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

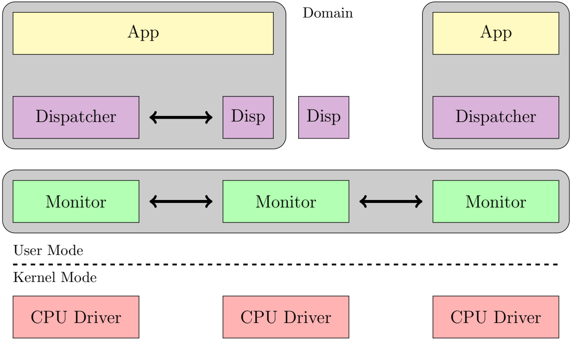
- **Answer**: Managed Language Runtime Environments (e.g. *Java Virtual Machine*, *Common Language Runtime*)
- Advantages over a native programming environment:
  - Single-system image
  - Transparent migration of threads
  - Dynamic optimisation and compilation
  - Language extensibility
- Investigate challenges of bringing up a JVM on Barrelfish.
- Comparing two different approaches:
  - Convential 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 Barrelfish Operating System

- Barrelfish 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.

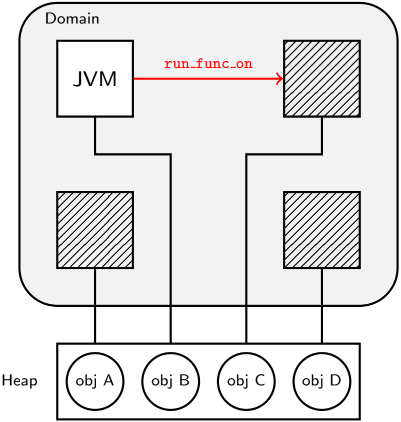


Implementation

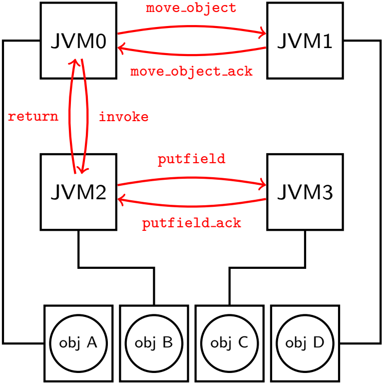
- Running real-world Java applications would require bringing up a full JVM (e.g. the *Jikes RVM*) on Barrelfish.
- Stresses the memory system (virtual memory is fully managed by the JVM), Barrelfish lacked necessary features (e.g. page fault handling, file system).
- Would have distracted from understanding 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

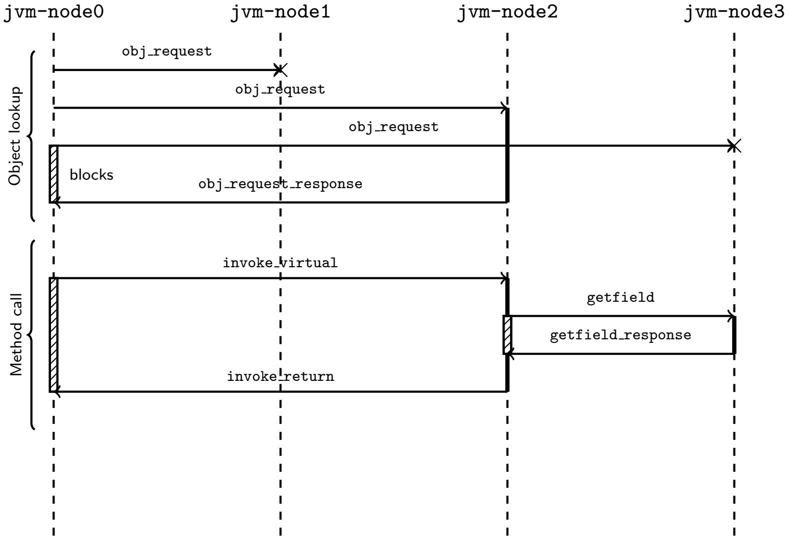
Shared memory



Distributed Approach



The distributed approach

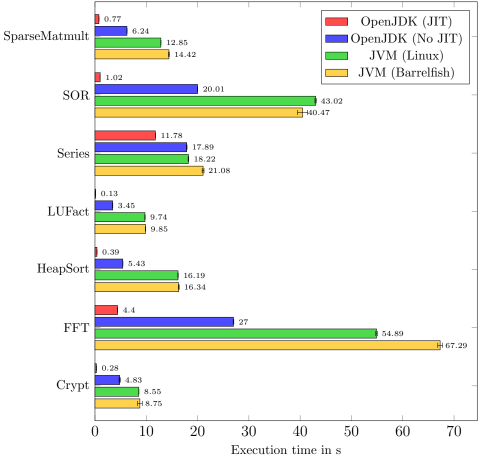


Performance Evaluation

- 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.

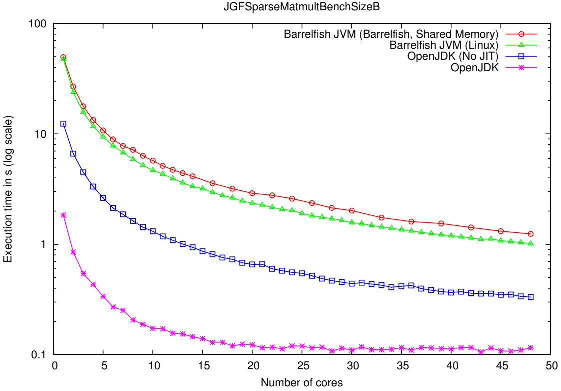
Single-core (sequential) performance

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



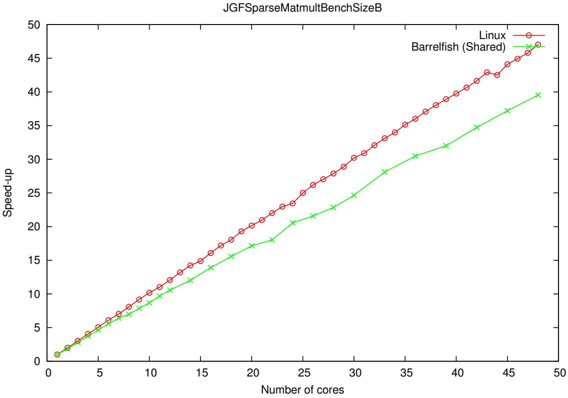
Performance of the shared-memory approach

- 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.



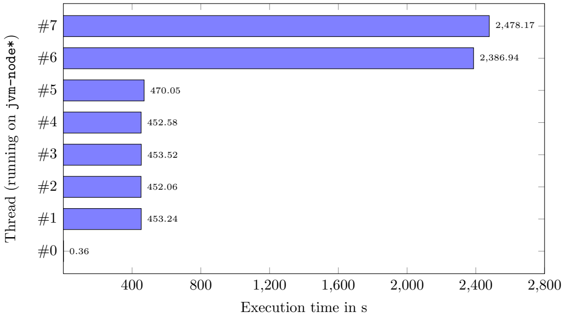
Performance of the distributed approach

- 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 Barrelfish.

Cores	Run-time in s	σ (Standard deviation)
1	2.70	0.002
2	458	7.891
3	396	3.545
4	402	7.616
5	444	2.128
6	514	36.77
7	1764	247.7
8	2631	335.9
16	9334	(only executed once)

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.



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, focussing on these aspects.

Questions?