# CSE502: Foundations of Parallel Programming

## Lecture 16: Java TryCatch Work-Stealing

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

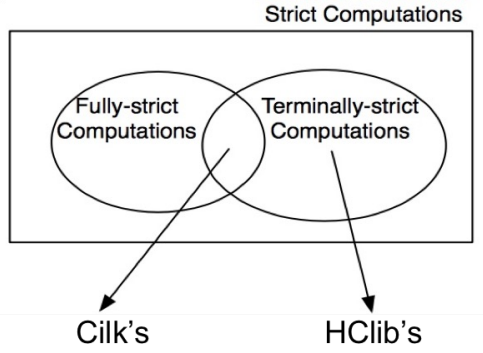
vivekk@iiitd.ac.in

# Last Class

```
cilk uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
  }
}
cilk int main(int argc, char** argv) {
  int result = spawn fib(40);
  sync;
}
```

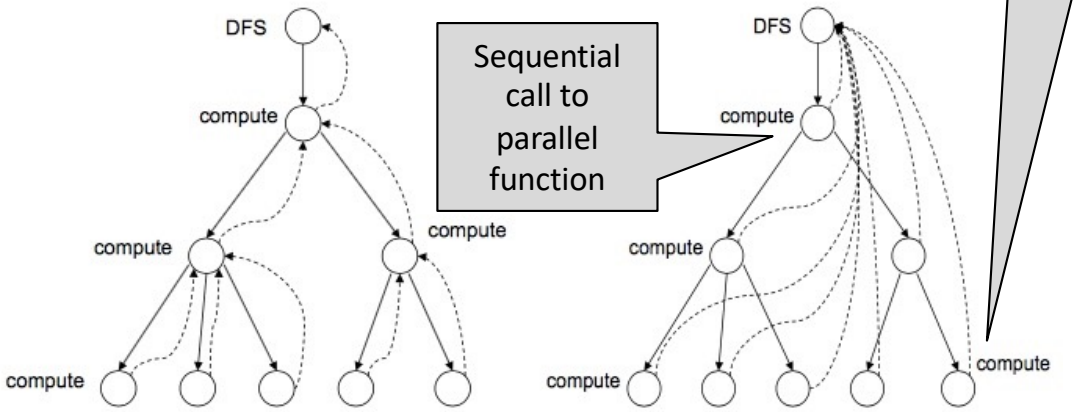
No data race on **local** variable x!

```
cilk uint64_t fib(uint64_t n) {
  uint64_t x = 0;
  inlet void summer(uint64_t result) {
    x += result;
    return;
  }
  if(n<2) {
    return n;
  } else {
    summer(spawn fib(n-1));
    summer(spawn fib(n-2));
    sync;
    return x;
  }
}
```




Escaping async –> a child can outlive its parent, i.e. avoiding unnecessary synchronization

Sequential call to parallel function



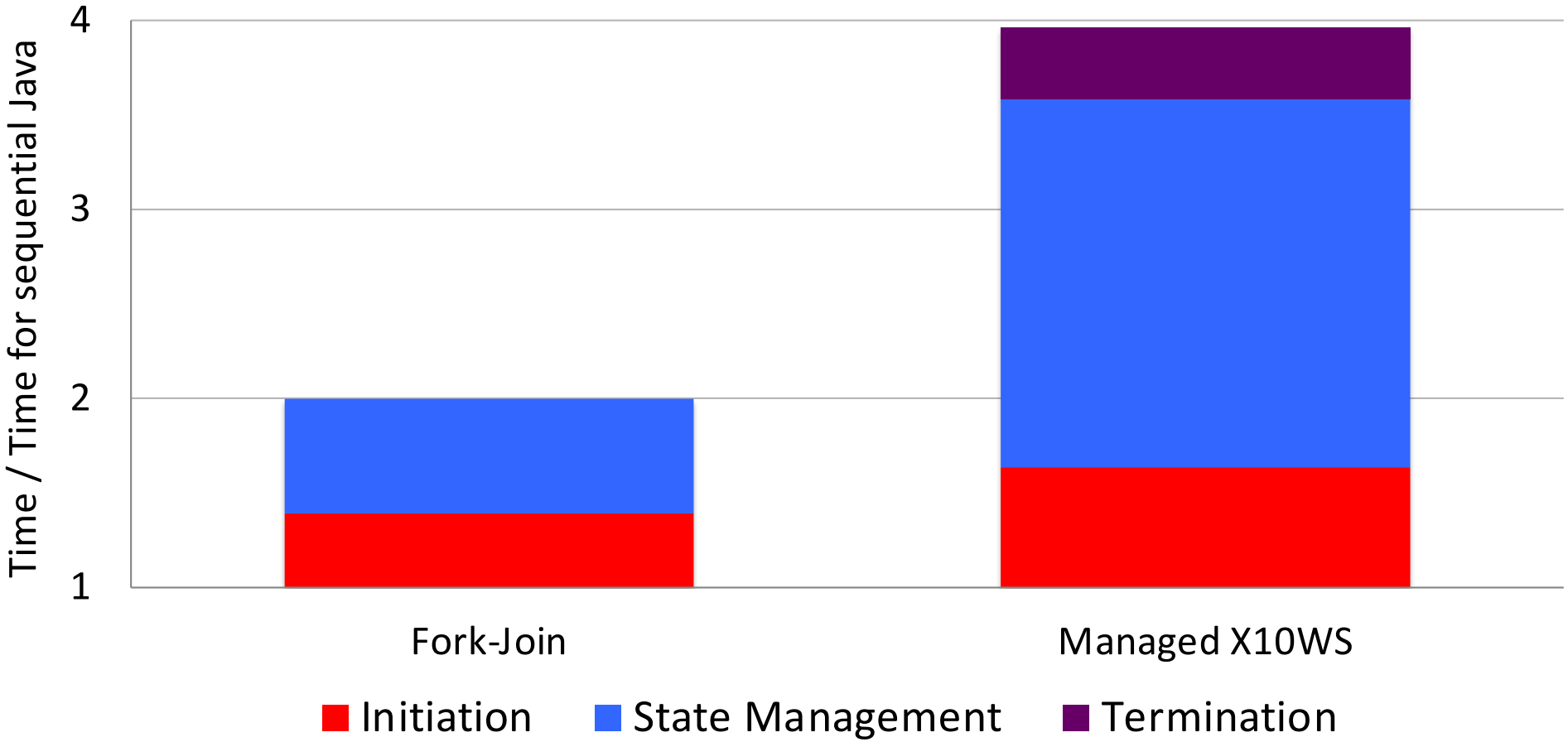

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    if (p == 0) {
      abort; /* Aborts existing children, */
    }        /* but not future ones.      */
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    if (p == 0) { /* Add check for future */
      return 0;   /* children            */
    }
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

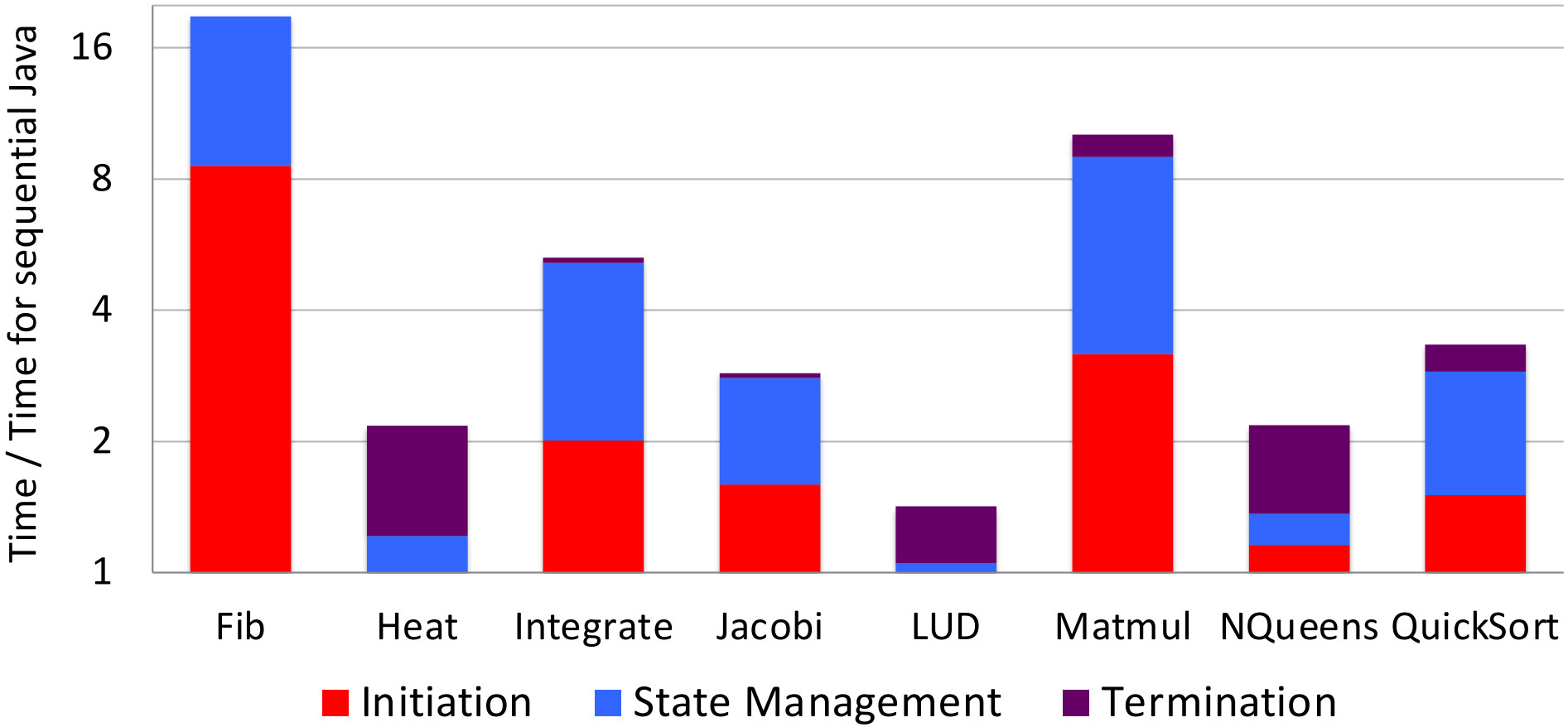
# Today's Class

- Solutions for Lab-3 and Lab-4
- Introduction to Java TryCatch Work-Stealing
- Quiz-4

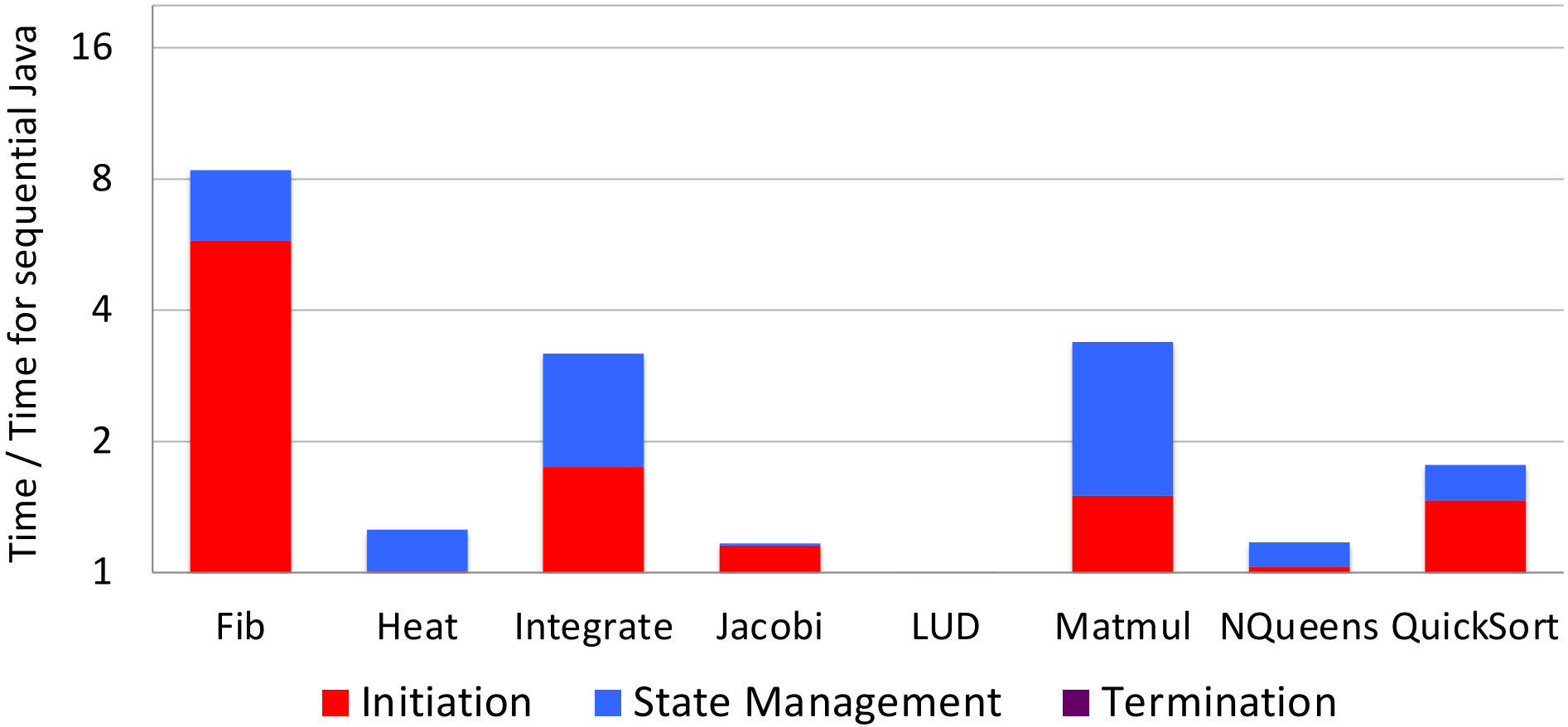
# Motivational Analysis
# Sequential Overhead



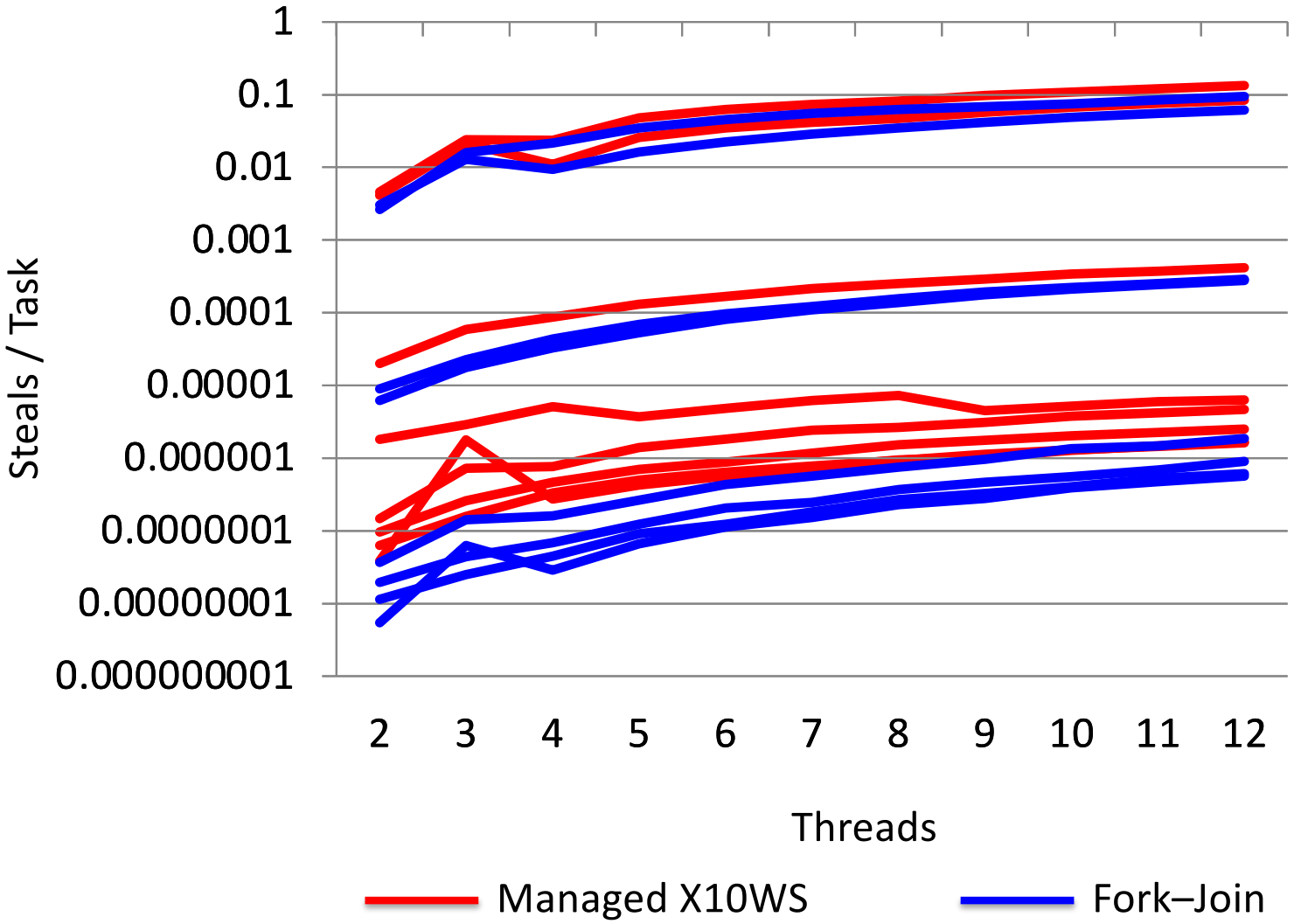
# Motivational Analysis
# Sequential Overhead: Managed X10WS



# Motivational Analysis
# Sequential Overhead: Fork–Join



# Motivational Analysis
# Steal Ratio



# Motivational Analysis Summary

- 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

# Fibonacci in TryCatchWS

```
public class Fibonacci {
    int n;
    public Fibonacci(int _n) {
        n = _n;
    }
    private int fib(int n) {
        if(n<2) return n;
        int x, y;
        x = fib(n-1);
        y = fib(n-2);
        return x + y;
    }
    public int compute() {
        return fib(n);
    }
    ......
}
```

Sequential

```
import java.util.concurrent.*;

public class Fibonacci
                    extends RecursiveTask<Integer> {
    int n;
    public Fibonacci(int _n) { n=_n; }

    public Integer compute() {
        if(n<2) return n;

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```

Java fork/join

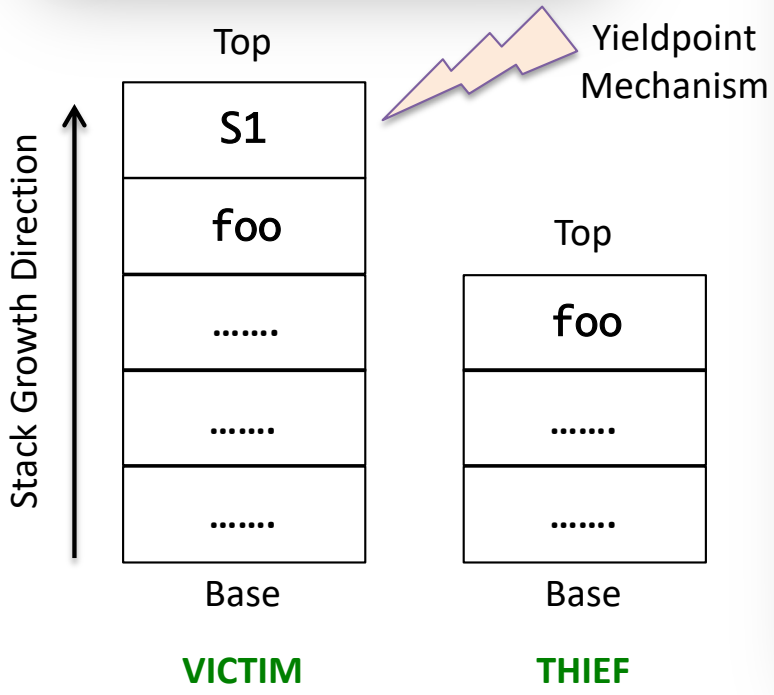
```
public class Fibonacci {
    int n;
    public Fibonacci(int _n) {
        n = _n;
    }
    private int fib(int n) {
        if(n<2) return n;
        int x, y;
        finish {
            async {
                x = fib(n-1);
                y = fib(n-2);
            }
        }
        return x + y;
    }
    public int compute() {
        return fib(n);
    }
    ......
}
```

TryCatchWS

- TryCatchWS support serial elision

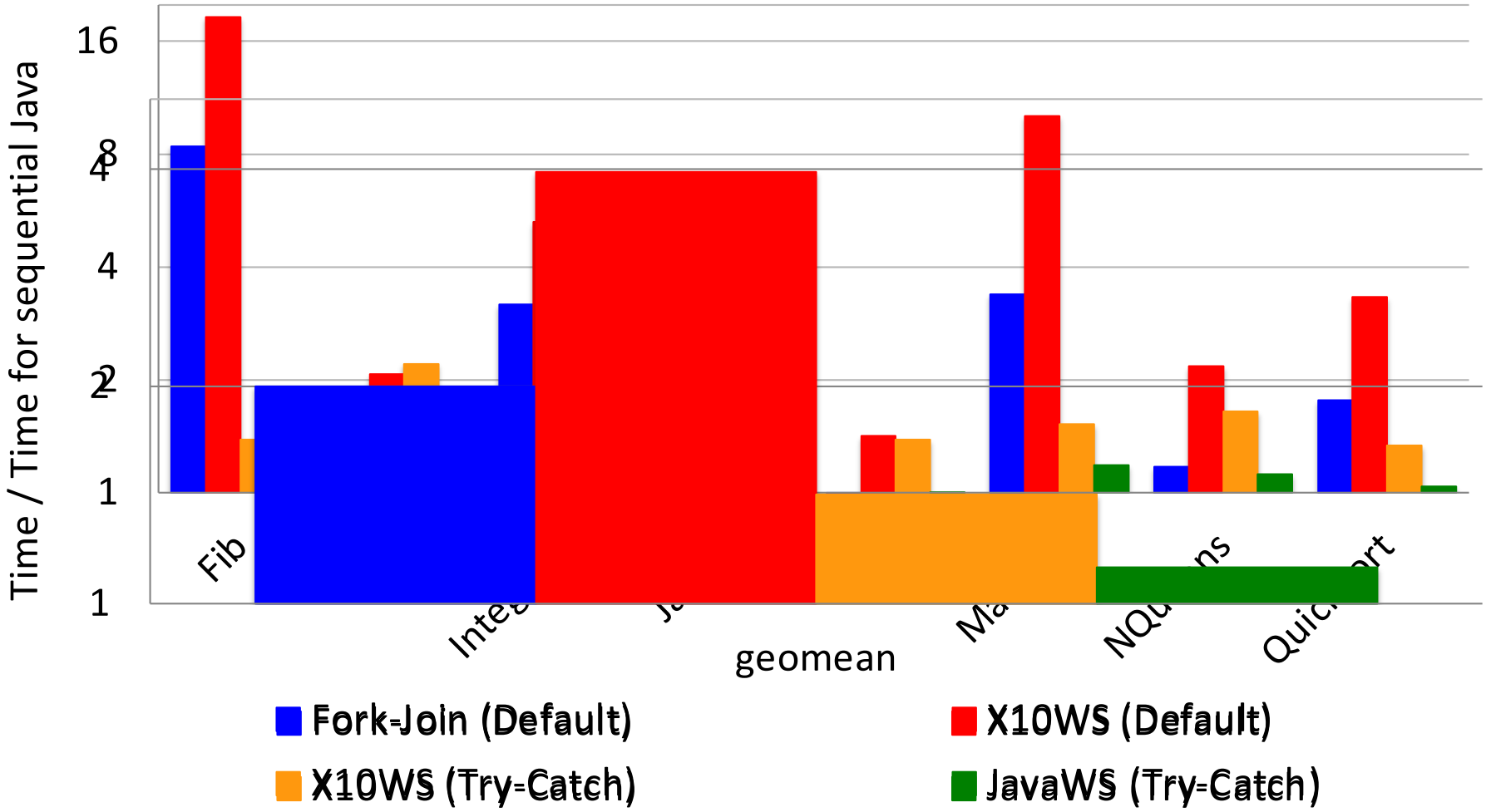
# TryCatchWS Implementation

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

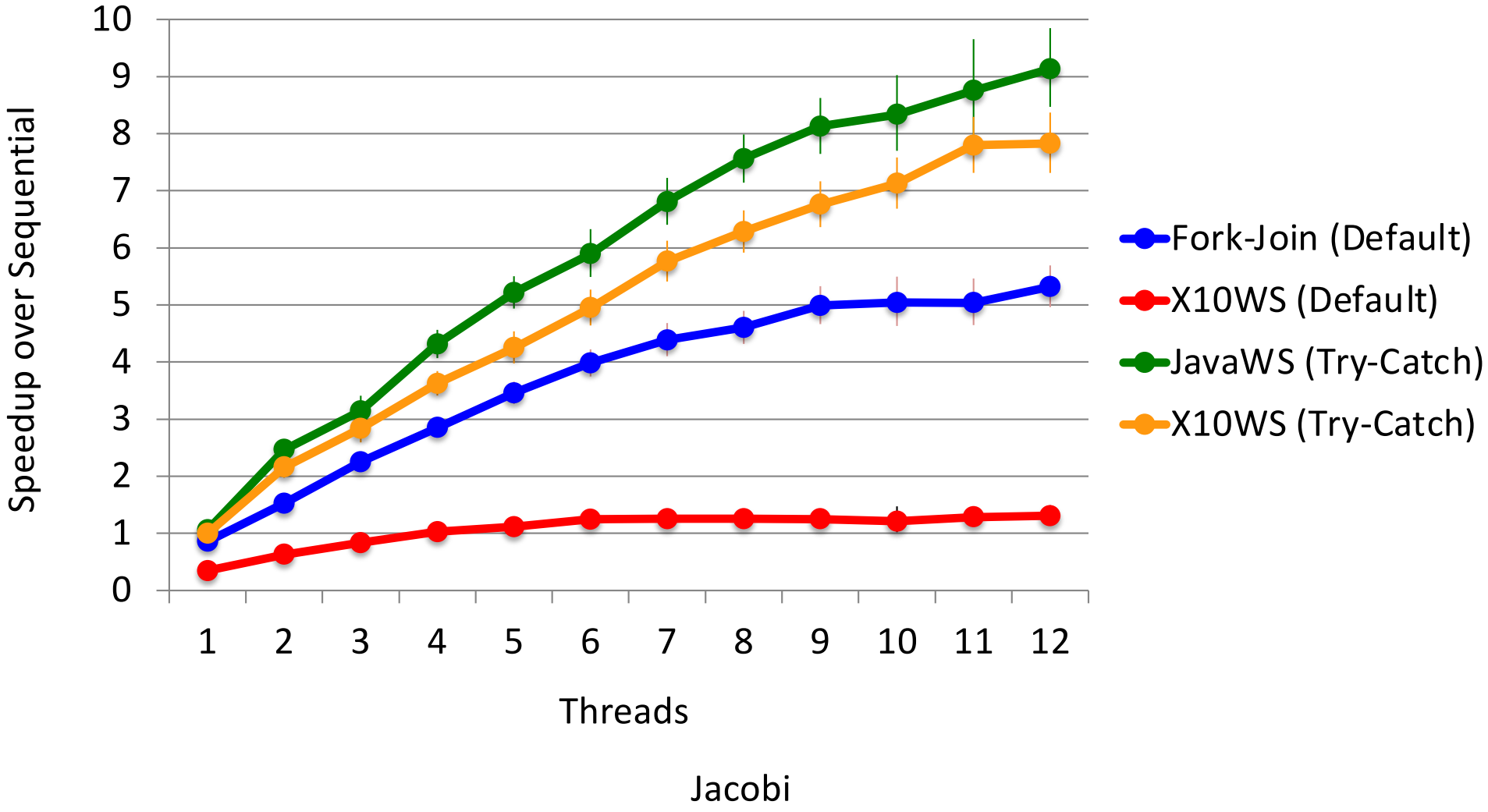



```
try {
    try {
        Runtime.continuationAvailable()
        X = S1();
        // if S2 stolen then throw ExceptionVictim
    } catch (ExceptionVictim v) {
        // 1. Runtime stores result X
        // 2. if S2 not complete, become a Thief
        // 3. else throw ExceptionFetch
    } catch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

# Results: Sequential Overhead



# Results: Work–Stealing Performance



# Next Class

- Introduction to OpenMP programming model