Friendly Barriers: Efficient Work-Stealing
With Return Barriers

Vivek Kumar¹, Stephen M Blackburn¹, David Grove²

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

```scala
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 Without The Baggage  |  Kumar et al.| OOPSLA’12
Initiation

State Management

Termination
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, …
Our Prior Work

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
Our Prior Work

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

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

Our Prior Work

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

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

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

---

**Our Stack Growth Direction**

- **S1**
  - foo
  - C
  - B
  - A

  **THIEF**
  - steal
  - ....
  - A

  **VICTIM**
  - foo
  - C
  - B
  - A

---

Friendly Barriers: Efficient Work-Stealing With Return Barriers | Kumar et al. | VEE 14
Our Prior Work

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

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

![Stack Growth Direction](image)

VICTIM

<table>
<thead>
<tr>
<th>S1</th>
<th>foo</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
</table>

THIEF

<table>
<thead>
<tr>
<th>S2</th>
<th>foo</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
</table>
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}

Our Prior Work

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

Stack Growth Direction

S1
- foo
- C
- B
- A

S2
- foo
- C
- B
- A

VICTIM

THIEF
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

![Graph showing Steals To Task Ratio with different lines representing Threads and a legend indicating 1 in 10 stolen and 1 in 10,000 stolen.]

Friendly Barriers: Efficient Work-Stealing With Return Barriers | Kumar et al. | VEE 14
Steal Rate

![Steal Rate Graph](#)
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
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

Stack Growth Direction

TOP
E
D
C
B
A
BASE

Implementation

Yieldpoint mechanism
Victim Moves The Return Barrier
Victim Moves The Return Barrier
Robbing A Victim With Return Barrier
Performance Evaluation
Dynamic Overhead

Threads = 16

Dynamic Overhead (%)

Jacobi  FFT  CilkSort  Barnes-Hut  UTS  LUD

DefaultWS  ReturnBarrierWS
Performance Benefit Relative to DefaultWS

Threads = 16

ReturnBarrierWS

Jacobi  FFT  CilkSort  Barnes-Hut  UTS  LUD
Free Steals From Return Barrier

![Graph showing Total Free Steals (%) vs Threads for different benchmarks: Jacobi, FFT, CilkSort, Barnes-Hut, UTS, and LUD. The graph indicates how the percentage of free steals changes as the number of threads increases for each benchmark.]
Evaluation

Overhead of Executing Return Barrier

- Return Barrier Overhead (%)
- Threads

Graph showing the overhead of executing return barriers for different benchmarks with varying thread counts, including:
- Jacobi
- FFT
- CilkSort
- Barnes-Hut
- UTS
- LUD

Friendly Barriers: Efficient Work-Stealing With Return Barriers | Kumar et al. | VEE 14
Comparative Performance

![Graph showing speedup over sequential for different methods in Jacobi]

- **ReturnBarrierWS**
- **Fork-Join**
- **Habanero-Java**

**Threads**

**Speedup over Sequential**
Comparative Performance

![Graph showing speedup over sequential for different thread counts with labels: ReturnBarrierWS, Fork-Join, Habanero-Java. The graph compares performance on UTS.](image)
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%