A JVM for the Barrelfish Operating System
2nd Workshop on Systems for Future Multi-core Architectures (SFMA’12)

Martin Maas (University of California, Berkeley)
Ross McIlroy (Google Inc.)

10 April 2012, Bern, Switzerland
Future multi-core architectures will presumably...
  ▶ ...have a larger numbers of cores
  ▶ ...exhibit a higher degree of diversity
  ▶ ...be increasingly heterogenous
  ▶ ...have no cache-coherence/shared memory

These changes (arguably) require new approaches for Operating Systems: e.g. *Barrelfish, fos, Tessellation*,...

Barrelfish’s approach: treat the machine’s cores as nodes in a distributed system, communicating via message-passing.

**But**: How to program such a system uniformly?

How to exploit performance on all configurations?

How to structure executables for these systems?
Introduction

- **Answer**: Managed Language Runtime Environments (e.g. Java Virtual Machine, Common Language Runtime)

- Advantages over a native programming environment:
  - Single-system image
  - Transparent migration of threads
  - Dynamic optimisation and compilation
  - Language extensibility

- Investigate challenges of bringing up a JVM on Barrelfish.

- Comparing two different approaches:
  - Conventional shared-memory approach
  - Distributed approach in the style of Barrelfish
Outline

1. The BarreLFish Operating System
2. Implementation Strategy
   ▶ Shared-memory approach
   ▶ Distributed approach
3. Performance Evaluation
4. Discussion & Conclusions
5. Future Work
The Barreelfish Operating System

- Barreelfish is based on the Multikernel Model: Treats multi-core machine as a distributed system.
- Communication through a lightweight message-passing library.
- Global state is replicated rather than shared.

![Diagram of Barreelfish core components]

- **_user mode**: run in user-mode and together, the monitors across all cores coordinate to provide most traditional OS functionality, such as memory management, spanning domains between cores and managing timers.
  - Monitors communicate with each other via inter-core communication. Global OS state (such as memory mappings) is replicated between the monitors and kept consistent using agreement protocols.

- **dispatcher**: each core runs one or more dispatchers. These are user-level thread schedulers that are up-called by the CPU driver to perform the scheduling for one particular process. Since processes in Barreelfish can span multiple cores, they may have multiple dispatchers associated with them, one per core on which the process is running. Together, these dispatchers form the "process domain". Dispatchers are responsible for spawning threads on the different cores of a domain, performing user-level scheduling and managing...
Implementation

- Running real-world Java applications would require bringing up a full JVM (e.g. the *Jikes RVM*) on Barrelfish.
- Stresses the memory system (virtual memory is fully managed by the JVM), Barrelfish lacked necessary features (e.g. page fault handling, file system).
- Would have distracted from understanding the core challenges.
- **Approach**: Implementation of a rudimentary Java Bytecode interpreter that provides just enough functionality to run standard Java benchmarks (*Java Grande Benchmark Suite*).
  - Supports 198 out of 201 Bytecode instructions (except `wide`, `goto_w` and `jsr_w`), Inheritance, Strings, Arrays, Threads, ...
  - No Garbage Collection, JIT, Exception Handling, Dynamic Linking or Class Loading, Reflection, ...
Shared memory vs. Distributed approach

**Shared memory**

- JVM
- Domain
- Heap: obj A, obj B, obj C, obj D
- run_func_on

**Distributed Approach**

- JVM0, JVM1, JVM2, JVM3
- move_object
- move_object_ack
- return
- invoke
- putfield
- putfield_ack
The distributed approach

```
<table>
<thead>
<tr>
<th>jvm-node0</th>
<th>jvm-node1</th>
<th>jvm-node2</th>
<th>jvm-node3</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj_request</td>
<td>obj_request</td>
<td>obj_request</td>
<td>obj_request</td>
</tr>
<tr>
<td>blocks</td>
<td>obj-request-response</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Object lookup

Method call

```

invoke_virtual

invoke!return

getfield

getfield_response
Performance Evaluation

- Performance evaluation using the sequential and parallel *Java Grande Benchmarks* (mostly Section 2 - compute kernels).
- Performed on a 48-core AMD Magny-Cours (Opteron 6168).
- Four 2x6-core processors, 8 NUMA nodes (8GB RAM each).
- Evaluation of the shared-memory version on Linux (using `numactl` to pin cores) and Barrelfish.
- Evaluation of the distributed version only on Barrelfish.
- Compared performance to industry-standard JVM (OpenJDK 1.6.0) with and without JIT compilation.
Single-core (sequential) performance

- Consistently within a factor of 2-3 of OpenJDK without JIT.

![Bar chart showing execution times for various benchmarks on different platforms.](image-url)
Performance of the shared-memory approach

- Using the parallel sparse matrix multiplication Java Grande benchmark JGFSparseMatmultBenchSizeB.
- Scales to 48 cores as expected (relative to OpenJDK).
Performance of the shared-memory approach

- Quasi-linear speed-up implies large interpreter overhead.
- Barrelfish overhead presumably from agreement protocols.

![Graph](image)

### Figure 4.13: Average speed-up of the shared-memory approach

The graph illustrates the average speed-up of the shared-memory approach across different numbers of cores. The data points show the performance of both Linux and Barrelfish (Shared) systems. The y-axis represents speed-up, while the x-axis shows the number of cores.

### Table 4.1: Results of JGFSparseMatmultBenchSizeA*

<table>
<thead>
<tr>
<th>Cores</th>
<th>Run-time in s</th>
<th>(Standard deviation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>12.70 ± 1.00</td>
<td>17.89 ± 0.69</td>
</tr>
<tr>
<td>24</td>
<td>57.89 ± 3.54</td>
<td>47.61 ± 2.77</td>
</tr>
<tr>
<td>33</td>
<td>95.68 ± 1.35</td>
<td>81.54 ± 5.32</td>
</tr>
<tr>
<td>44</td>
<td>234.27 ± 2.71</td>
<td>216.13 ± 6.12</td>
</tr>
<tr>
<td>54</td>
<td>443.88 ± 2.27</td>
<td>427.65 ± 7.27</td>
</tr>
<tr>
<td>65</td>
<td>143.25 ± 1.63</td>
<td>136.76 ± 1.25</td>
</tr>
<tr>
<td>71</td>
<td>764.32 ± 2.47</td>
<td>747.24 ± 2.57</td>
</tr>
<tr>
<td>82</td>
<td>313.12 ± 1.89</td>
<td>305.90 ± 1.96</td>
</tr>
</tbody>
</table>

Since some executions exhibited a high variance, a result is given for this experiment.
Performance of the distributed approach

- Distributed approach is orders of magnitude slower than shared-memory approach.
- Sparse Matrix Multiplication is a difficult benchmark for this implementation: 7 pairs of messages for each iteration of the kernel (almost no communication for shared-memory).
- Overhead arguably caused by inter-core communication (150-600 cycles) and message handling in Barrelish.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Run-time in s</th>
<th>σ (Standard deviation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.70</td>
<td>0.002</td>
</tr>
<tr>
<td>2</td>
<td>458</td>
<td>7.891</td>
</tr>
<tr>
<td>3</td>
<td>396</td>
<td>3.545</td>
</tr>
<tr>
<td>4</td>
<td>402</td>
<td>7.616</td>
</tr>
<tr>
<td>5</td>
<td>444</td>
<td>2.128</td>
</tr>
<tr>
<td>6</td>
<td>514</td>
<td>36.77</td>
</tr>
<tr>
<td>7</td>
<td>1764</td>
<td>247.7</td>
</tr>
<tr>
<td>8</td>
<td>2631</td>
<td>335.9</td>
</tr>
<tr>
<td>16</td>
<td>9334</td>
<td>(only executed once)</td>
</tr>
</tbody>
</table>
Performance of the distributed approach

- Measuring completion time of threads on different cores shows performance limitation due to inter-core communication.
- All data "lives" on the same home node (Core #0).
- Cores 0-5 within a single processor, 6 & 7 is off-chip.

The results show that without optimisation, the distributed approach is too slow to be feasible, at least for this benchmark. Measuring the run-time of each individual thread gives evidence that this is caused by the overhead of message passing: While a thread running on the home node of the working set (jvm-node0) completes very quickly, threads on other cores take orders of magnitude longer (Figure 4.14). The diagram also confirms that communication with cores on other chips (#6 and #7) is significantly more expensive than on-chip communication (Figure 4.3).

For this particular benchmark, the distributed JVM has to exchange 7 pairs of messages for each iteration of the loop in Listing 4.1 (1 getfield, 1 astore, 5aload), while the shared-memory approach requires almost no inter-core communication (all arrays reside in the local cache most of the time and there is little contention, since different threads write to different parts of the output array). There are two basic aspects that add to the overhead of the message passing:

- **Inter-core communication**: Each message transfer has to invoke the cache coherence protocol, causing a delay of up to 150-600 cycles, depending on the architecture and the number of hops [12].
- **Message handling**: The client has to yield the interpreter thread, poll for messages, execute the message handler code and unblock the interpreter thread. This involves two context switches and a time in-
Discussion & Future Work

- Preliminary results show that future work should focus on reducing message-passing overhead and number of messages.
- How can these overheads be alleviated?
  - Reduce inter-core communication: Caching of objects and arrays, like a directory-based MSI cache-coherence protocol.
  - Reduce message-passing latency: Hardware support for message-passing (e.g. running on the Intel SCC).
- Additional areas of interest:
  - Garbage Collection on such a system.
  - Relocation of objects at run-time.
  - Logical partitioning of objects.
- Future work should investigate bringing up the Jikes RVM on Barrelfish, focusing on these aspects.
Questions?