Work–Stealing by Stealing State from Live Stack Frames of a Running Application

Vivek Kumar
Australian National University

Daniel Frampton
Australian National University

David Grove
IBM T.J. Watson Research Center

Olivier Tardieu
IBM T.J. Watson Research Center

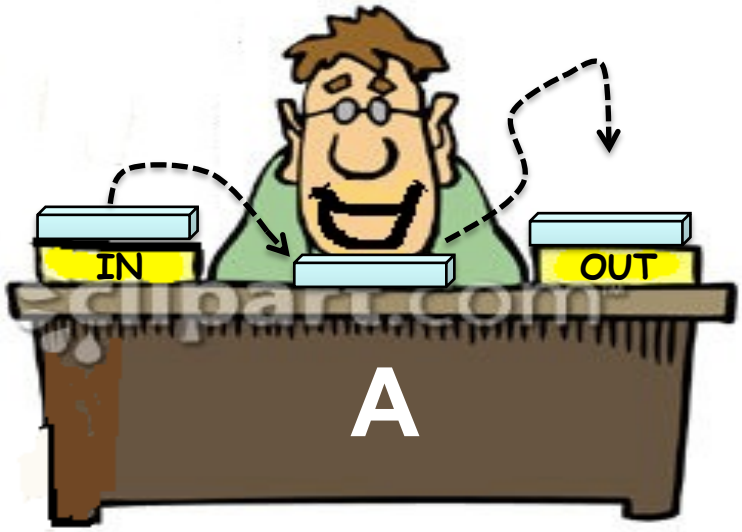
Stephen M. Blackburn
Australian National University

Generously supported by IBM & the Australian Research Council

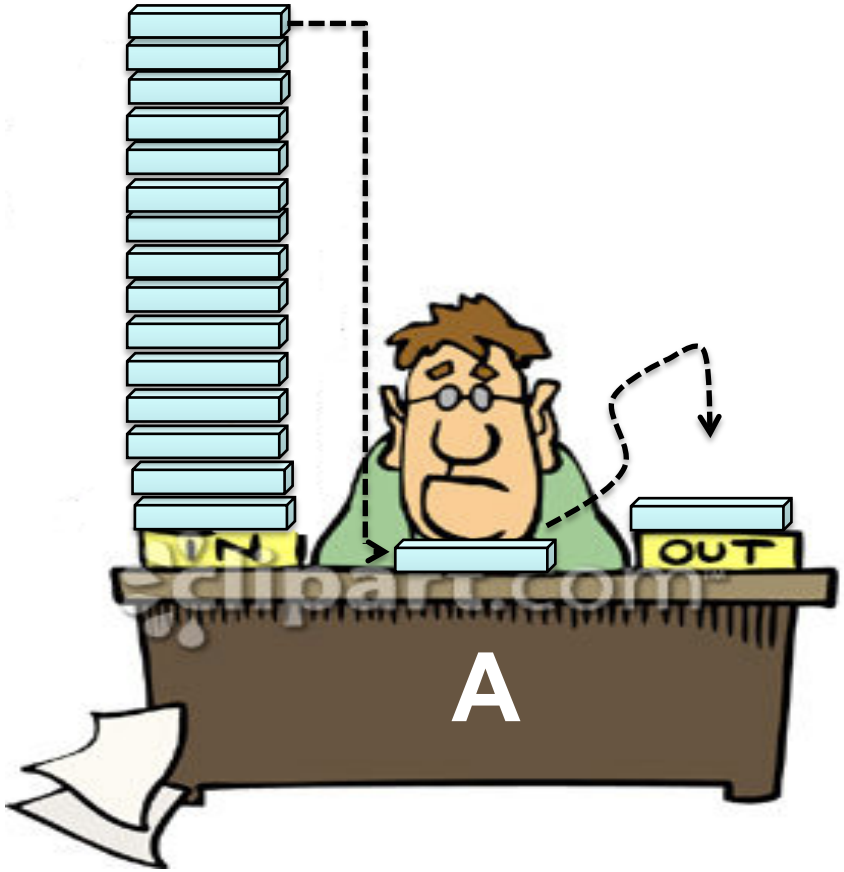
Motivation

- Multicore era
  - Dynamic task parallelism
- Load balancing
  - Work–sharing
    - Central task queue
    - Scalability bottleneck with increase in threads
  - **Work–stealing**
    - Fixed number of threads
    - One task queue per thread

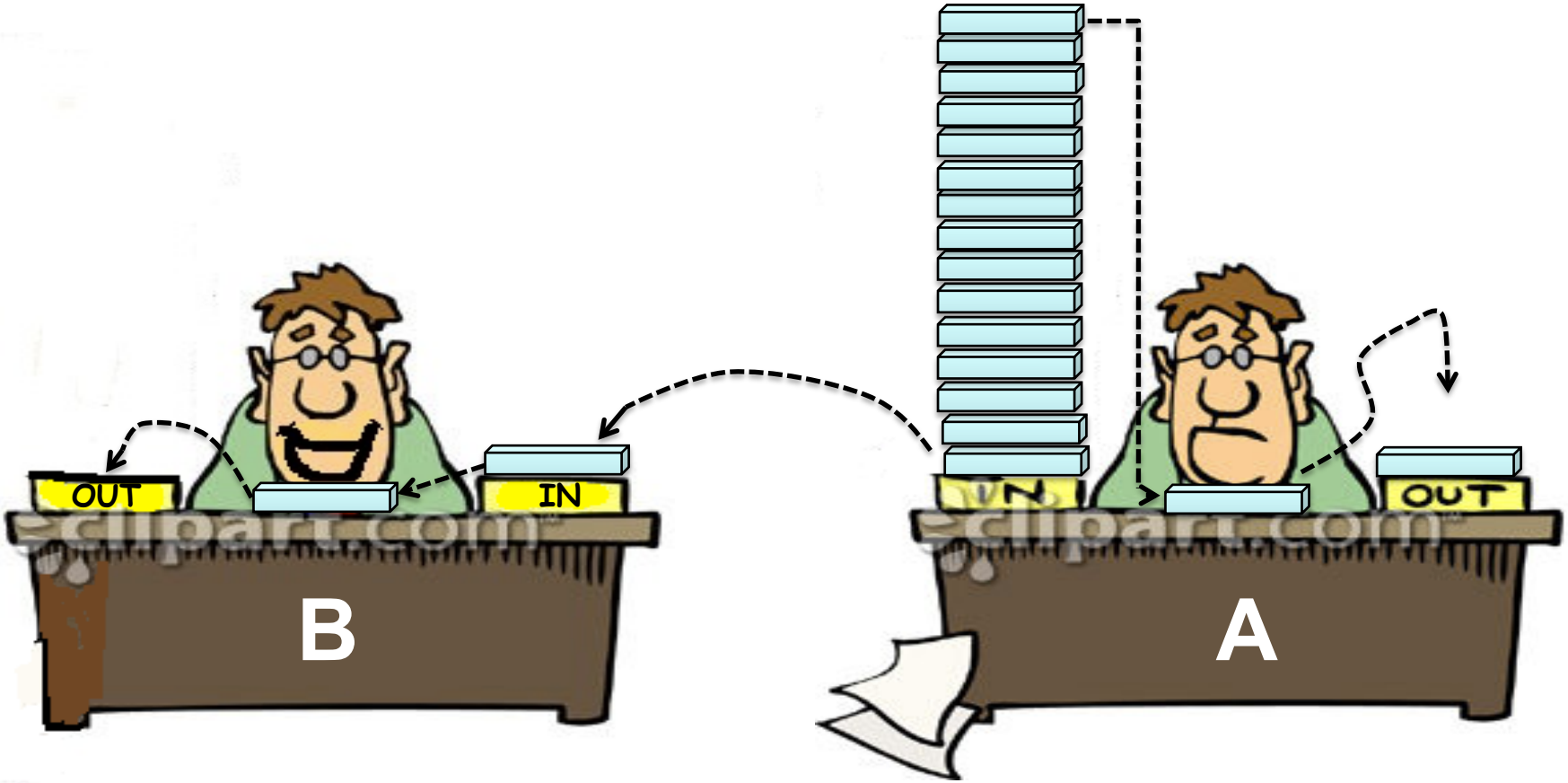
Work–Stealing



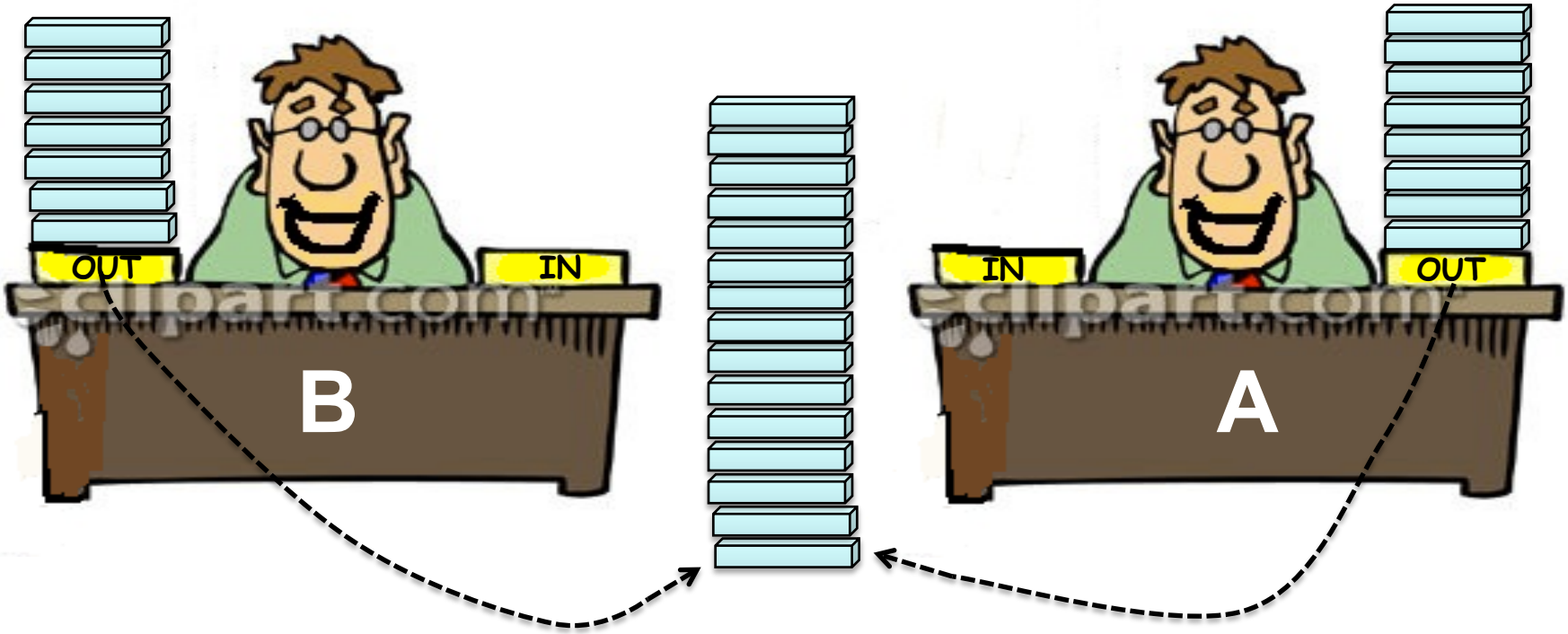
Work–Stealing



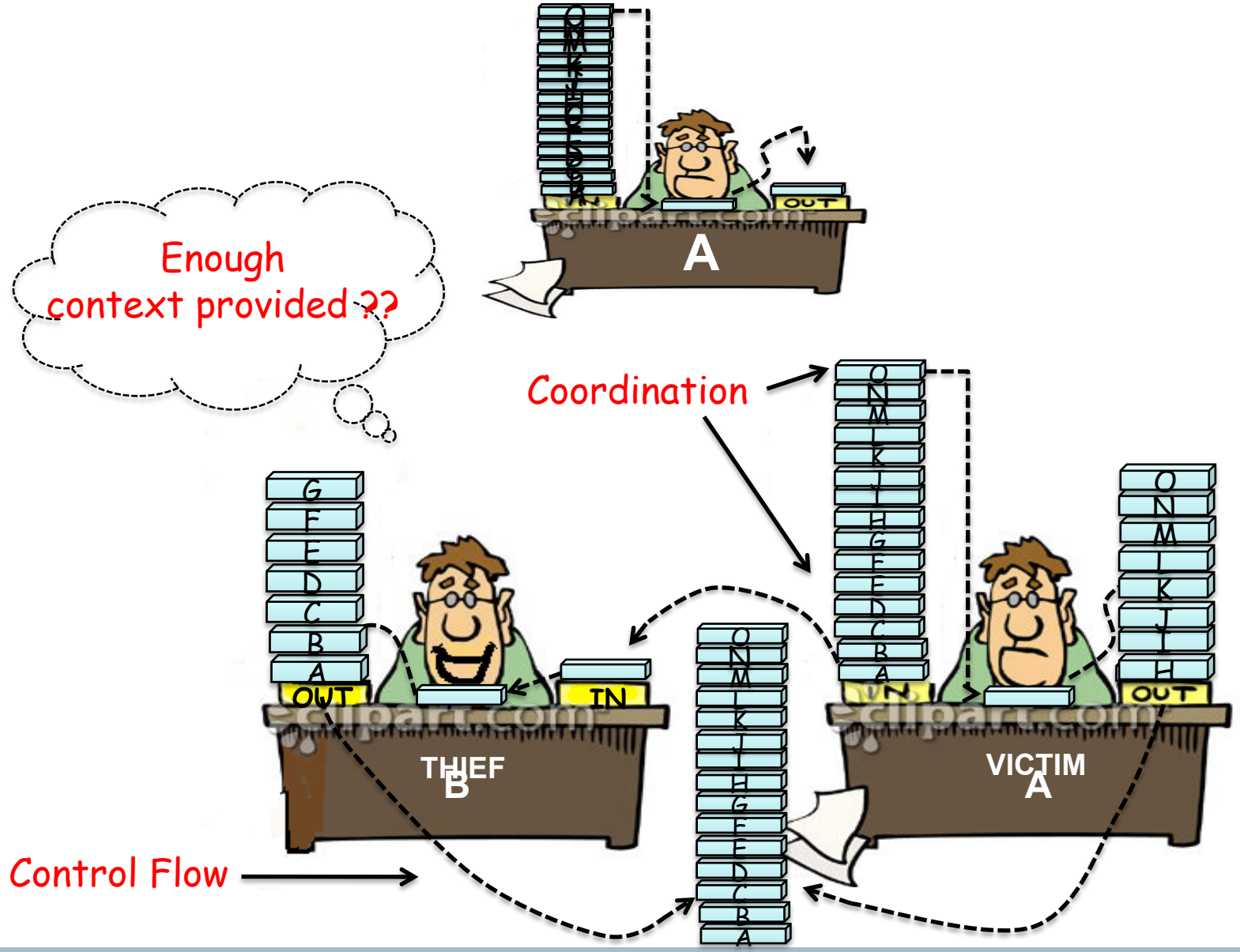
Work–Stealing



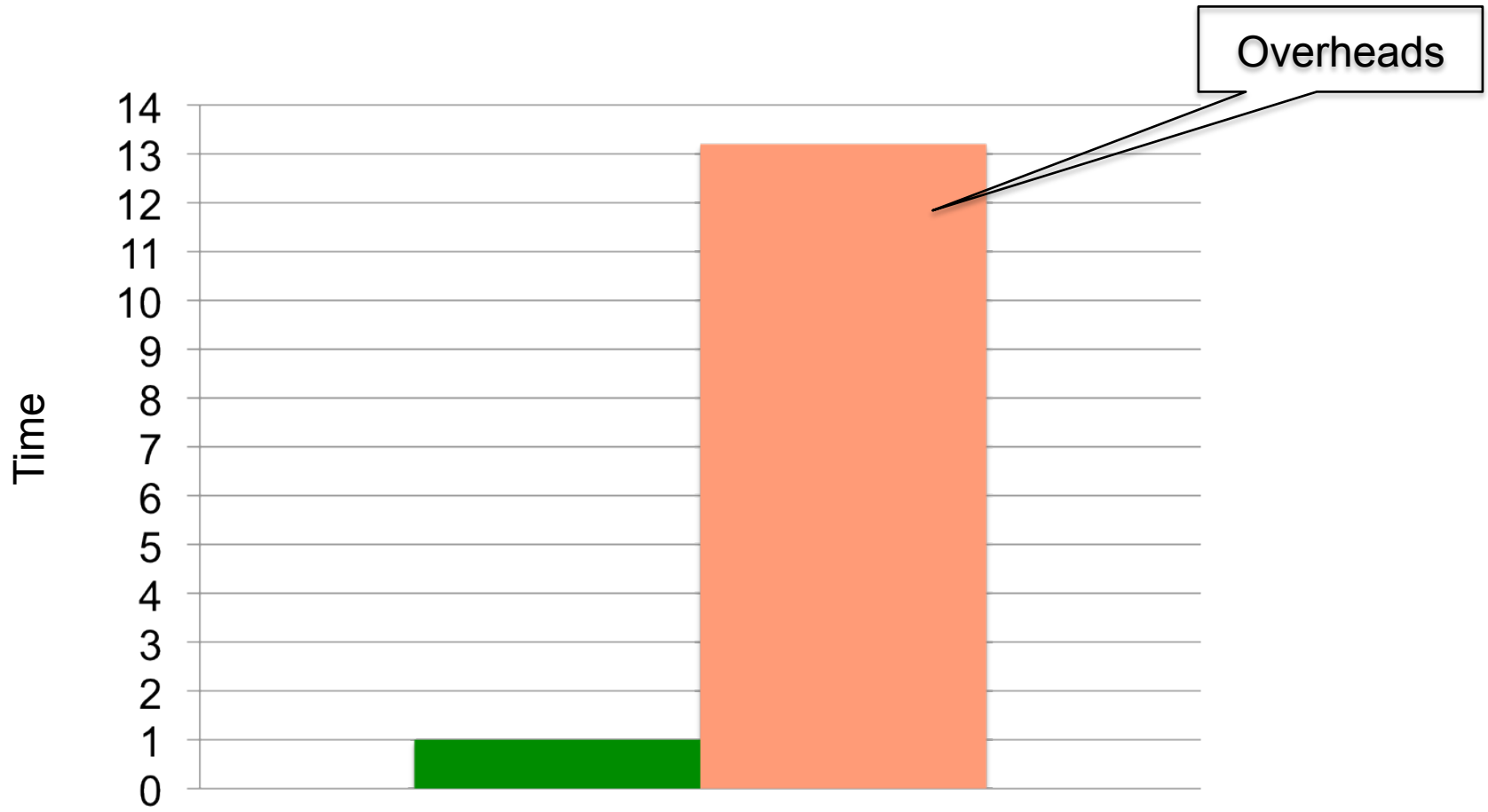
Work–Stealing



Overheads

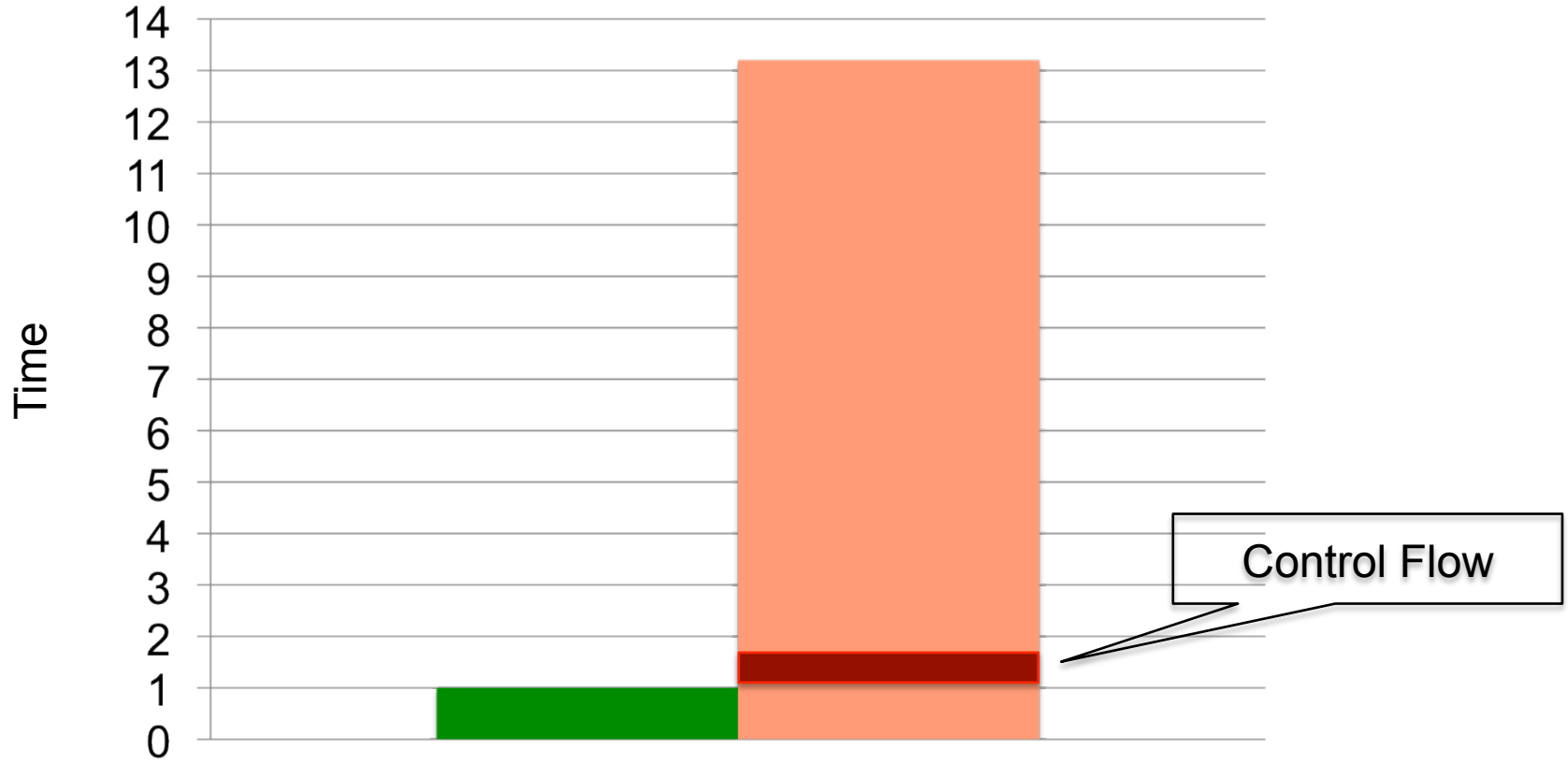

Enough
context provided ??
A
Coordination
THIEF
B
VICTIM
A
OUT
IN
Control Flow

Overheads



X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time

Overheads



X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time

Overhead – Control Flow

```
final static class _$main extends MainFrame {
    public def fast(worker:Worker):void {
        this.n = 20;
        this._pc = 1;
        val tmp:_$mainF0 = new _$mainF0(this);
        tmp.fast(worker);
        this.result = this.a + this.b
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
                this.result = this.a + this.b;
            }
    }
    public def back(workerWorker,frame:Frame):void {}
}
```

```
final static class _$mainF0 extends FinishFrame {
    public def fast(worker:Worker):void {
        val tmp = new _$mainF0A0(ff,ff);
        tmp.fast(worker);
    }
    public def resume(worker:Worker):void {
    public def back(worker:Worker,frame:Frame):void {}
}
```

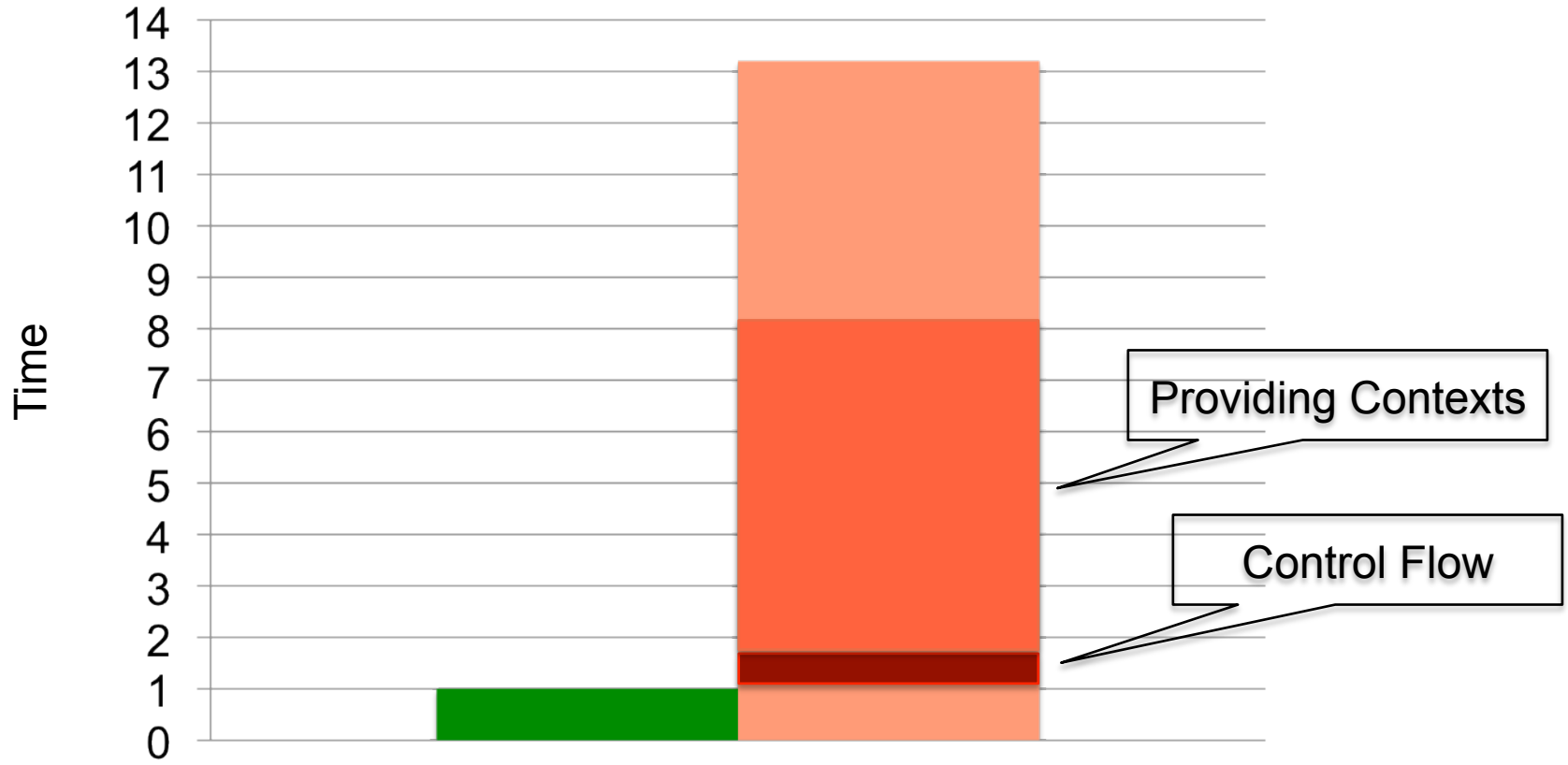
```
final static class _$mainF0A0 extends RegularFrame {
    public def fast(worker:Worker):void {
        this._pc = 1;
        push(worker);
        val tmp:_$mainF0A0B0 = new _$mainF0A0B0(ff);
        tmp.fast(worker);
        _$main.b = B(_$main.n - 1);
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
                case 1:
                _$main.b = B(_$main.n - 1);
        }
    }
    public def back(workerWorker,frame:Frame):void {}
}

final static class _$mainF0B0A0 extends AsyncFrame {
    public def fast(worker:Worker):void {
        _$main.a = A(_$main.n);
            poll(worker);
    }
    public def resume(worker:Worker):void {
    public def back(workerWorker,frame:Frame):void {}
}
```

```
public static def main() {
    val n = 20;
    val a:int;
    val b:int;
    finish {
        async{ a = A(n);}
        b = B(n-1);
    }
    val result = a + b;
}
```

Steal Point

Overheads



X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time

Overhead – Providing Contexts

```
final static class _$main extends MainFrame {
    public def fast(worker:Worker):void {
        this.n = 20;
        this._pc = 1;
        val tmp:_$mainF0 = new _$mainF0(this);
        tmp.fast(worker);
        this.result = this.a + this.b
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
                this.result = this.a + this.b;
            }
    }
    public def back(workerWorker,frame:Frame):void {}
}
```

```
final static class _$mainF0 extends FinishFrame {
    public def fast(worker:Worker):void {
        val tmp = new _$mainF0A0(ff,ff);
        tmp.fast(worker);
    }
    public def resume(worker:Worker):void {
    public def back(worker:Worker,frame:Frame):void {}
}
```

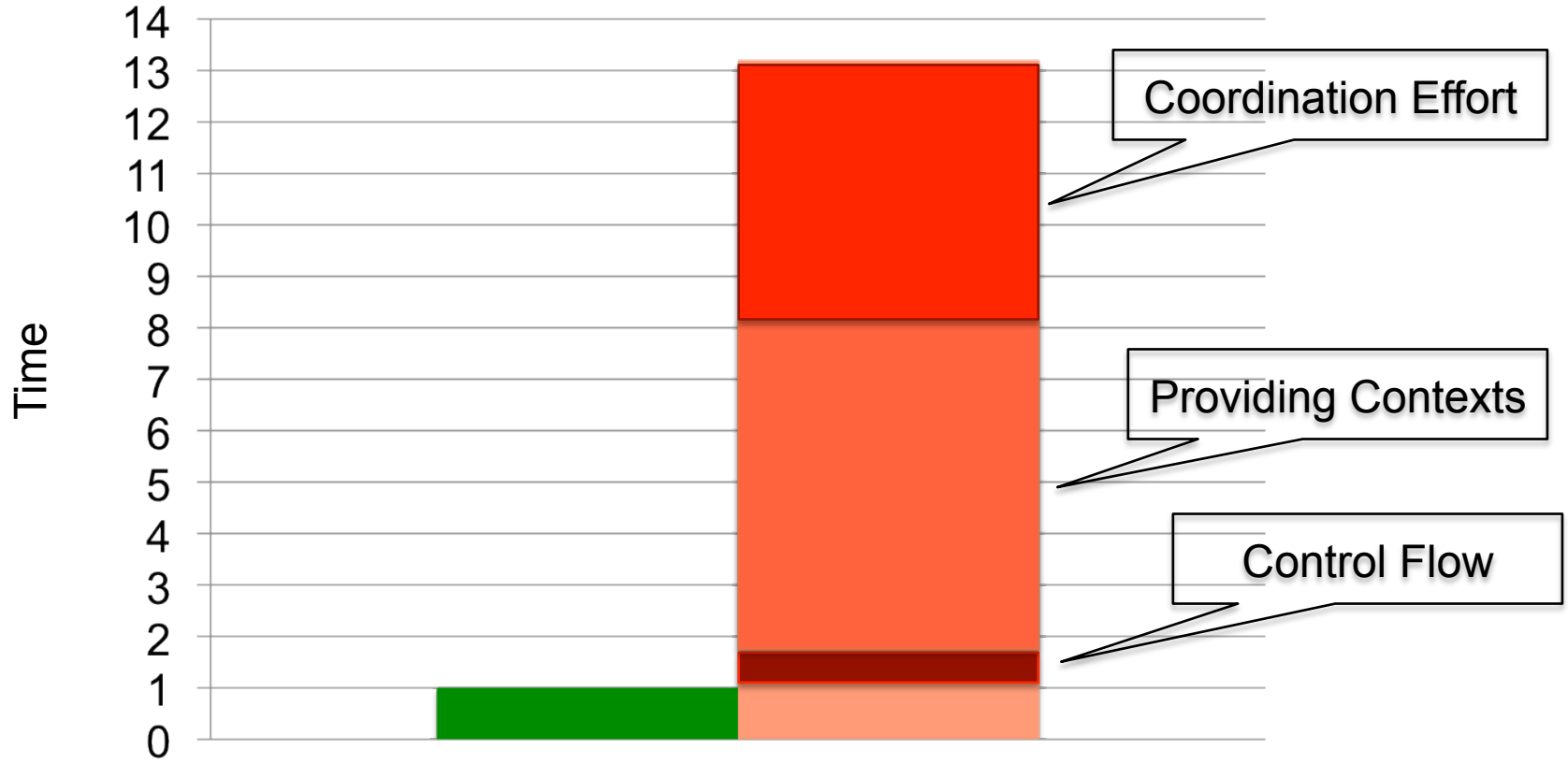
```
final static class _$mainF0A0 extends RegularFrame {
    public def fast(worker:Worker):void {
        this._pc = 1;
        push(worker);
        val tmp:_$mainF0A0B0 = new _$mainF0A0B0(ff);
        tmp.fast(worker);
        _$main.b = B(_$main.n - 1);
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
                case 1:
                _$main.b = B(_$main.n - 1);
        }
    }
    public def back(workerWorker,frame:Frame):void {}
}


final static class _$mainF0B0A0 extends AsyncFrame {
    public def fast(worker:Worker):void {
        _$main.a = A(_$main.n);
            poll(worker);
    }
    public def resume(worker:Worker):void {
    public def back(workerWorker,frame:Frame):void {}
}
```

```
public static def main() {
    val n = 20;
    val a:int;
    val b:int;
    finish {
        async{ a = A(n);}
        b = B(n-1);
    }
    val result = a + b;
}
```

Steal Point

Overheads



X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time

Overhead – Coordination Effort

```
final static class _$main extends MainFrame {
    public def fast(worker:Worker):void {
        this.n = 20;
        this._pc = 1;
        val tmp:_$mainF0 = new _$mainF0(this);
        tmp.fast(worker);
        this.result = this.a + this.b
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
                this.result = this.a + this.b;
            }
    }
    public def back(workerWorker,frame:Frame):void {}
}
```

```
final static class _$mainF0 extends FinishFrame {
    public def fast(worker:Worker):void {
        val tmp = new _$mainF0A0(ff,ff);
        tmp.fast(worker);
    }
    public def resume(worker:Worker):void {
    public def back(worker:Worker,frame:Frame):void {}
}
```

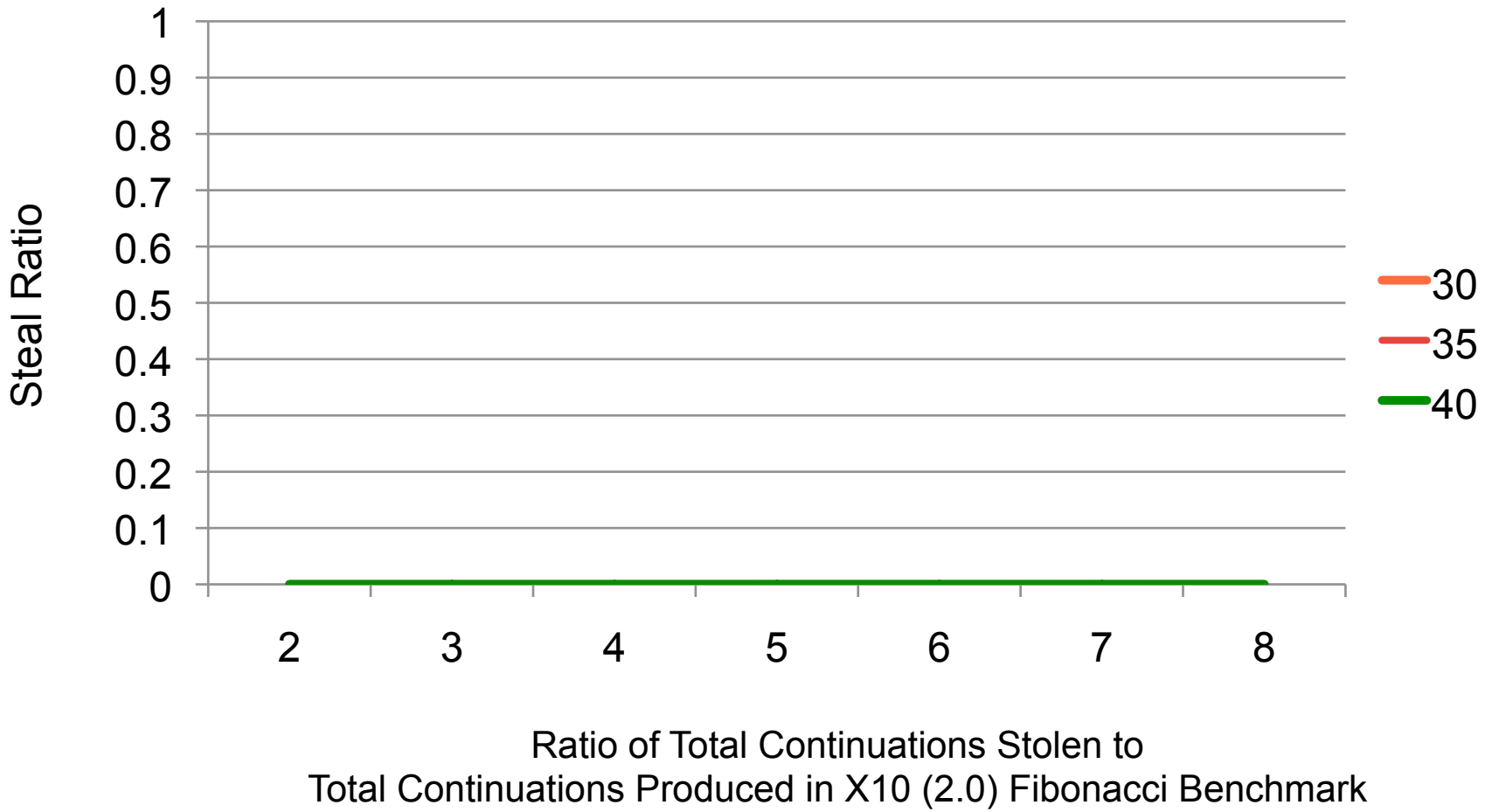
```
final static class _$mainF0A0 extends RegularFrame {
    public def fast(worker:Worker):void {
        this._pc = 1;
        push(worker);
        val tmp:_$mainF0A0B0 = new _$mainF0A0B0(ff);
        tmp.fast(worker);
        _$main.b = B(_$main.n - 1);
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
                _$main.b = B(_$main.n - 1);
        }
    }
    public def back(workerWorker,frame:Frame):void {}
}

final static class _$mainF0B0A0 extends AsyncFrame {
    public def fast(worker:Worker):void {
        _$main.a = A(_$main.n);
        poll(worker);
    }
    public def resume(worker:Worker):void {
    public def back(workerWorker,frame:Frame):void {}
}
```

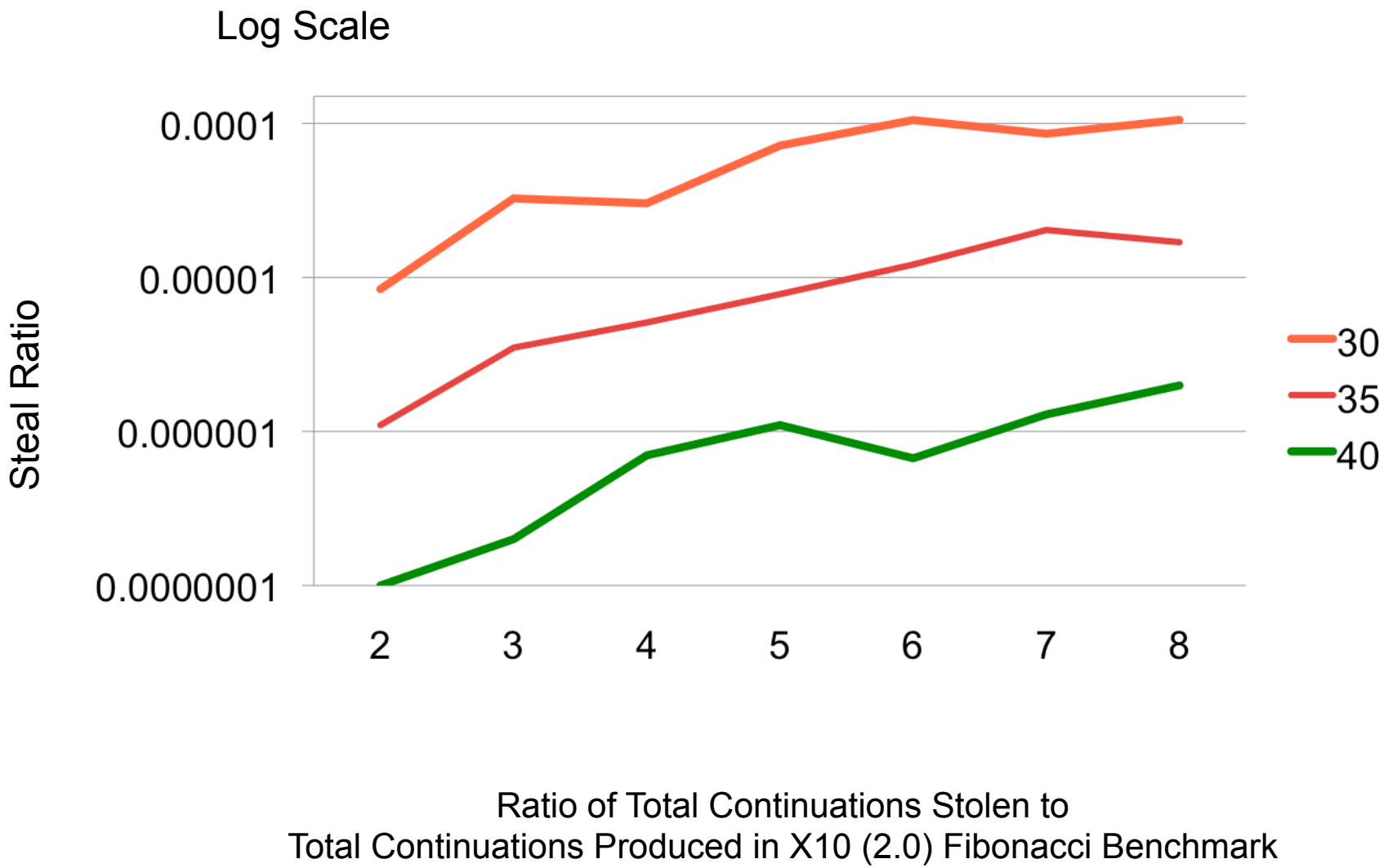
```
public static def main() {
    val n = 20;
    val a:int;
    val b:int;
    finish {
        async{ a = A(n);}
        b = B(n-1);
    }
    val result = a + b;
}
```

Steal Point

Steal Ratio


Steal Ratio
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
2
3
4
5
6
7
8
30
35
40
Ratio of Total Continuations Stolen to
Total Continuations Produced in X10 (2.0) Fibonacci Benchmark

Steal Ratio

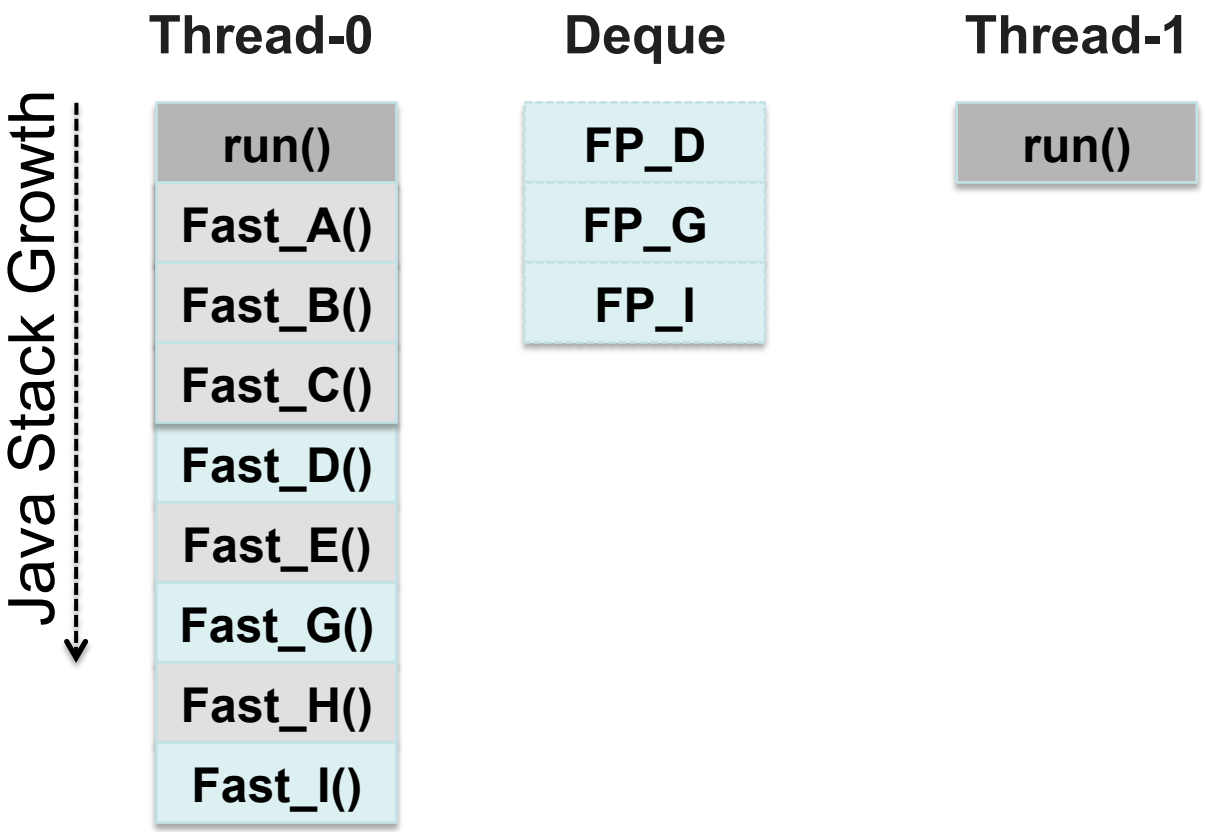

Log Scale
Steal Ratio
0.0001
0.00001
0.000001
0.0000001
2
3
4
5
6
7
8
30
35
40
Ratio of Total Continuations Stolen to
Total Continuations Produced in X10 (2.0) Fibonacci Benchmark

VM Supported Work–Stealing

- Our philosophy:
  - Small steal ratio
    - ➢ Saving contexts for every continuation inefficient
  - Provide contexts only when steal occurs
- Our approach:
  - Thief steals victim's Java stack frame
  - Thief forces the victim to yield to start the steal

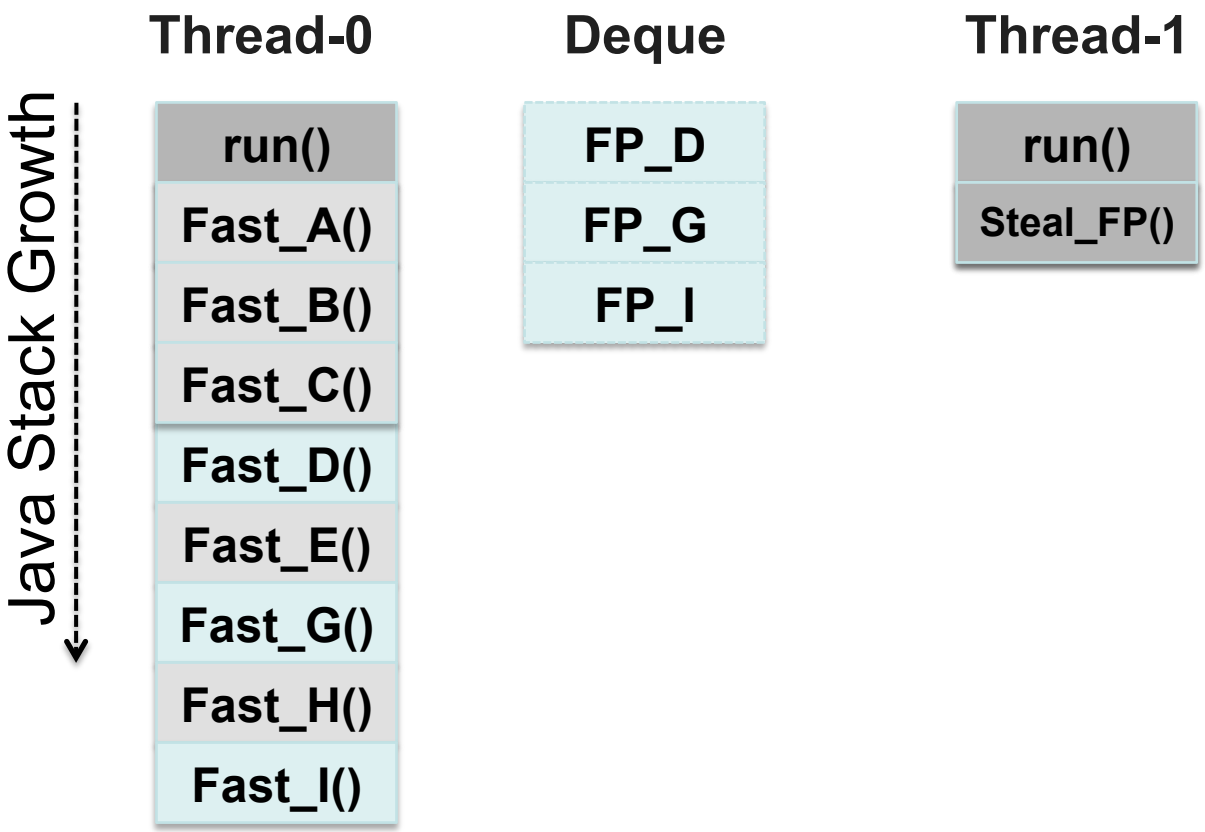
VM Assistance

- Stealing the Stack Frames



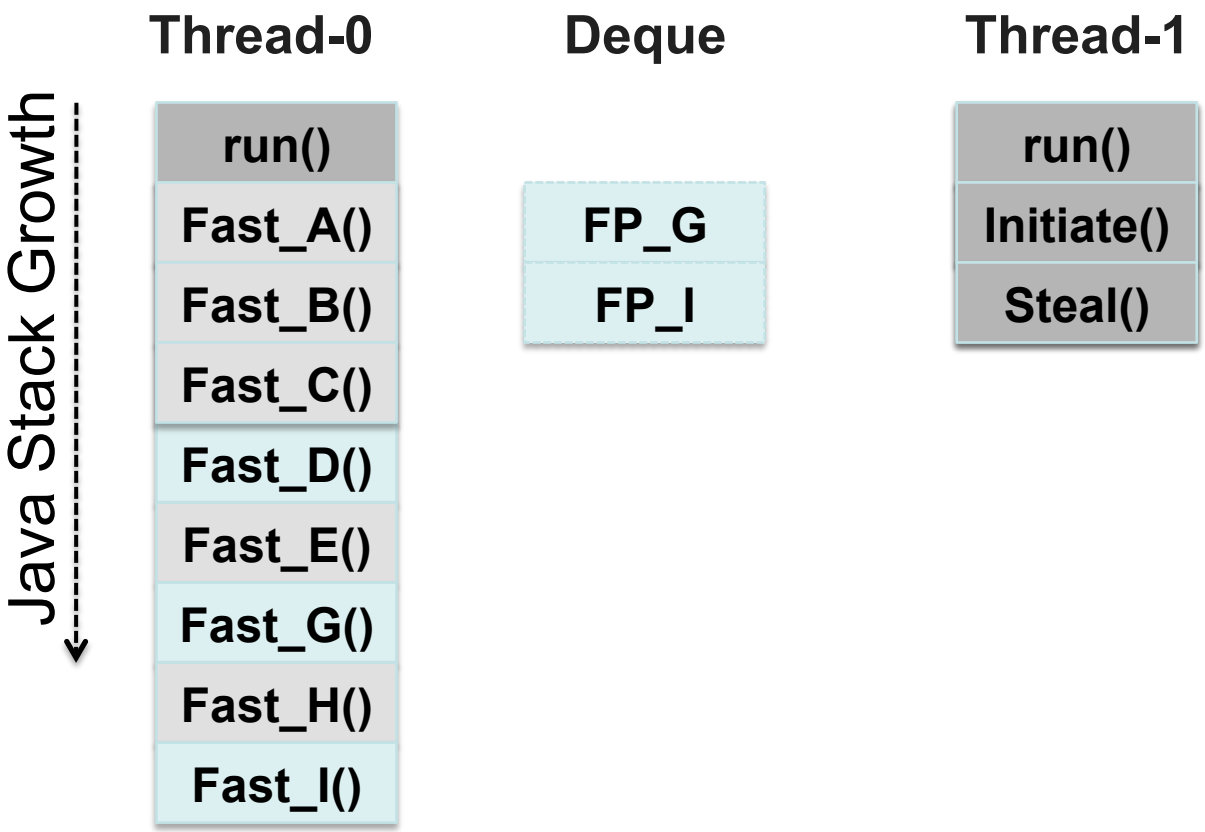
VM Assistance

- Stealing the Stack Frames



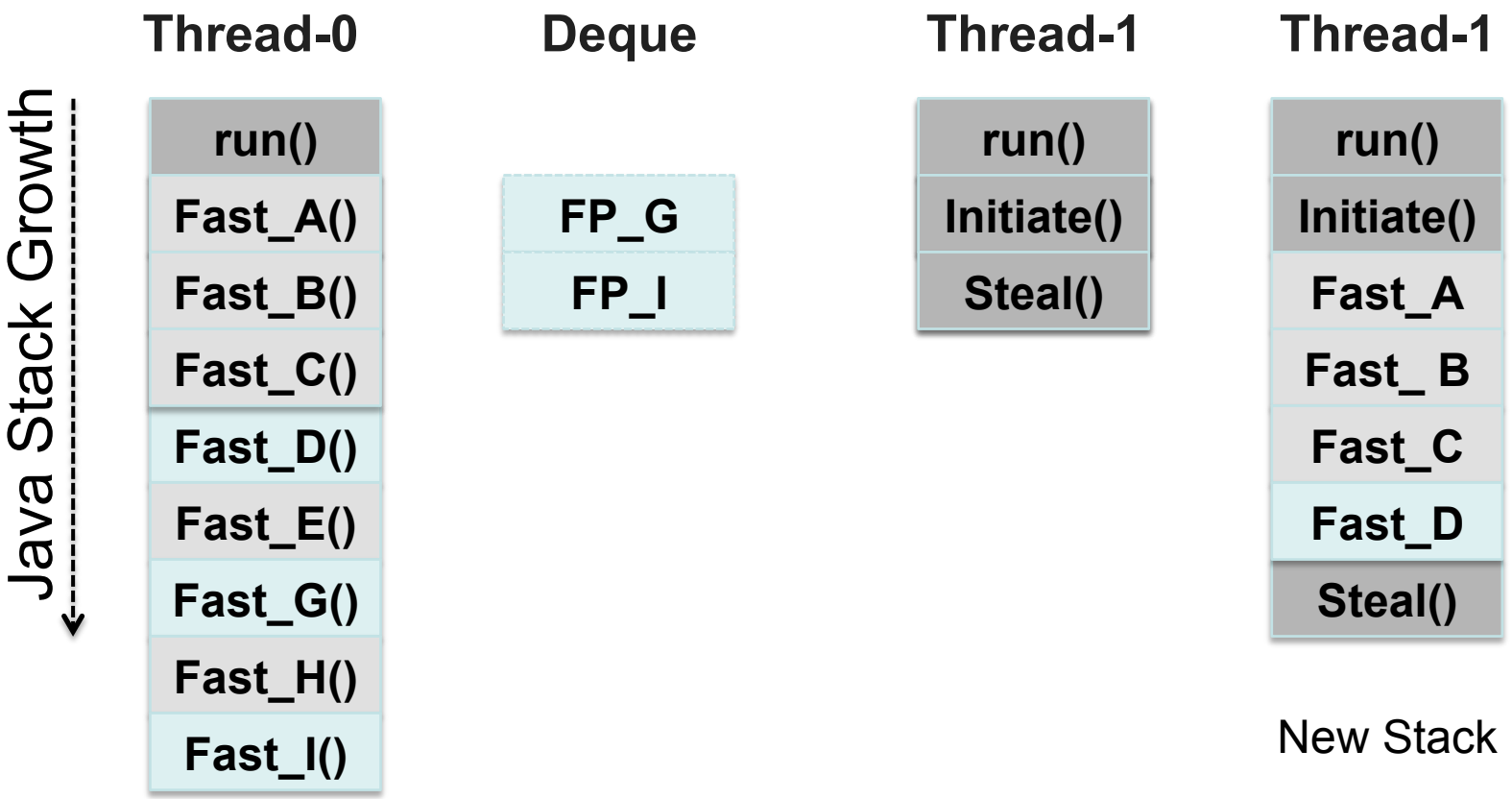
VM Assistance

- Stealing the Stack Frames



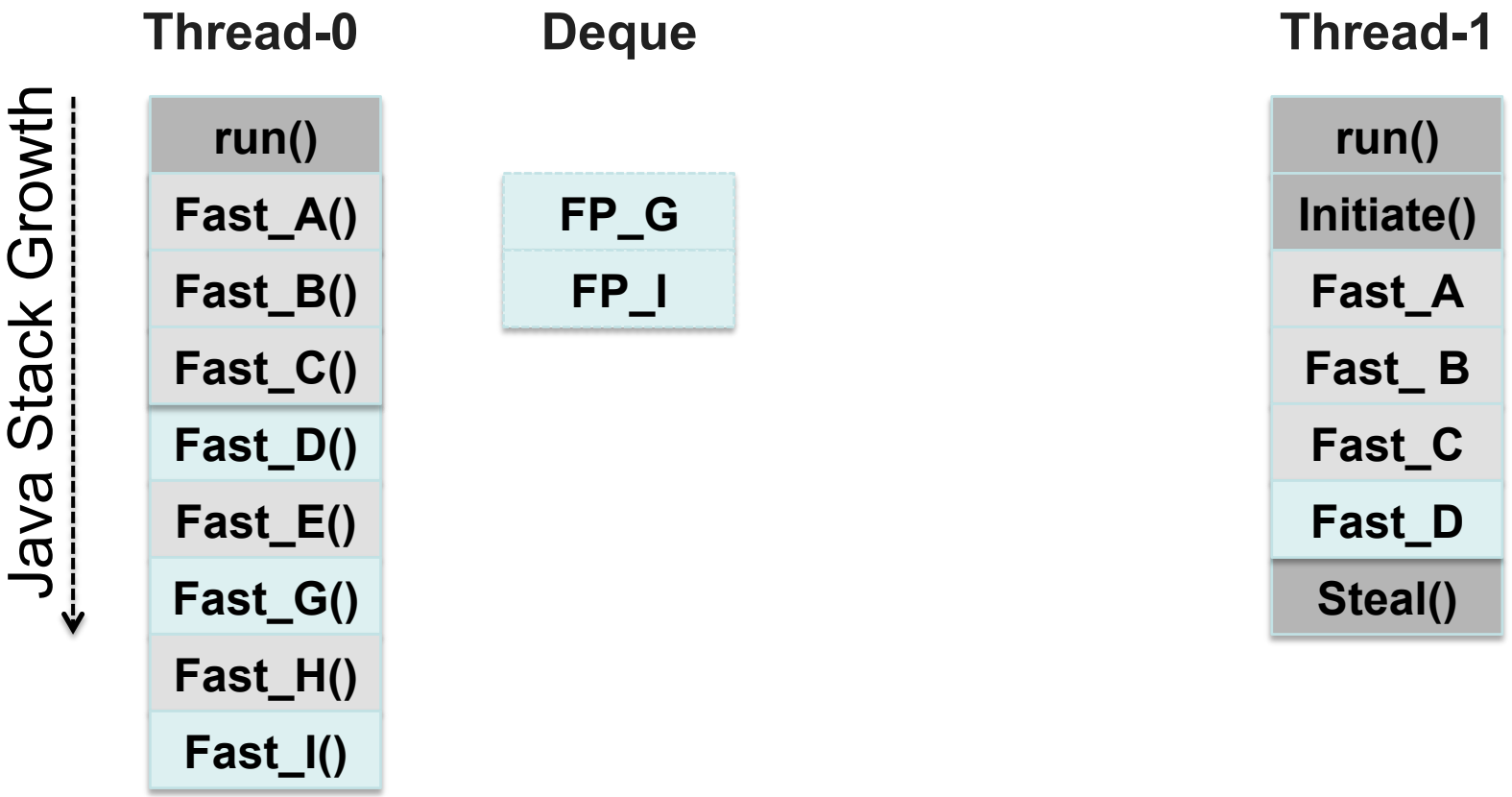
VM Assistance

- Stealing the Stack Frames



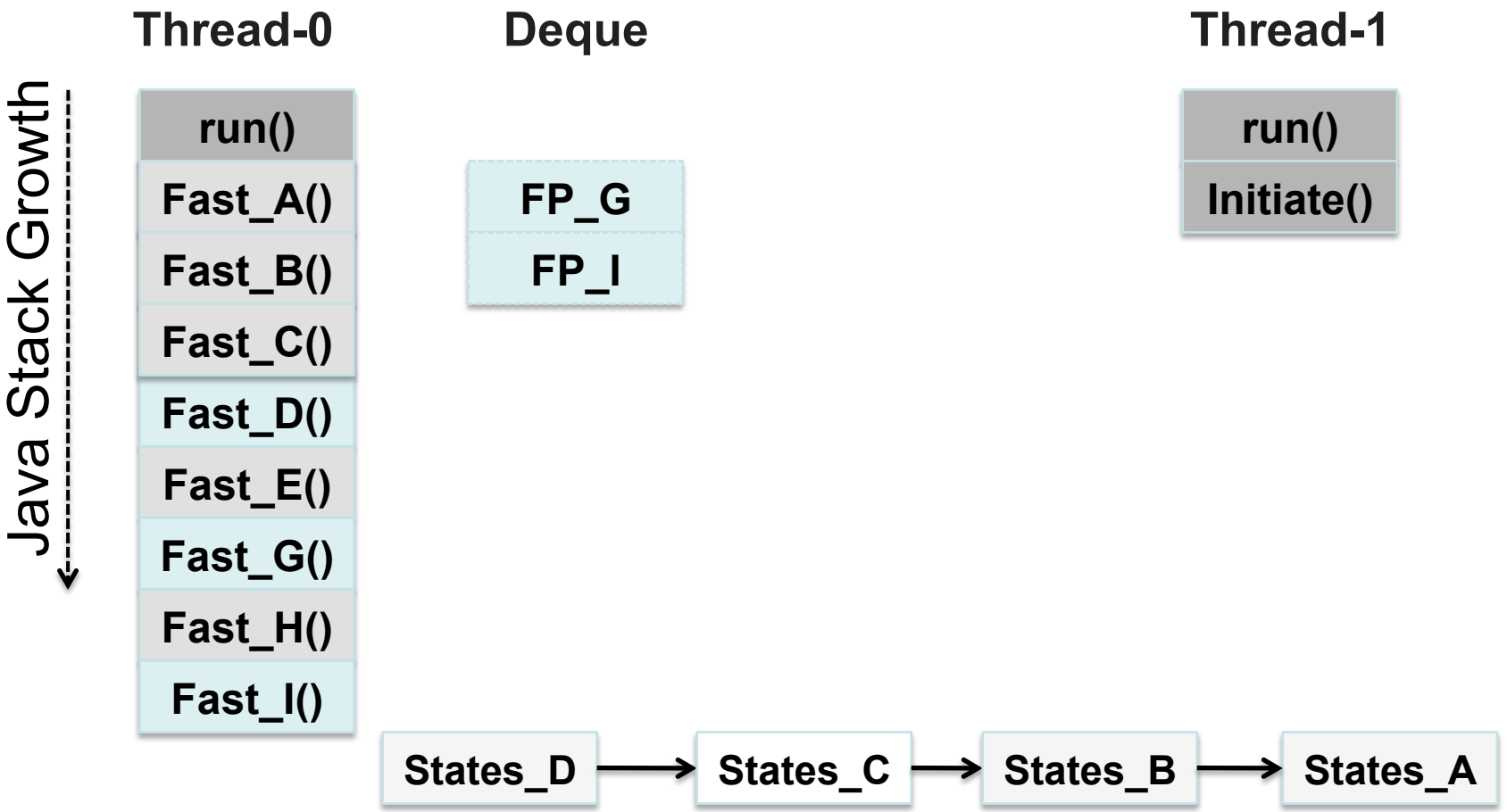
VM Assistance

- Stealing the Stack Frames



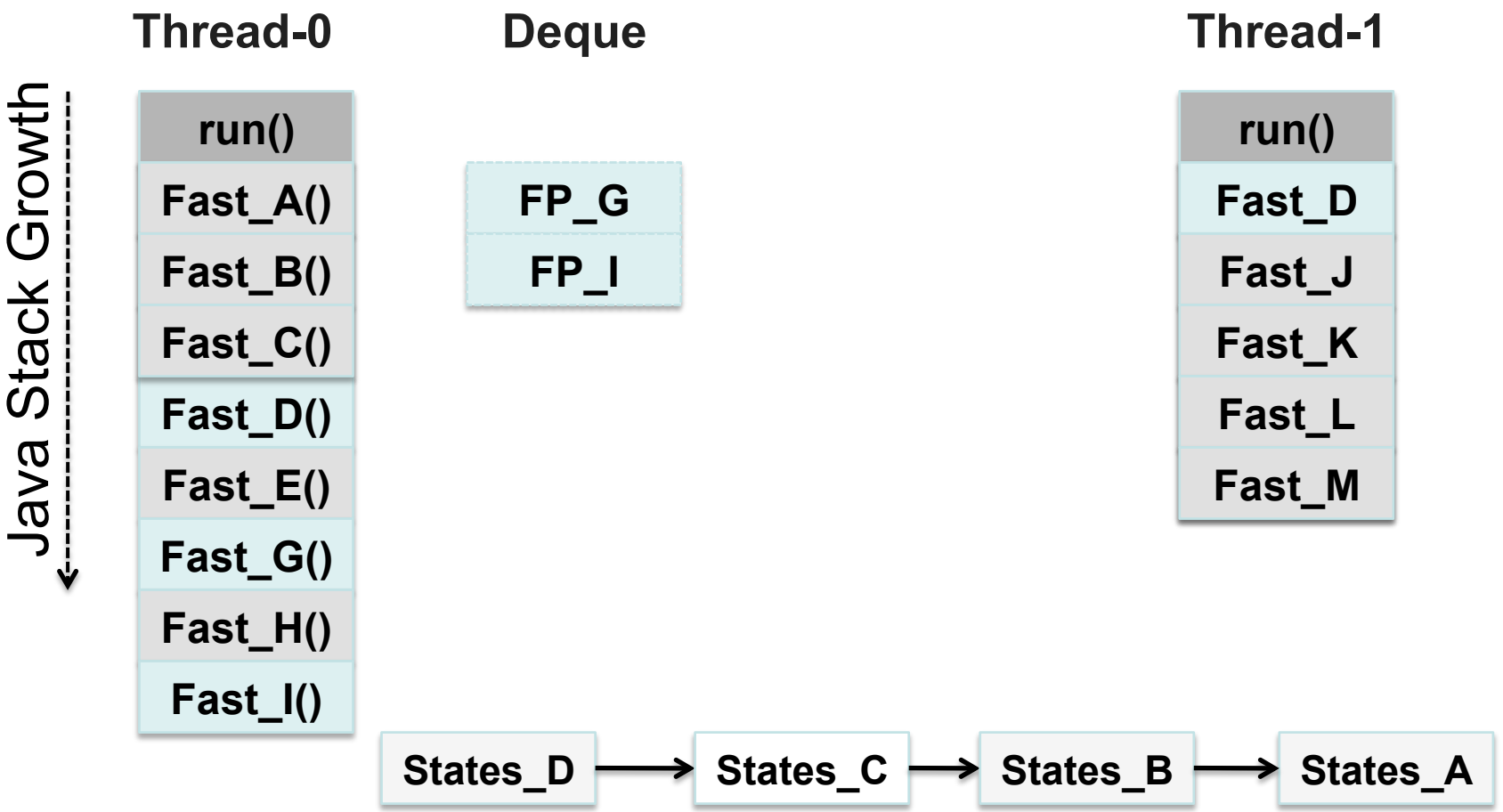
VM Assistance

- Stealing the Stack Frames



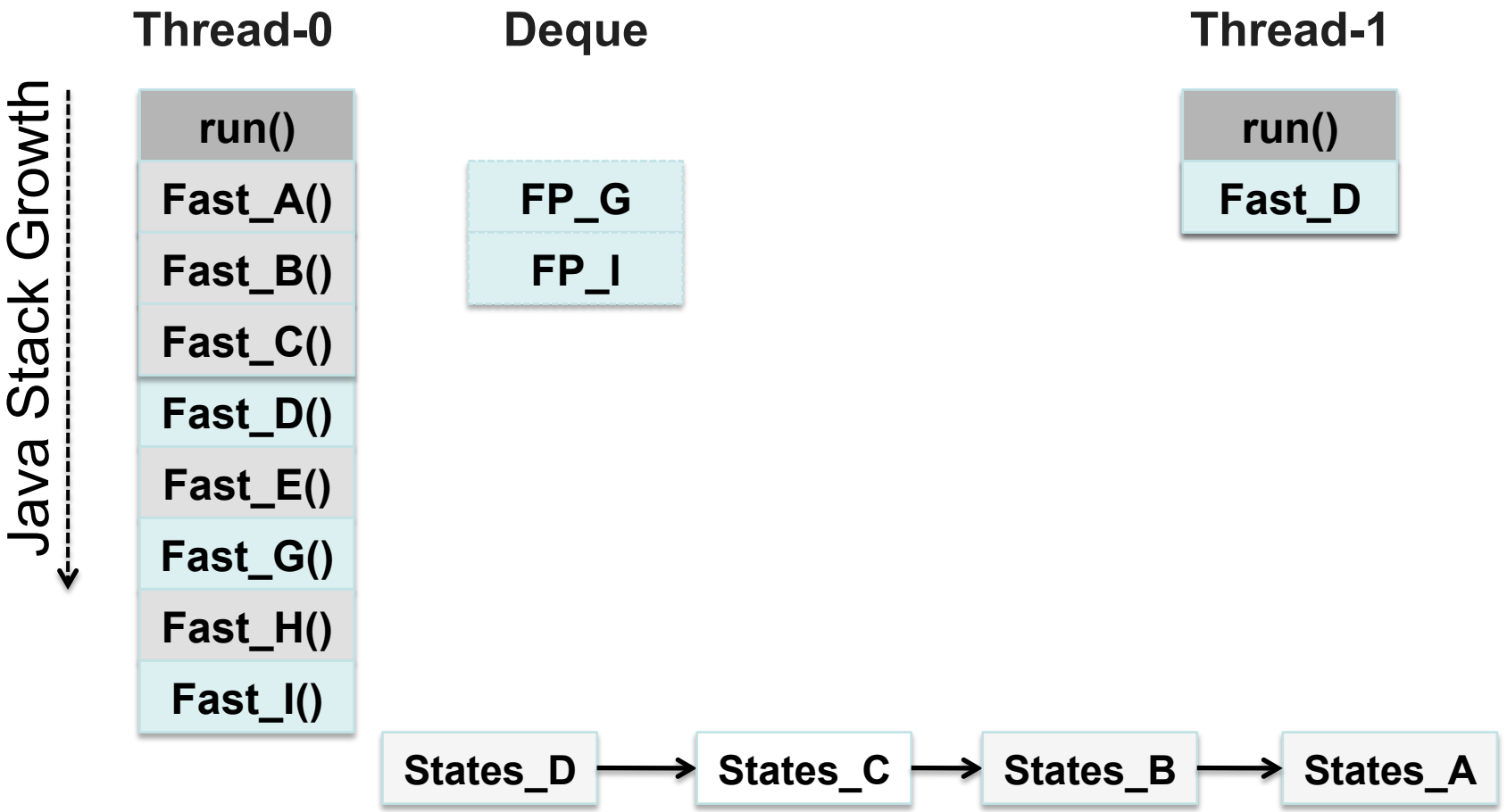
VM Assistance

- Stealing the Stack Frames



VM Assistance

- Stealing the Stack Frames

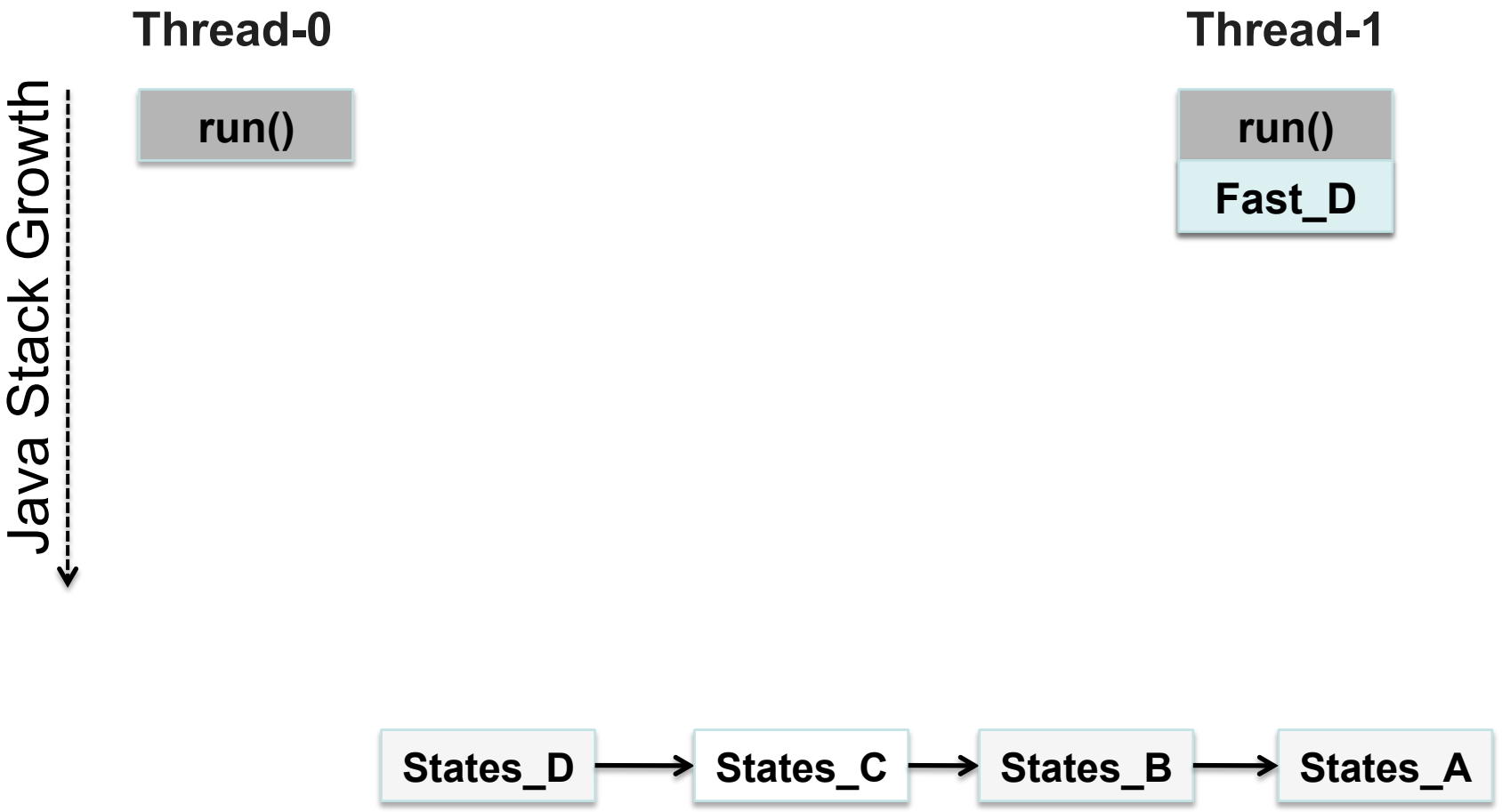


VM Assistance

- Stealing the Stack Frames

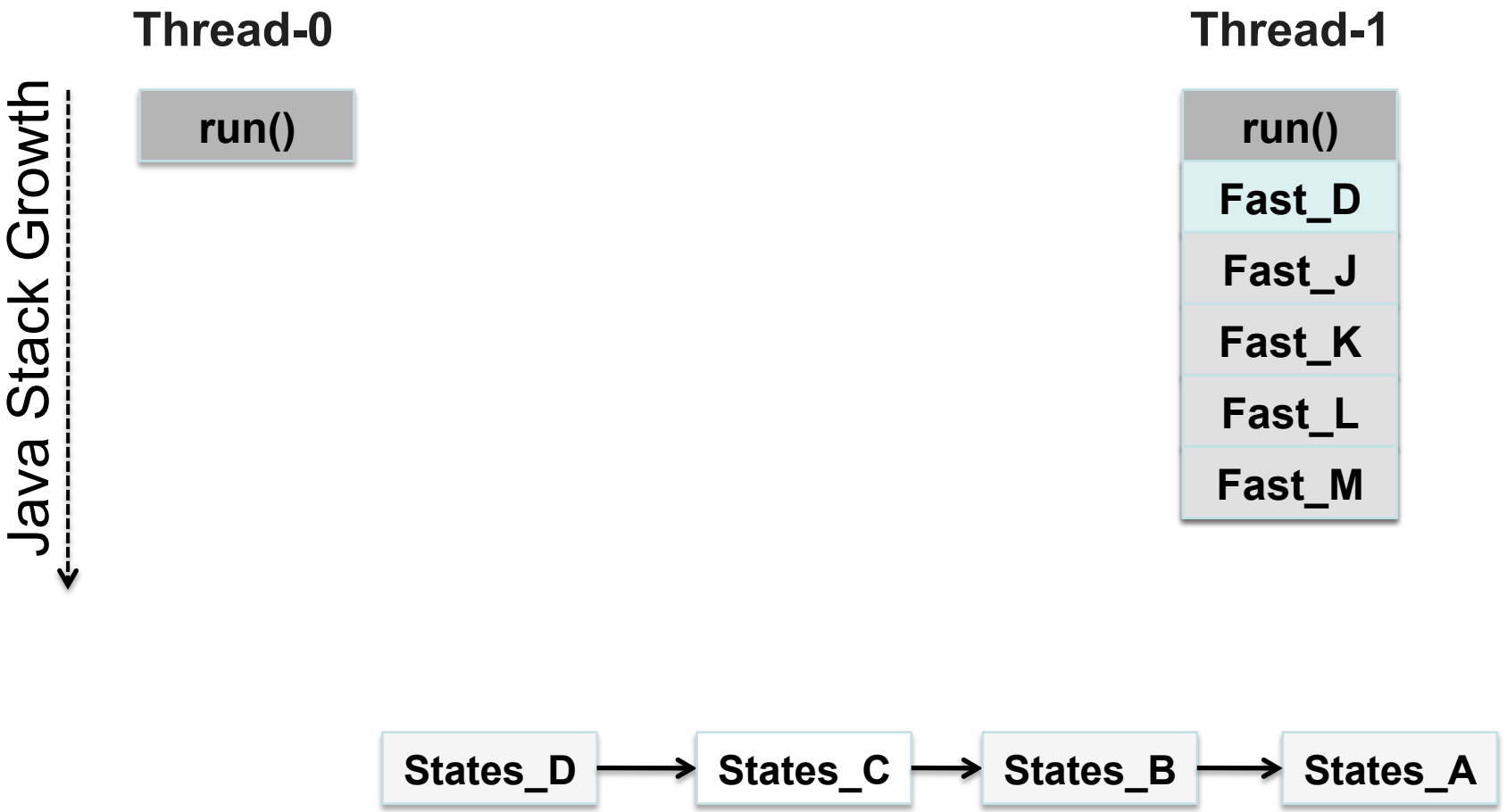
VM Assistance

- Stealing the Stack Frames



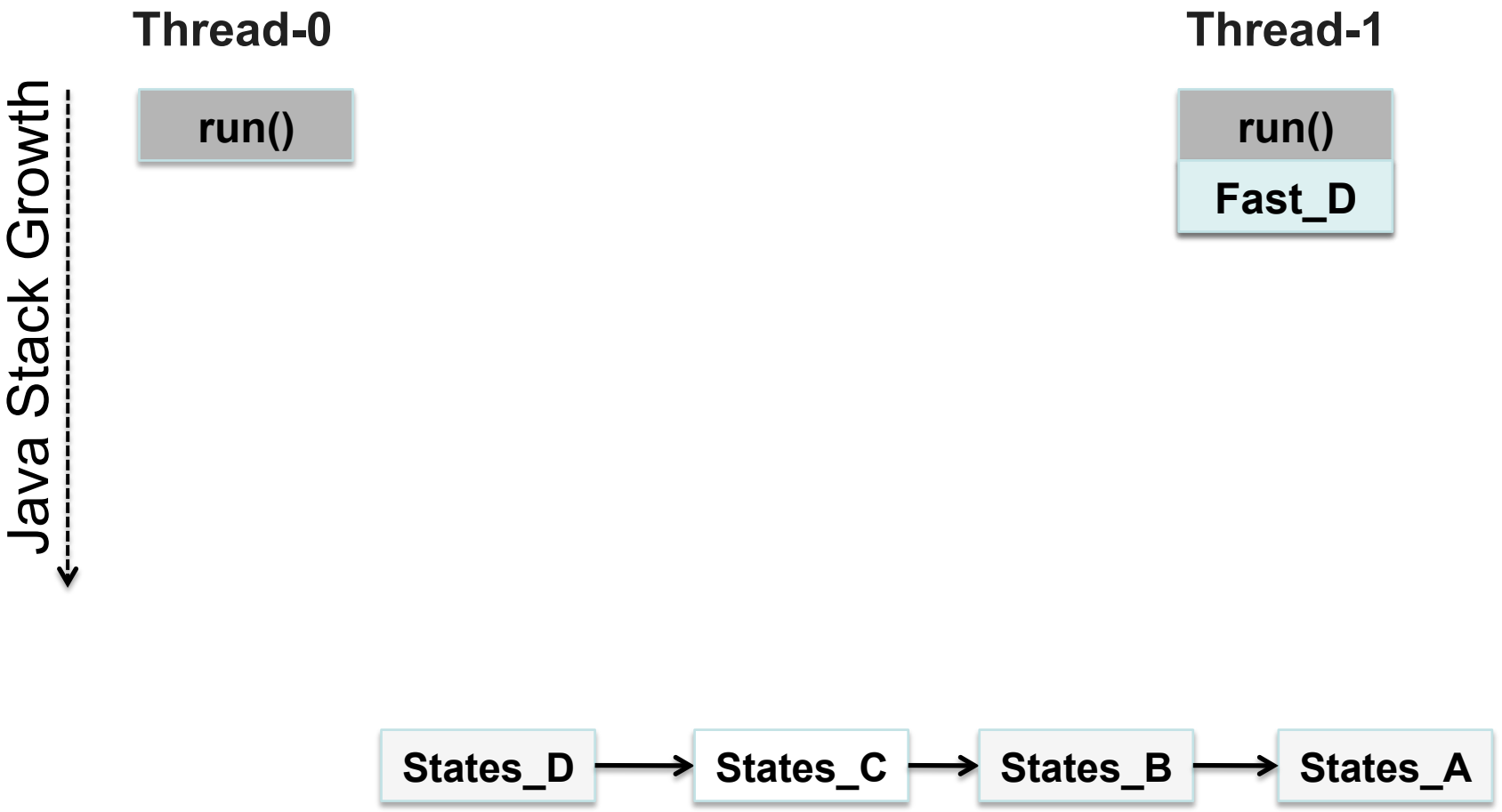
VM Assistance

- Stealing the Stack Frames



VM Assistance

- Stealing the Stack Frames



VM Assistance

- Stealing the Stack Frames

VM Assistance

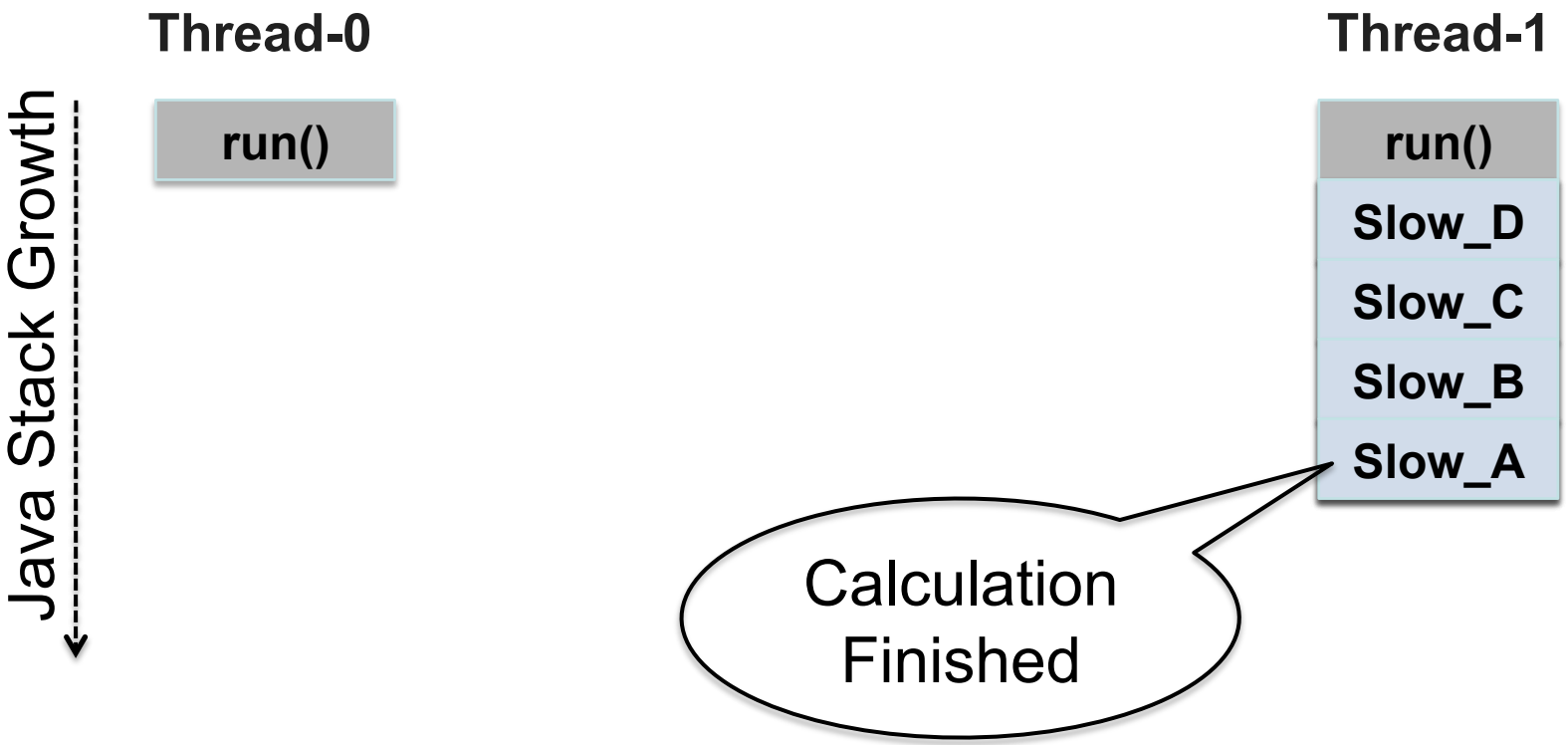
- Stealing the Stack Frames

VM Assistance

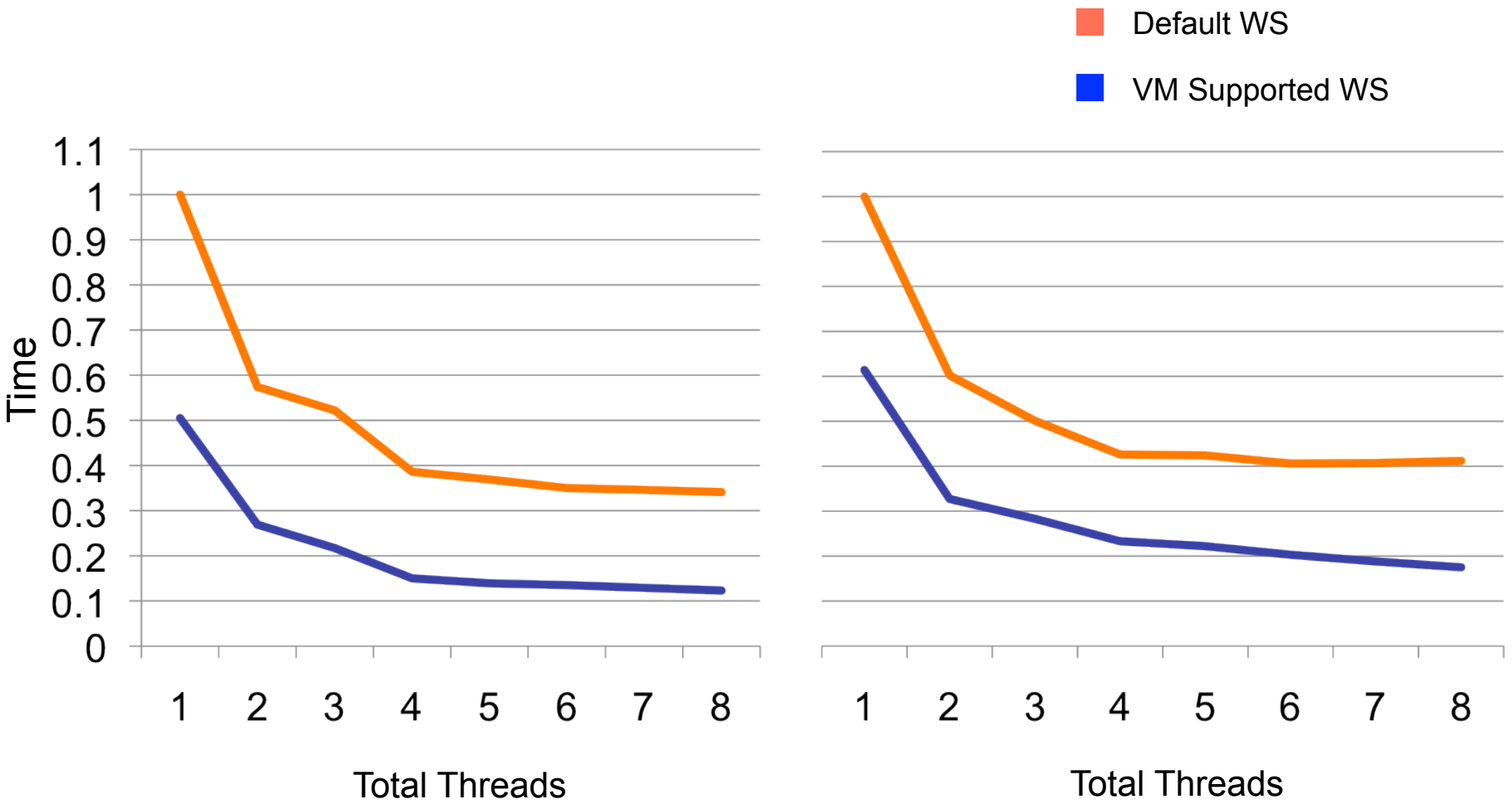
- Stealing the Stack Frames

VM Assistance

- Stealing the Stack Frames



Experimental Results – Execution Time

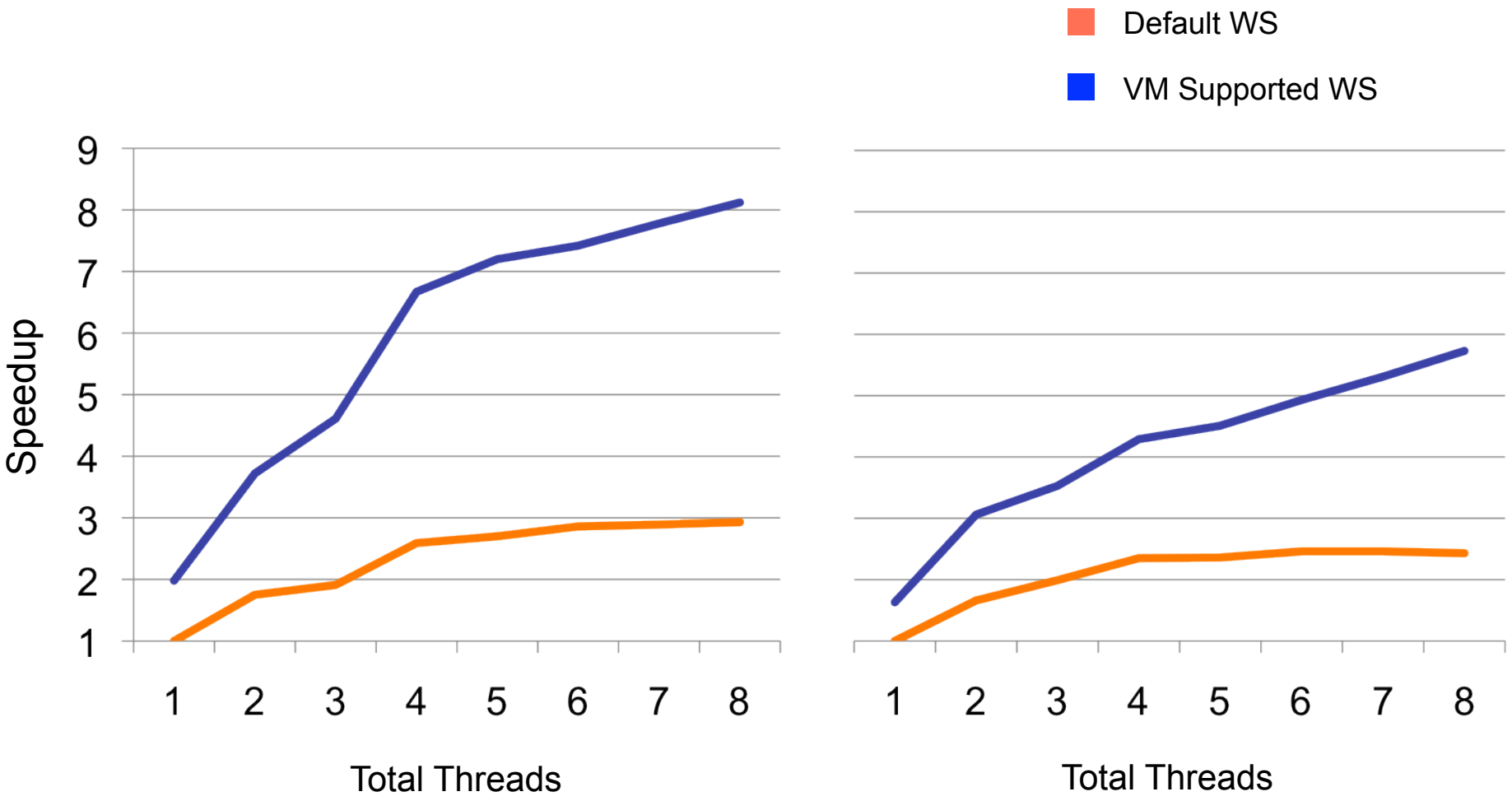


a) Integrate (1000)

b) Fib (40)

Execution Time Normalized With Single Thread Default Work-Stealing Time

Experimental Results – Speedup



a) Integrate (1000)

b) Fib (40)

Speedup Relative to Single Thread Default Work-Stealing Time

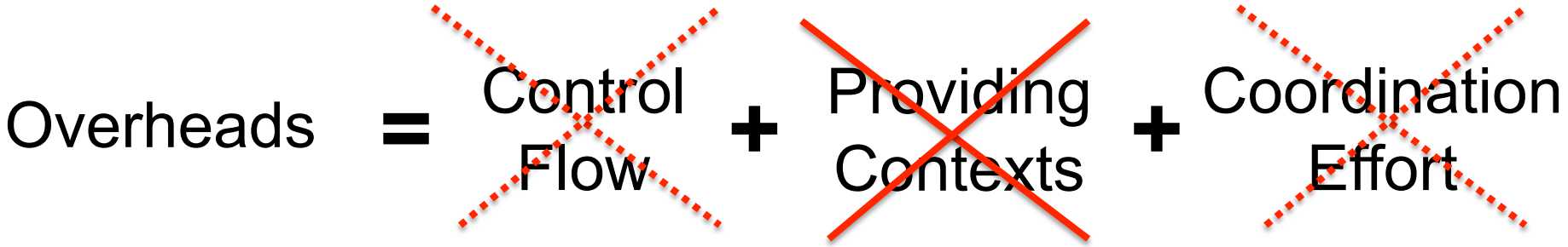
Summary

- Multicore era
  - Dynamic task parallelism.
- Load balancing
  - Work–stealing schedulers

Overheads = Control Flow + ~~Providing Contexts~~ + Coordination Effort

Future Work

- Multicore era
  - Dynamic task parallelism.
- Load balancing
  - Work–stealing schedulers



➢ Test with high steal ratio benchmarks

➢ Research new VM extensions to make X10 run faster.

Questions …..?