Towards An Adaptive Framework for Performance Portability
Work in Progress (submission #23)

Patrick Maier    Magnus Morton    Phil Trinder

School of Computing Science
University of Glasgow

IFL 2015
**Performance Portability?**

**Write once, run anywhere (a.k.a. the holy grail of portability)**

<table>
<thead>
<tr>
<th>Decade</th>
<th>Type of Application</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970s</td>
<td>portable operating systems</td>
<td>C</td>
</tr>
<tr>
<td>1990s</td>
<td>portable web applications</td>
<td>Java</td>
</tr>
<tr>
<td>2010s</td>
<td>portable linear algebra kernels</td>
<td>OpenCL</td>
</tr>
</tbody>
</table>
Performance Portability?

Write once, run anywhere (a.k.a. the holy grail of portability)

<table>
<thead>
<tr>
<th>Decade</th>
<th>Category</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970s</td>
<td>portable operating systems</td>
<td>C</td>
</tr>
<tr>
<td>1990s</td>
<td>portable web applications</td>
<td>Java</td>
</tr>
<tr>
<td>2010s</td>
<td>portable linear algebra kernels</td>
<td>OpenCL</td>
</tr>
</tbody>
</table>

It is accepted that portability incurs a performance hit.

- C compilers are very mature; performance hit vs. assembly is small.
- JIT compilation has brought Java performance within reach of C++.
- OpenCL code can perform as well as hand-written kernels.
Performance Portability?

Write once, run anywhere (a.k.a. the holy grail of portability)

<table>
<thead>
<tr>
<th>Decade</th>
<th>Portability Area</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970s</td>
<td>Portable operating systems</td>
<td>C</td>
</tr>
<tr>
<td>1990s</td>
<td>Portable web applications</td>
<td>Java</td>
</tr>
<tr>
<td>2010s</td>
<td>Portable linear algebra kernels</td>
<td>OpenCL</td>
</tr>
</tbody>
</table>

It is accepted that portability incurs a performance hit.

- C compilers are very mature; performance hit vs. assembly is small.
- JIT compilation has brought Java performance within reach of C++.
- OpenCL code can perform as well as hand-written kernels.

But: Performance of OpenCL code very sensitive to architecture.

- Necessary architecture-specific tuning defeats portability.
Performance Portability?

Write once, run anywhere (a.k.a. the holy grail of portability)

<table>
<thead>
<tr>
<th>Decade</th>
<th>Domain</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970s</td>
<td>portable operating systems</td>
<td>$\sim C$</td>
</tr>
<tr>
<td>1990s</td>
<td>portable web applications</td>
<td>$\sim$ Java</td>
</tr>
<tr>
<td>2010s</td>
<td>portable linear algebra kernels</td>
<td>$\sim$ OpenCL</td>
</tr>
</tbody>
</table>

It is accepted that portability incurs a performance hit.

- C compilers are very mature; performance hit vs. assembly is small.
- JIT compilation has brought Java performance within reach of C++.
- OpenCL code can perform as well as hand-written kernels.

**But:** Performance of OpenCL code very sensitive to architecture.

- Necessary architecture-specific tuning defeats portability.

Performance Portability

Same parallel code runs across different architectures with reasonable efficiency.
Grand Vision

Idea: Combine trace-based JIT compiler with demand-driven parallel scheduler.

- Language: Racket + skeleton library
Grand Vision

**Idea:** Combine trace-based JIT compiler with demand-driven parallel scheduler.
- Language: Racket + skeleton library

**Slogan:** Compile once, run anywhere *in parallel*.
- Performance portability hypothesis to be tested
  - by benchmarking problems with irregular parallelism
  - on several (CPU-centric) architectures (desktop, NUMA server, small cluster).
Grand Vision

**Idea:** Combine trace-based JIT compiler with demand-driven parallel scheduler.
- Language: Racket + skeleton library

**Slogan:** Compile once, run anywhere *in parallel*.
- Performance portability hypothesis to be tested
  - by benchmarking problems with irregular parallelism
  - on several (CPU-centric) architectures (desktop, NUMA server, small cluster).

**Some technical details:**
- **Focus** parallelisation where it matters.
  - Using JIT compiler’s hot code detection.

- Estimate task granularity by online *profiling* and/or *static analysis* of traces.
  - Linear structure of traces enables cheap yet accurate analyses.

- Adapt task granularity by online *code transformation*.
  - Rewriting according to programmable rules expressing semantic equivalences.
Grand Vision — Functional Block Diagram

**Execution Engine (a)**
* interpret code until compiled trace or hot loop
* exec/prof compiled trace
* record hot loop trace

**Trace Compiler (b)**
* optimise and compile recorded trace

**Trace Analyser (c)**
* estimate trace runtime

**Rewrite Engine (d)**
* transform task graph to improve granularity

**Scheduler (e)**
* maintain task graph
* schedule ready tasks
* balance load by moving tasks between schedulers

---

Maier, Morton, Trinder (Glasgow) Towards ... Performance Portability
Language and Compiler

**Language**: Racket (Scheme dialect)

- dynamically typed, strict functional language
- elaborate macro system
- concurrency, shared-memory parallelism, distributed computation, ...
**Language and Compiler**

**Language:** Racket (Scheme dialect)
- dynamically typed, strict functional language
- elaborate macro system
- concurrency, shared-memory parallelism, distributed computation, ...

<table>
<thead>
<tr>
<th>Compiler</th>
<th>JIT</th>
<th>language support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Racket</td>
<td>function-level</td>
<td>full language</td>
</tr>
<tr>
<td>standard VM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 years development</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pycket</td>
<td>trace-level</td>
<td>DOES NOT SUPPORT</td>
</tr>
<tr>
<td>PyPy-derived VM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 year development</td>
<td></td>
<td></td>
</tr>
<tr>
<td>often beats Racket</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>* concurrency (threads)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* parallelism (futures)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* distributed comp (places)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* exceptions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* sockets</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Scheduler

- Centralised control.
- Actor-like processes (no shared state, single threaded, message passing).

\[ \text{Scheduler} \]

\begin{itemize}
  \item Centralised control.
  \item Actor-like processes (no shared state, single threaded, message passing).
\end{itemize}
Task Graphs

full futures

tasks

empty futures
Task Graphs

full futures

map skeleton

tasks

empty futures

fold skeleton
Task Graphs
**Skeletons**

### Map skeletons

- **par-map** :: Closure (a -> b) -> [a] -> [b]
- **par-map/chunk** :: Int -> Closure ([a] -> [b]) -> [a] -> [b]
- **par-map/stride** :: Int -> Closure ([a] -> [b]) -> [a] -> [b]

### Fold skeletons

- **par-fold** :: Closure ([a] -> a) -> [a] -> a
- **par-fold/depth** :: Int -> Closure ([a] -> a) -> [a] -> a

### Divide and conquer skeletons

- **par-d&c** :: Closure (a->b, a->[a], [b]->b) -> a -> b
- **par-d&c/depth** :: Int -> Closure (a->b, a->[a], [b]->b) -> a -> b
Skeletons Transformations

Skeletons are related by an equational theory.

Some map skeleton equations

1. \( \text{map } f \ $ \text{map } g \ xs \ = \ \text{map } (x \rightarrow f \ $ \ g \ x) \ xs \)
2. \( \text{map } f \ xs \ = \ \text{concat } \ $ \text{map } (\text{map } f) \ $ \text{chunk } k \ xs \)
3. \( \text{map } f \ xs \ = \ \text{par-map } (\text{Closure } f) \ xs \)
4. \( \text{concat } \ $ \text{map } g \ $ \text{chunk } k \ xs \ = \ \text{par-map/chunk } k \ (\text{Closure } g) \ xs \)
Skeletons are related by an equational theory.

Some map skeleton equations

1. $\text{map } f \circ \text{map } g \, xs = \text{map } (x \rightarrow f \circ g \, x) \, xs$
2. $\text{map } f \, xs = \text{concat } \circ \text{map } (\text{map } f) \circ \text{chunk } k \, xs$
3. $\text{map } f \, xs = \text{par-map } (\text{Closure } f) \, xs$
4. $\text{concat } \circ \text{map } g \circ \text{chunk } k \, xs = \text{par-map/chunk } k \, (\text{Closure } g) \, xs$

Equations can be used as bi-directional rewrite rules.

- Instantiate granularity parameter $k$ when applying (2) from left to right.

Sample transformation

\[
\begin{align*}
\text{par-map } (\text{Closure } f) \circ \text{par-map } (\text{Closure } g) \, xs \\
&= \text{map } f \circ \text{map } g \, xs \\
&= \text{map } (x \rightarrow f \circ g \, x) \, xs \\
&= \text{concat } \circ \text{map } (\text{map } (x \rightarrow f \circ g \, x)) \circ \text{chunk } 5 \, xs \\
&= \text{par-map/chunk } 5 \, (\text{Closure } (\text{map } (x \rightarrow f \circ g \, x))) \, xs
\end{align*}
\]
**Skeletons Transformations II**

**Transform task graph** when observed task cost (i.e. runtime) distribution not in target range (10 – 100 milliseconds).

**Transformation strategy:**

1. **Repeatedly**
   - Rewrite task graph according to skeleton equations
     - **randomised** selection of rewrite rules;
     - **cost model guided** instantiation of granularity parameters.
   - Predict costs of rewritten tasks.

2. Select a task graph whose cost distribution falls within target range.

**Compute cost model** on the fly during JITting.

**Use cost model**

- to predict task execution time, and
- to infer suitable values for granularity parameters (e.g. chunk size).
Trace-based Cost Models

Tracing JIT compilers automatically produce
- traces (= sequences of instructions), and
- trace counters.

**Simple cost model** piggybacking on tracing JIT

\[ cost(\text{trace}) = \sum_{\text{inst} \in \text{trace}} cost(\text{inst}) \]

\[ cost(\text{task}) = \sum_{\text{trace} \in \text{task}} \text{count}(\text{trace}) \cdot cost(\text{trace}) \]

Simple cost model **parametric in cost of instructions**.
- “Learn” cost of instructions by training cost model on a Pycket benchmark suite.
Bad news: Cost model not very accurate for comparing whole programs.
Bad news: Cost model not very accurate for comparing whole programs.
Good news: Cost model quite accurate for comparing task transformations.
**Limitations:** Single server (max 24 cores).

<table>
<thead>
<tr>
<th><strong>Microbenchmarks</strong></th>
<th>skeleton</th>
<th>irregular?</th>
<th>comm. volume</th>
<th>C gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>divide/conquer</td>
<td>no</td>
<td>low</td>
<td>3.4×</td>
</tr>
<tr>
<td>SumEuler</td>
<td>parallel map</td>
<td>moderate</td>
<td>low</td>
<td>1.3×</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>parallel map</td>
<td>moderate</td>
<td>moderate</td>
<td>3.2×</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>parallel map</td>
<td>no</td>
<td>high</td>
<td>1.2×</td>
</tr>
</tbody>
</table>
**Evaluating Scheduler**

**Limitations:** Single server (max 24 cores).

<table>
<thead>
<tr>
<th>Microbenchmarks</th>
<th>skeleton</th>
<th>irregular?</th>
<th>comm. volume</th>
<th>C gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>divide/conquer</td>
<td>no</td>
<td>low</td>
<td>3.4×</td>
</tr>
<tr>
<td>SumEuler</td>
<td>parallel map</td>
<td>moderate</td>
<td>low</td>
<td>1.3×</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>parallel map</td>
<td>moderate</td>
<td>moderate</td>
<td>3.2×</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>parallel map</td>
<td>no</td>
<td>high</td>
<td>1.2×</td>
</tr>
</tbody>
</table>

![Graph showing absolute speedup for various tasks](image)

- **Fib 47 (threshold 29)**
- **SumEuler [1 .. 50,000]**
- **Mandelbrot 5000x4000**
- **MatMult 3200x3200**
Impact of Transformations

SumEuler does not scale well because of low task granularity ($\approx 1.6$ ms).

Transformation 1: Split input interval into even chunks.
- Irregular parallelism: scaling very sensitive to task size.
- Top speedup: 15.9 (up from 11.6)
Impact of Transformations

SumEuler does not scale well because of low task granularity ($\approx 1.6$ ms).

Transformation 2: Stride through input interval.
- Fairly regular parallelism: scaling independent of task size.
- Top speedup: 16.4 (up from 11.6)
Summary:
- Scheduler running parallel Racket code in Pycket.
- Skeleton transformations can speedup parallel code.
- Not yet demonstrated: best transformation dependent on architecture.

Current limitations:
- Task graph scheduling not fully implemented.
- Limited to single server architecture.
- High communication/serialisation overheads.

Work in progress:
- Hook cost analysis into JIT compiler.
- Task graph transformation engine.