Back to the Futures:
Incremental Parallelization of Existing Sequential Runtimes

James Swaine¹    Kevin Tew²    Peter Dinda¹
Robert Bruce Findler¹    Matthew Flatt²

¹Northwestern University
²University of Utah
Slow-Path Barricading

- Incremental
- Seq. performance intuition carries over
- Low development investment
- Good scaling (negligible sequential overhead)
An Observation

- Runtime “fast-path” operations generally have few side effects
- Thus, safe for parallelism
Slow-Path Barricading

Partition operations into 3 categories:
- Safe (run in parallel)
- Unsafe (runtime side effects)
- A few others (a priori unsafe, but important)

Safety may be dependent on arguments
Slow-Path Barricading

- **One runtime thread** where everything is safe
- Barricades active on all other threads:
  - Detect and intercept unsafe ops
  - Halt a thread until unsafe op can be completed by runtime thread
- Add primitives allowing programmer to explicitly donate the runtime thread’s time to the barricaded thread, allowing it to pass through and continue
Racket Language Extension

\[
\text{future} : (\rightarrow \alpha) \rightarrow \alpha \quad \text{future}
\]

\[
\text{touch} : \alpha \text{ future} \rightarrow \alpha
\]
(let ([f1 (future
          (λ () (+ 1 2)))]
       [f2 (future
            (λ () (+ 3 4)))])
       (+ (touch f1) (touch f2)))

<table>
<thead>
<tr>
<th>Time</th>
<th>Main thread</th>
<th>Future f1</th>
<th>Future f2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time 1</td>
<td>spawn f1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time 2</td>
<td>spawn f2</td>
<td>+ 1 2</td>
<td></td>
</tr>
<tr>
<td>Time 3</td>
<td>touch f1</td>
<td>3</td>
<td>+ 3 4</td>
</tr>
<tr>
<td>Time 4</td>
<td>touch f2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Time 5</td>
<td>+ 3 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(let ([f1 (future
    (λ ()
      (printf "Hello!\n")
      (+ 1 2))])
  [f2 (future (λ () (+ 3 4)))]
  (+ (touch f1) (touch f2)))
Racket Implementation

- Racket runtime:
  - Substrate for the Racket language
  - 100,000+ lines of C code
  - Simple, eager JIT compiler
- Global data includes:
  - Execution state (exception handlers)
  - Symbol table
  - Macro expansion caches
  - GC metadata
# Racket Operations

<table>
<thead>
<tr>
<th>Safe</th>
<th>Unsafe</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>allocation</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>JIT compilation</td>
</tr>
<tr>
<td>unsafe-fl+</td>
<td>hash-set!</td>
<td></td>
</tr>
<tr>
<td>unsafe-fl/</td>
<td>printf</td>
<td></td>
</tr>
<tr>
<td>unsafe-vector-ref</td>
<td>vector-ref</td>
<td></td>
</tr>
<tr>
<td>unsafe-vector-set!</td>
<td>printf</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>call/cc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>open-input-file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>error</td>
</tr>
</tbody>
</table>
Barricades in Racket

- All code JIT compiled (if possible)
- Fast-path ops - inlined
- Slow-path ops - C functions
“Other” Operations

- We leverage Racket’s user-level thread infrastructure for:
  - Allocation
  - JIT compilation
  - Racket threads: preemptive to programmers, cooperative to runtime
  - Cooperation points allow for polling
Garbage Collection

- GC = special form of synchronized operation (stop the world)

- Cooperation points become barriers
Slow-Path Barricading

☑ Incremental
☑ Seq. performance intuition carries over
☐ Low development investment
☐ Good scaling (negligible sequential overhead)
Development Person-Hours (Racket)

- Performed by non-expert (no prior knowledge) and runtime developer

<table>
<thead>
<tr>
<th></th>
<th>Expert</th>
<th>Non-Expert (me)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>41</td>
<td>536</td>
<td>577</td>
</tr>
</tbody>
</table>
Parrot Implementation

- Parrot runtime:
  - Register-based virtual machine
  - Pluggable runloop allows switching between interpreters
  - Dynamic (virtual functions)
  - Each bytecode is checked prior to execution for safety
  - Includes argument checking
Development Person-Hours (Racket)

Performed by expert (active runtime implementation contributor)

<table>
<thead>
<tr>
<th></th>
<th>Expert</th>
<th>Non-Expert</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>52</td>
<td>-</td>
<td>52</td>
</tr>
</tbody>
</table>
Performance Evaluation

- 3 microbenchmarks
  - Signal convolution
  - Mergesort
  - Sparse matrix-vector multiplication
- 2 NAS Parallel Benchmarks kernels
  - Integer Sort
  - Fourier Transform
- 2 test machines:
  - 8-core workstation (Mac OS X)
  - 16-core mid-range server (Linux)
<table>
<thead>
<tr>
<th></th>
<th>Racket 8-core Mac</th>
<th>Racket 16-core Linux</th>
<th>Parrot 8-core Mac</th>
<th>Parrot 16-core Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Mergesort</td>
<td>8</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mat-Vec Mult.</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Integer Sort</td>
<td>6</td>
<td>Never</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fourier Transform</td>
<td>Never</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Scaling**

- **Good**
- **Poor**

**Values** = # of threads to beat sequential impl.

Friday, March 18, 2011
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Racket 8-core Mac</th>
<th>Racket 16-core Linux</th>
<th>Parrot 8-core Mac</th>
<th>Parrot 16-core Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mergesort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mat-Vec Mult.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fourier Transform</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Scaling**
- Good
- Poor

**NAS Fourier Transform**

**Thread Count**

**Seconds**

**Friday, March 18, 2011**
### Parrot Signal Convolution

#### Thread Count vs. Seconds

- **Sequential**
- **Sequential Parrot**

#### Performance Metrics

- **Scaling**
  - Good
  - Poor

#### Data Points

- **Convolution**
  - 3
- **Mergesort**
  - 1
- **Mat-Vec Mult.**
  - 1
- **Integer Sort**
  - 1
- **Fourier Transform**
  - 1

#### Additional Information

- **Thread Count**
  - 0, 2, 3, 4, 5, 6, 7, 8

- **Seconds**
  - 0, 100, 200, 300, 400

- **Friday, March 18, 2011**

---

**Figure Notes:**
- Graph showing performance scaling for Parrot Signal Convolution across different thread counts and seconds.
- Comparison of sequential performance with parallel execution on Mac and Linux.
- Evaluation criteria: Good, Poor scaling.

---

**Table Note:**
- Detailed comparison table not visible in the image.
Slow-Path Barricading

- Incremental
- Seq. performance intuition carries over
- Low development investment
- Good scaling (negligible sequential overhead)
Thanks!

- Try parallel Racket today: http://racket-lang.org/download/
- Try slow path barricading in your runtime system; the main system developer should be able to add it within a few weeks of work