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
Introduction

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

- Barrelsfish 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 Barrelsfish components]

- **Monitor**: The monitor runs 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.
- **Dispatchers**: 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 Barrelsfish 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 Barrelich.
- Stresses the memory system (virtual memory is fully managed by the JVM), Barrelich 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

Run func on

JVM

Domain

obj A

obj B

obj C

obj D

Heap

Distributed Approach

move_object

move_object_ack

JVM0

JVM1

return

invoke

JVM2

JVM3

putfield

putfield_ack

obj A

obj B

obj C

obj D
The distributed approach
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.
Sequential Performance

4.5 Multi-core performance

Performance on multiple cores was evaluated using the parallel SparseMatmult benchmark from the JGF benchmark suite. The benchmark was chosen since it stresses inter-core communication and does not use \( \text{Math.sqrt()} \), which exhibits different performance on Barrelfish and Linux (Figure 4.8).
Performance of the shared-memory approach

- Using the parallel sparse matrix multiplication Java Grande benchmark 
  JGFSparseMatmultBenchSizeB

Raw Performance

Speed-up

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

Run-time of the experiments (which would have been up to 25h otherwise).
Since the benchmark only measures the execution of the kernel, this gives a very close approximation for the run-time of 
JGFSparseMatmultBenchSizeA, divided by 10. I call this benchmark 
JGFSparseMatmultBenchSizeA*. Since some executions exhibited a high variance, is given for this experiment.
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 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 ($getfield, 1 astore, 5 aload$), 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 interval.

<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 shared-memory approach is similar on Linux and Barrelfish (overhead arguably from agreement protocols).

Distributed approach is orders of magnitude slower. Overhead caused by inter-core communication (150-600 cycles) and message handling in Barrelfish.

For this benchmark, have to exchange 7 pairs of messages for each iteration of the kernel, while shared-memory approach requires almost no inter-core communication.

How can these overheads be alleviated?

- Caching of objects and arrays (reduce communication).
- Hardware support for message-passing (e.g. Intel SCC).
Conclusion & Future Work

- Preliminary results show that future work should focus on reducing message-passing overhead and number of messages.
- Promising future work for the JVM:
  - A caching protocol for arrays, similar to a directory-based MSI cache coherence protocol.
  - Running the Barrelfish JVM on the Intel SCC.
- Additional areas of interest:
  - Garbage Collection on such a system.
  - Relocation of objects at run-time.
- Future work should investigate bringing up the Jikes RVM on Barrelfish, focussing on these aspects.
Questions?