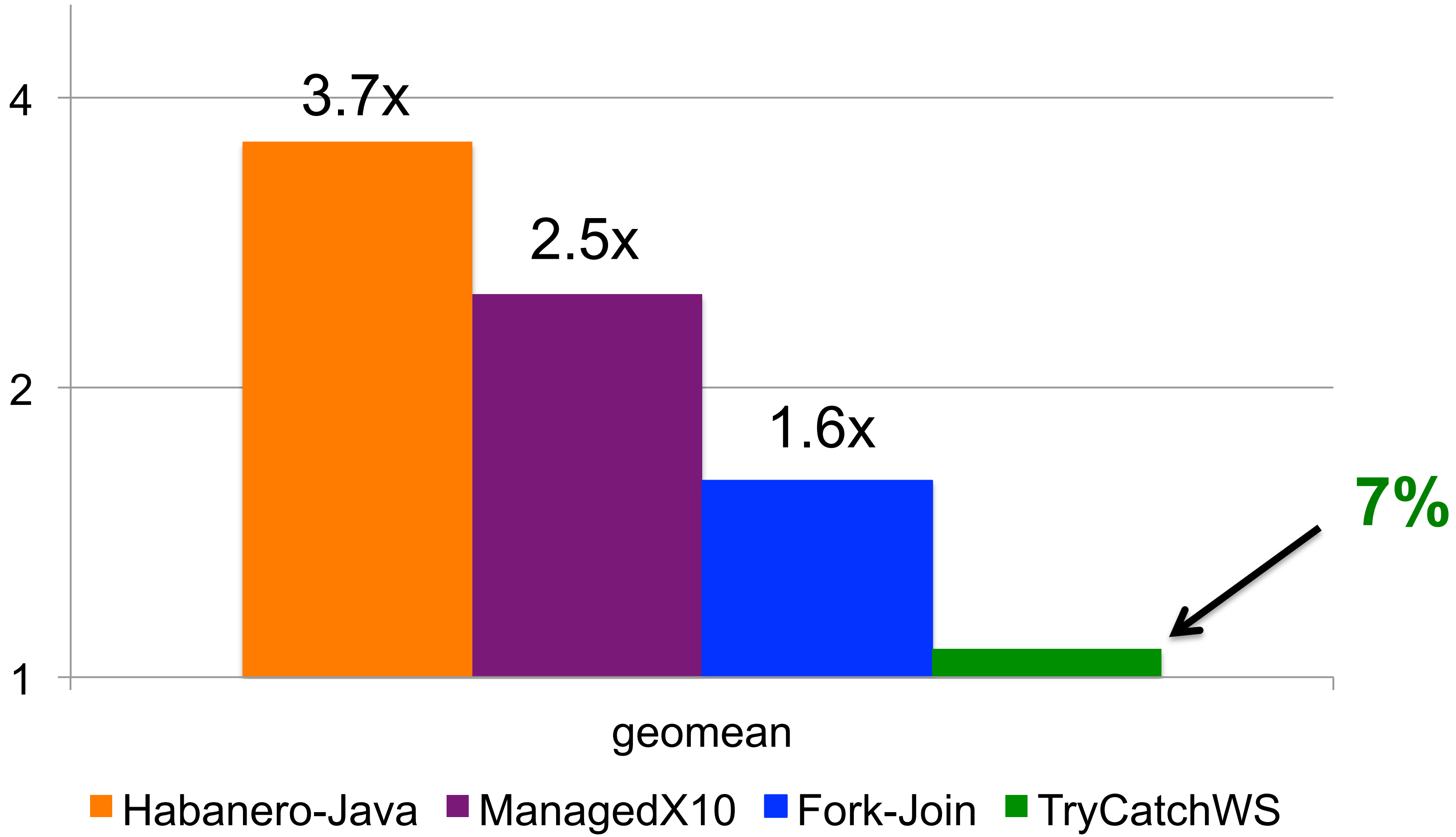
Implementation

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

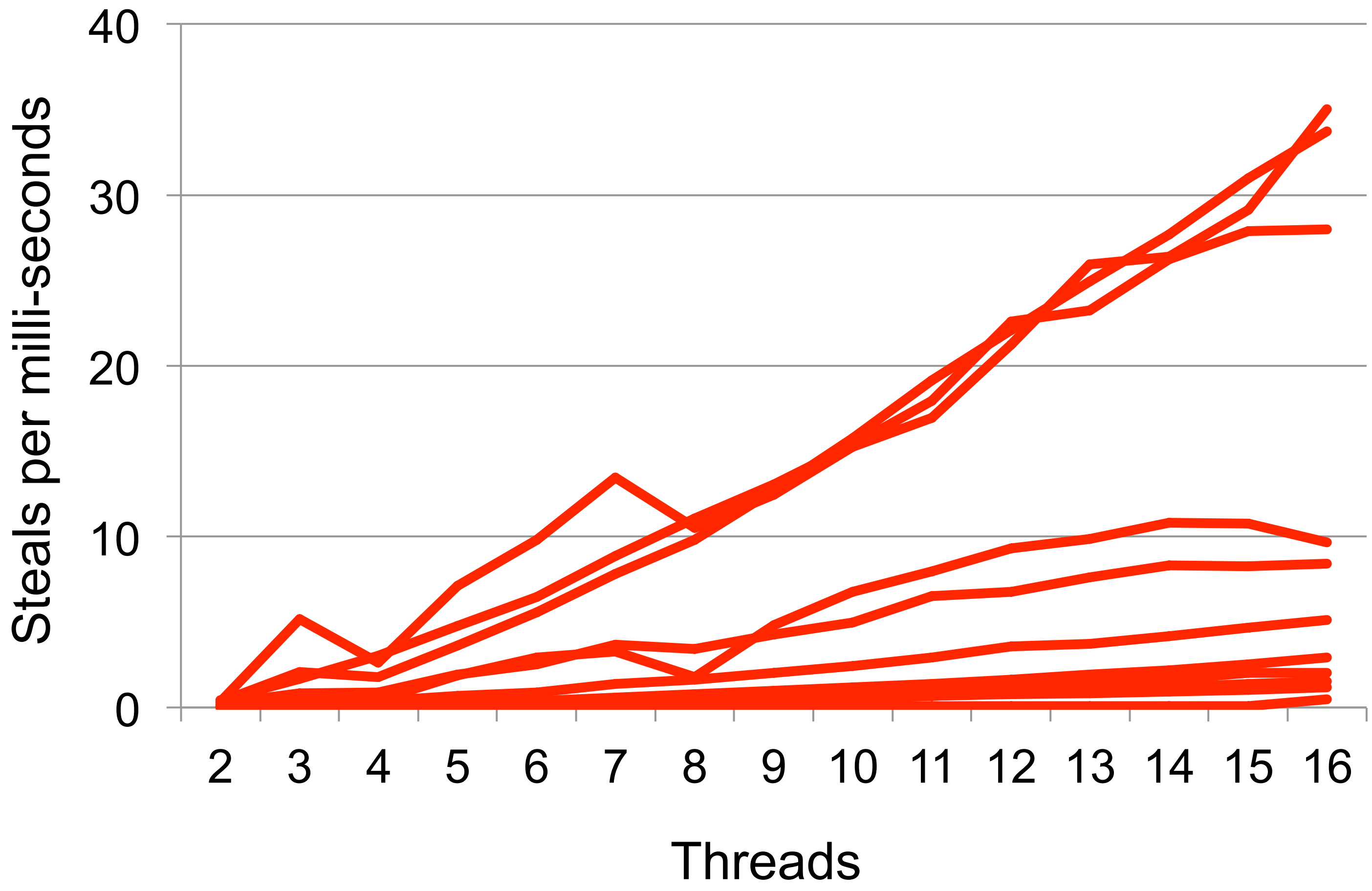
- Yieldpoint mechanism
- On-stack replacement
- Java try/catch exceptions
- Dynamic code patching



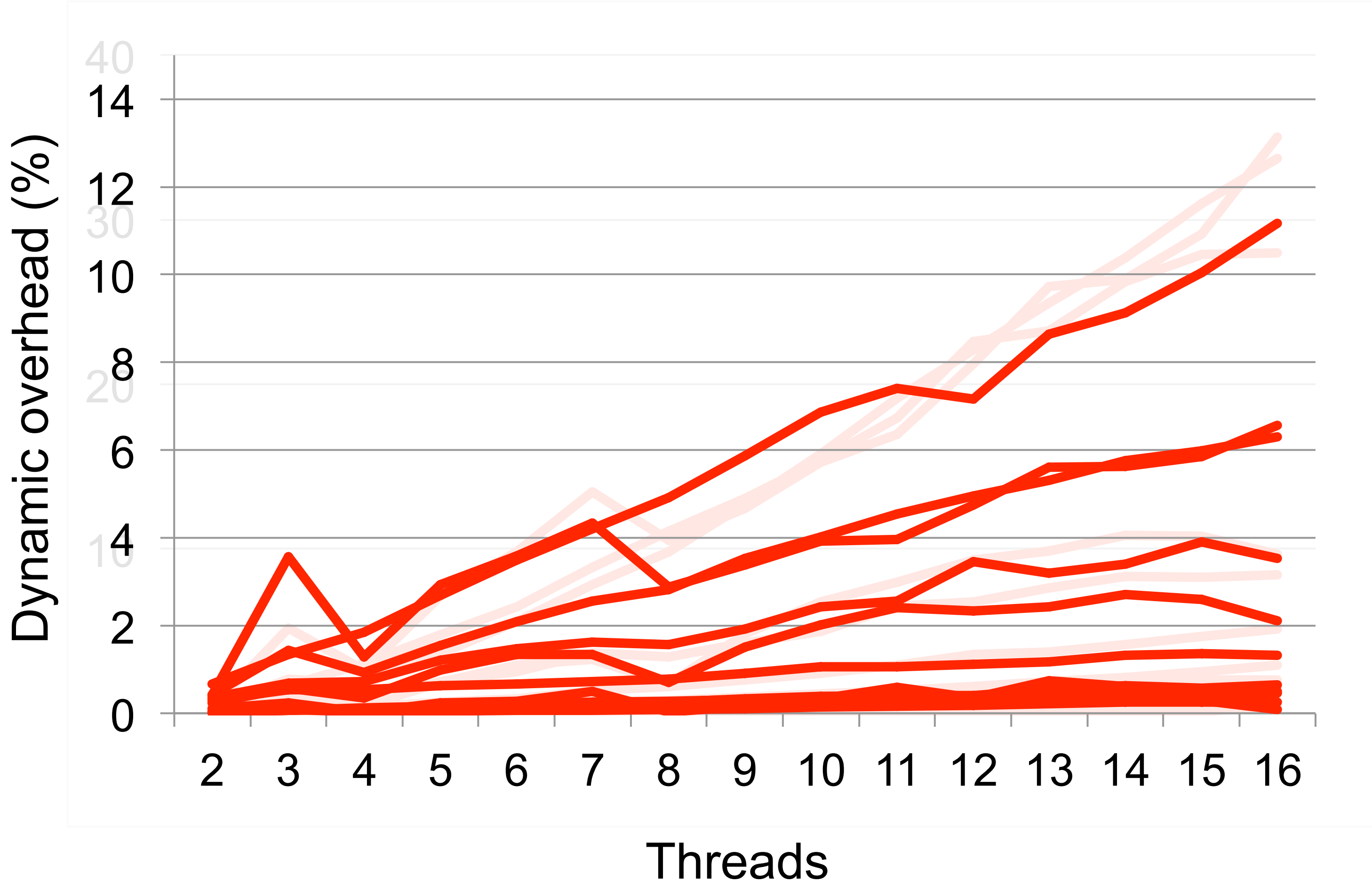
Sequential Overhead



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

Dynamic Overhead

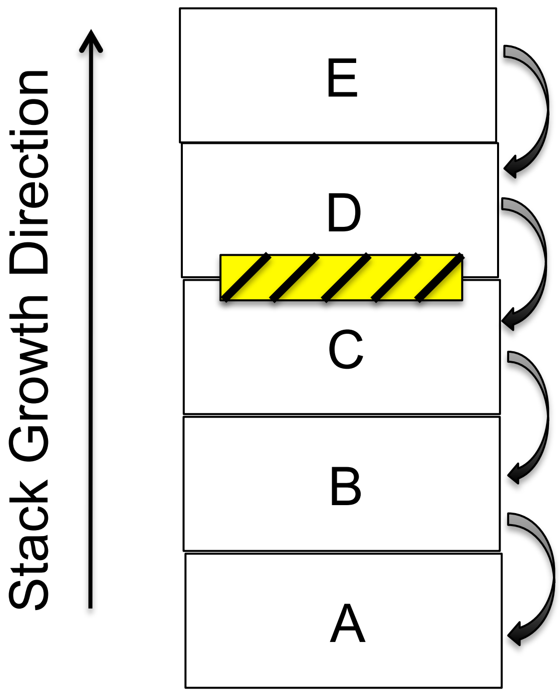


Insights

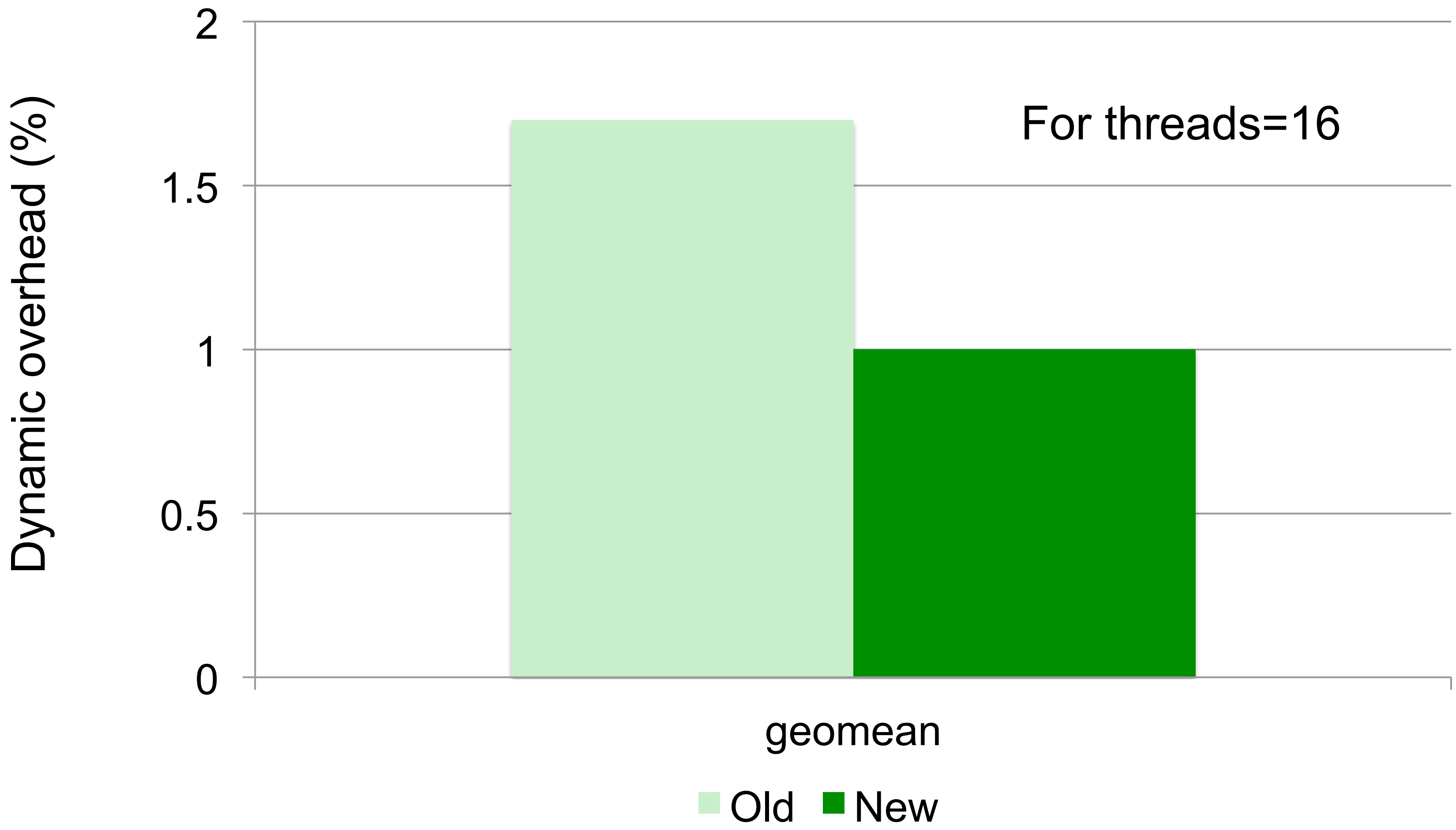
- **Still the same**
  - Re-use existing mechanisms inside modern managed runtimes

Return Barrier

Hijack a return and bridge to some other method



Dynamic Overhead




ON
OFF
MUTE
VOL+
CH+
VOL-
CH-

Productivity in a Large Code Base

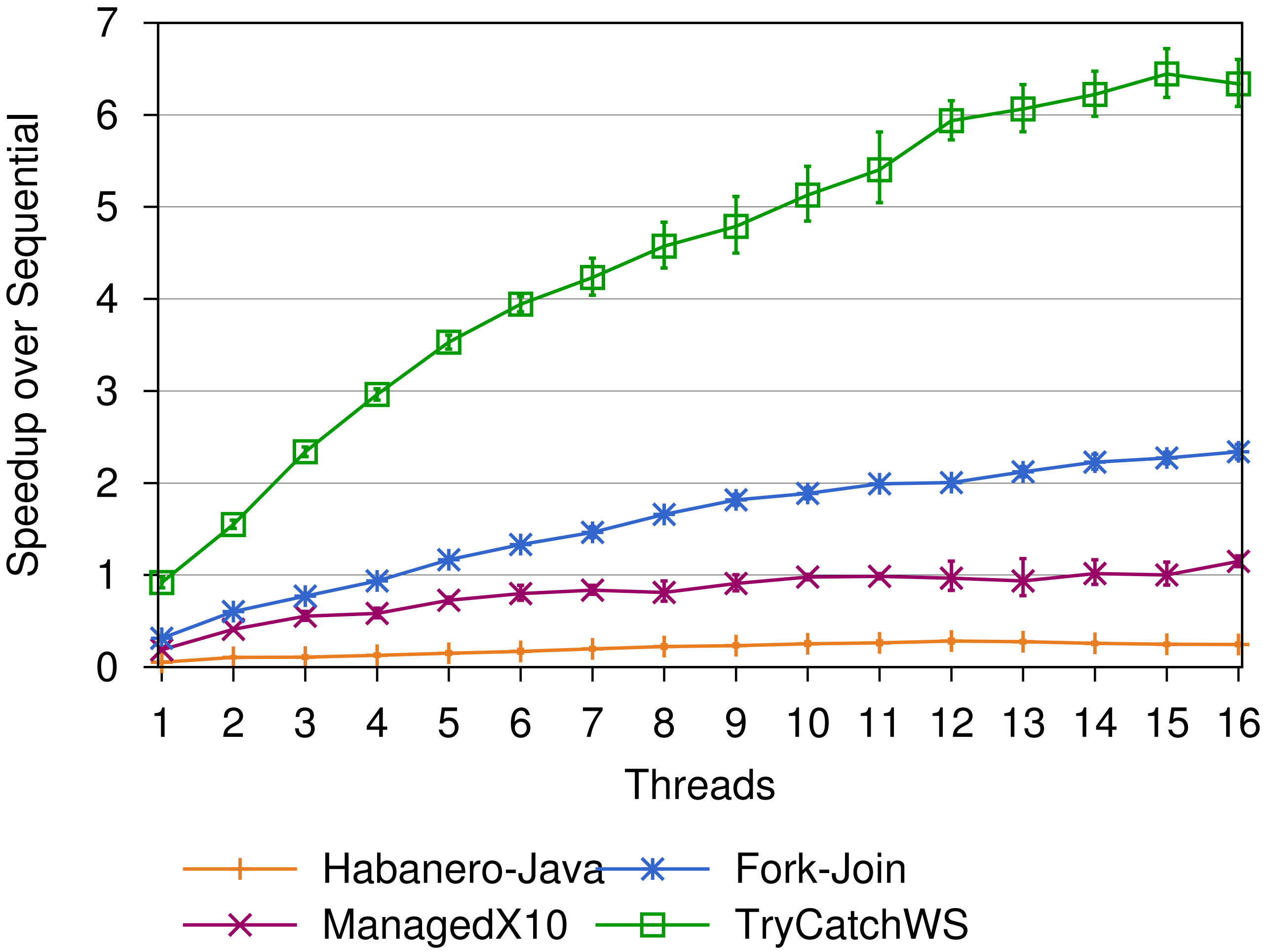
- Project with several hundred files
- Multiple dependencies (inheritance…)
- Achieving parallelism
  - Minimal changes
  - Track fields with atomic updates
  - Avoid deadlocks

Java Language Annotations

- Annotate and leave the rest on compiler
- Parallelism
  - syncsteal {…}
  - steal {…}
- Data centric concurrency control *(Dolby et al. 2012)*
  - @Atomicsets(X)
  - @Atomic(X)
  - @AliasAtomic(Y=this.X)

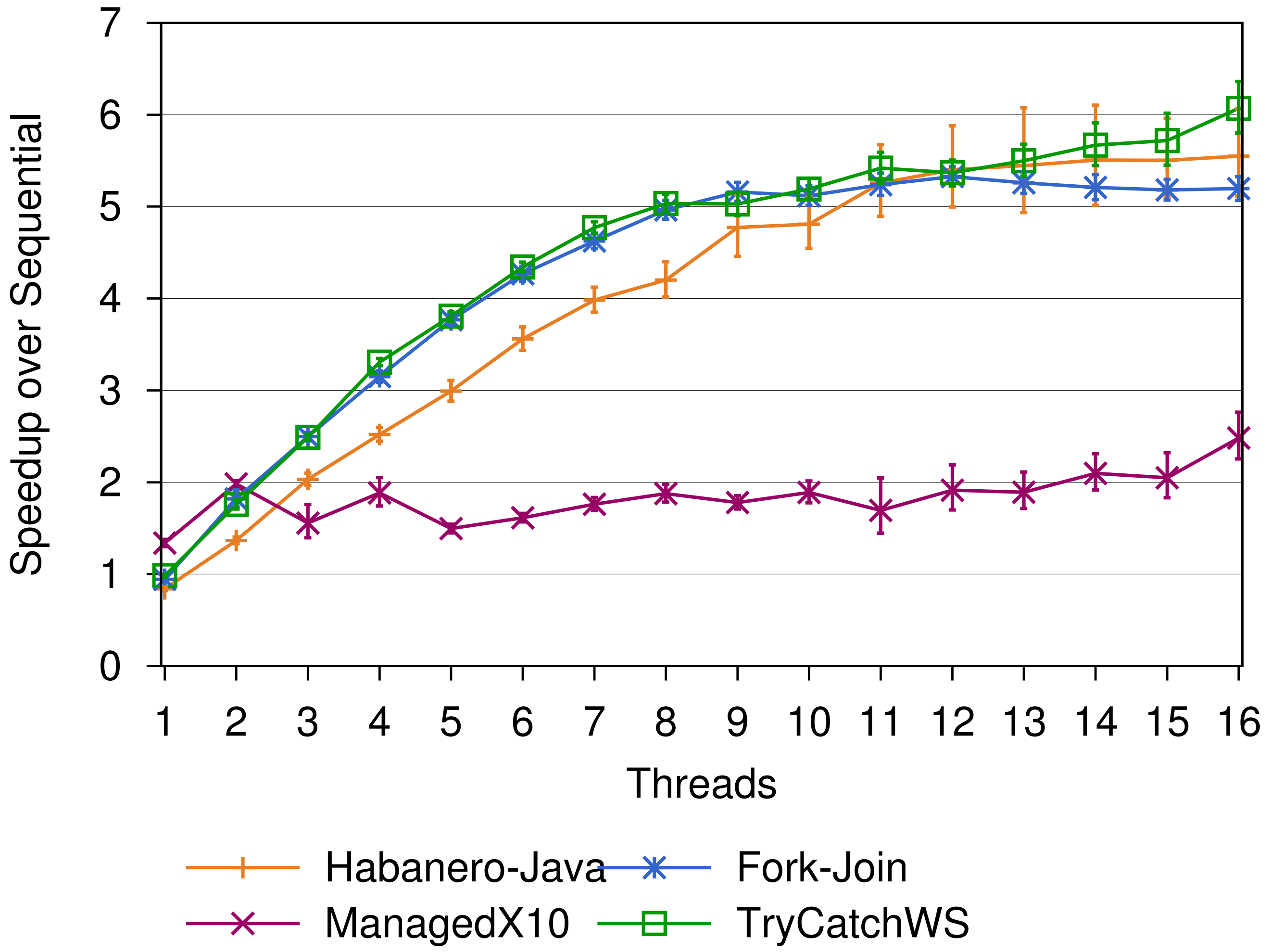
So Where Do We Stand …?

Work–Stealing Performance



Jacobi

Work–Stealing Performance



UTS

Summary and Conclusion

- Work–stealing overheads – sequential and dynamic
- Reused existing mechanisms inside modern managed runtimes
  - Yieldpoint mechanism
  - On-stack replacement
  - Java try/catch exception handling
  - Dynamic code patching
  - Return barrier
- Effectively eliminated sequential overhead (only 7%)
- Halved the dynamic overhead
- Annotations in Java to generate work-stealing calls and synchronization blocks