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

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 Without The Baggage  |  Kumar et al.| OOPSLA’12
Work–Stealing
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

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

Stack Growth Direction

- **S1**: Yieldpoint Mechanism
- **foo**: On-stack replacement
- **C**: Java try/catch exceptions
- **B**: Dynamic code patching
- **A**: Stack Growth Direction

**VICTIM**

- **steal**
- ....

**THIEF**
Our Prior Work

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

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

VICTIM

THIEF

THIEF

Stack Growth Direction
Our Prior Work

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

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

Our Prior Work

- 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

![Graph showing Steals To Task Ratio with lines for different numbers of threads and two annotation points: 1 in 10 stolen and 1 in 10,000 stolen.]

DefaultWS

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

![Graph showing Steal Rate with Threads on the x-axis and Steals per milli-seconds on the y-axis. The graph includes a line labeled DefaultWS.](image-url)
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

Implementation

Frame Pointer

Frame Address

Frame C
Method C

Stack Growth Direction

TOP
Frame D

BASE

A
B
C
D
E
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

![Diagram showing stack growth direction and yieldpoint mechanism]
Victim Moves The Return Barrier
Victim Moves The Return Barrier

![Diagram showing stack growth direction and barrier movement.](Image)
Robbing A Victim With Return Barrier

Yieldpoint mechanism
Performance Evaluation
Dynamic Overhead

Evaluation

Dynamic Overhead (%)

Threads = 16

Friendly Barriers: Efficient Work-Stealing With Return Barriers | Kumar et al. | VEE 14
Performance Benefit Relative to DefaultWS

Threads = 16

Performance Benefit

Jacobi | FFT | CilkSort | Barnes-Hut | UTS | LUD

ReturnBarrierWS
Free Steals From Return Barrier

![Graph showing the percentage of total free steals for different algorithms across various threads. The x-axis represents the number of threads, ranging from 2 to 16. The y-axis represents the total free steals percentage. The graphs include lines for algorithms such as Jacobi, FFT, CilkSort, Barnes-Hut, UTS, and LUD. The graph illustrates how the percentage of total free steals changes as the number of threads increases.]
Evaluation

Overhead of Executing Return Barrier

![Graph showing the overhead of executing return barrier for different work-stealing algorithms. The x-axis represents the number of threads, ranging from 2 to 16, and the y-axis represents the return barrier overhead in percentage. The graph includes lines for algorithms like Jacobi, FFT, CilkSort, Barnes-Hut, UTS, and LUD.](image-url)
Comparative Performance

![Graph showing speedup over sequential for different methods with varying threads. The graph compares ReturnBarrierWS, Fork-Join, and Habanero-Java methods. The y-axis represents speedup, and the x-axis represents the number of threads.](image-url)

Jacobi
Comparative Performance

Speedup over Sequential

UTS

- ReturnBarrierWS
- Fork-Join
- Habanero-Java

Threads
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%