A JVM for the Barrelfish Operating System

Martin Maas*
UC Berkeley
maas@cs.berkeley.edu

Ross McIlroy†
Google Inc.
rmcilroy@google.com

ABSTRACT

Barrelfish is a research operating system based on the Multikernel model, an OS structure that treats heterogeneous multi-core systems as a network of independent nodes communicating via message-passing. Arguably, such a system can benefit from high-level programming models such as the Java Virtual Machine, since they can provide a single-system image and facilitate migration of threads between cores, making the system easier to program. This work investigates the core challenges of this approach by bringing up a JVM on Barrelfish. We compare two different implementations, one based on shared memory, the other on message passing (enabling it to run on non-cache coherent systems).

1. INTRODUCTION

The past years have seen tremendous changes in hardware for commodity computer systems: Core counts are increasing and systems are becoming more diverse as new types of caches, interconnects and coprocessors (such as GPUs or programmable network adapters) appear. There is also a sense that future architectures will likely be non-cache coherent. For example, Mattson et al. [7] claim that the cost of cache coherence protocols prevents scaling to ever larger number of cores and that cores should instead communicate via message passing (such as the Intel SCC [8]).

This development puts challenges on operating systems as well as application programmers, who will have to take these new architectures into account. Arguably, higher-level programming models, such as the Java Virtual Machine (JVM), can help to hide these changes from the programmer. At the same time, there is a desire to run existing applications written for these programming models on novel architectures.

We investigate two different approaches of implementing a JVM for this kind of system. Our baseline approach uses shared memory and relies on hardware cache coherence. We then show a distributed version where each core is hosting a separate instance of the JVM (managing a different part of the heap) and communicates with the other instances via message passing. This allows us to assess the feasibility of such an approach and identify challenges in bringing a JVM to non-cache coherent systems.

The JVM is built on top of the Barrelfish operating system [3]. Barrelfish is based on the Multikernel model, an OS design that treats heterogeneous multi-core systems as a network of independent nodes communicating via message-passing. By doing so, it can exploit message-passing hardware while maintaining compatibility with shared memory architectures. State is replicated rather than shared, and traditional OS functionality such as memory management, I/O or power management is implemented as services running on different OS nodes. More information on Barrelfish is given in Section 4.

The main contributions of this paper are to show an early approach of implementing a JVM for novel architectures and operating systems like Barrelfish. It then shows the associated challenges, in particular the overhead of message passing, as well as identifying work necessary to solve them.

2. MOTIVATION

Writing applications for non-cache coherent and heterogeneous many-core systems is difficult and systems like Barrelfish require significant porting efforts to run existing applications. These problems can be alleviated by the use of a high-level runtime environment such as a JVM. Such a model has several advantages over the traditional model of writing explicit distributed applications:

- **Single-system image**: The run-time hides the distributed nature of the system, allowing it to run software that has not explicitly been developed for heterogeneous many-core systems. McIlroy et al. [9] showed that a JVM is a suitable abstraction for such systems.
- **Transparent migration of threads**: Migration of threads between heterogeneous cores is difficult in the current model, since it requires code to be compiled for each ISA and state to be translated between the different architectures. In a JVM, it is sufficient to provide a run-time environment for each core type, since code and state of the program are hardware-independent.
- **Optimisations**: The JVM provides high-level information (such as class structures) that can be used to optimise the code, e.g. by using this information for scheduling decisions or adaptive recompilation.

*Work performed while at the University of Cambridge.
†Work performed while at Microsoft Research, Cambridge.

- **Extensibility**: Java can be extended using annotations and language extensions, which makes it possible to gather additional information from the programmer to improve scalability to many-core architectures.

Some of these advantages also apply to parallel programming models such as Cilk. However, Java has several advantages over these models: It is a general-purpose programming language and is widely adopted for both client and server applications. This is important since Barrelfish targets commodity systems rather than HPC scenarios. The high adoption also means that there are stable benchmarks, development tools and libraries available. Furthermore, Java has the advantage of having excellent open source implementations and being prevalent in scientific research [4].

3. APPROACH AND GOALS

One way to approach this task would have been to port an existing JVM, such as the Jikes RVM [4]. While this will eventually be required to run real-world Java applications, we discarded it for the first iteration. Jikes requires memory management and file system facilities that were not yet available on Barrelfish (e.g. page fault handling, a hierarchical file system for the class loader). Porting Jikes would therefore have been a major effort that would have distracted from understanding the fundamental challenges of providing a high-level run-time on a system like Barrelfish. We found it important to understand those challenges first.

We therefore decided to implement a rudimentary Java Bytecode interpreter that provides just enough functionality to run a set of benchmarks and simple programs (we use the Java Grande Benchmark Suite [6]). While this means that the absolute numbers have to be treated with care, the preliminary results from this prototype allow us to investigate challenges that any JVM will have to deal with when running on Barrelfish. They can therefore be used as a first step towards porting a full JVM.

4. BACKGROUND

Several OS designs have been proposed to deal with the problems introduced by heterogeneous and non-cache coherent many-core systems [3, 11, 12].

Barrelfish's approach is based on the Multikernel model [3]. It runs an instance of the OS on each of the system's cores and inter-core communication is made explicit and implemented as a light-weight message passing library. While Barrelfish provides implementations of libraries such as `libc` that hide the distributed nature of the system for certain system calls (e.g. `malloc`), the programmer has to be aware of the distributed nature of the system and adopt an event-based software design. Each kernel instance contains the following core components (Figure 1):

- **CPU Driver**: This is Barrelfish's equivalent to a traditional kernel. It schedules and mediates core access of user-level processes, handles interrupts and provides local inter-process communication and low-level primitives for inter-core signalling (e.g. inter-processor interrupts). The CPU driver is lightweight and abstracts away little, while hiding the underlying hardware from the rest of the system. This allows Barrelfish to support heterogeneous systems, since the CPU driver exposes an ABI that is (mostly) independent from the underlying architecture of the core.

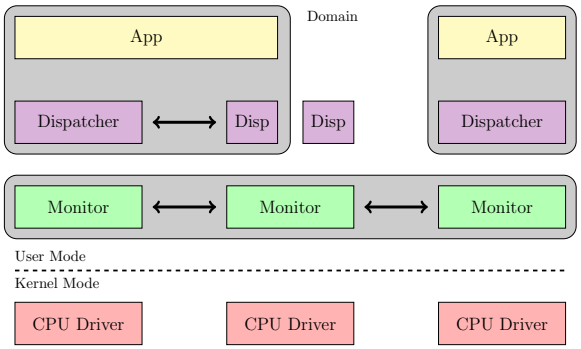

Figure 1: Barrelfish's core components

- **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 or timers. Monitors communicate with each other via inter-core communication. Global state (e.g. memory mappings) is replicated between monitors and kept consistent using agreement protocols.

- **Dispatchers**: Each core runs one or more dispatchers, a user-level thread scheduler that is called by the CPU driver to perform the scheduling for a particular process. Since processes in Barrelfish 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 a "domain", which shares a virtual address space on architectures that support shared memory. Dispatchers are responsible for spawning threads, performing user-level scheduling and managing thread synchronisation (e.g. waking up threads on remote cores). They communicate via message passing and replicate per-process state across the domain's cores.

Additionally, cores may run drivers, applications and system services. Communication between these services works via message passing as well. To enable connections between specific services, Barrelfish provides a global nameservice.

The low-level support for inter-core communication depends on the hardware and is implemented in the CPU driver. On a system that supports message passing in hardware, Barrelfish will use those capabilities, while on a shared-memory system, it will use the cache coherence protocol. Based on these primitives, the system provides a user-level RPC mechanism, exposed via a library that provides methods to poll for messages, set up handler methods, connect to other cores and send messages to them.

5. IMPLEMENTATION

The Barrelfish JVM comprises a class loader, linker and Java Bytecode interpreter supporting 198 out of the 201 instructions (the missing instructions are `wide`, `goto_w` and `jsr_w`). It supports many core Java features such as inheritance, strings, arrays, threads and synchronization. However, the JVM does not support garbage collection, just-in-time compilation, dynamic linking (classes are statically linked at start-up), exception handling and reflection. These features are not required to run most Java Grande benchmarks.

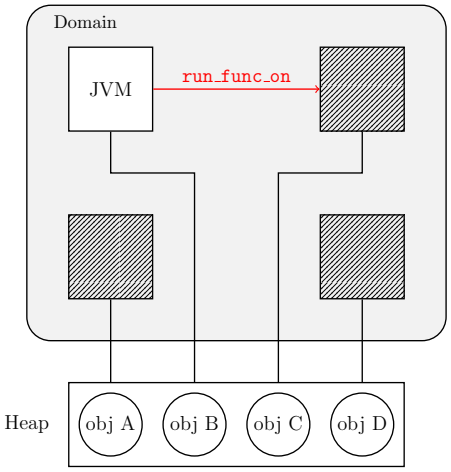

Figure 2: The Shared-memory approach

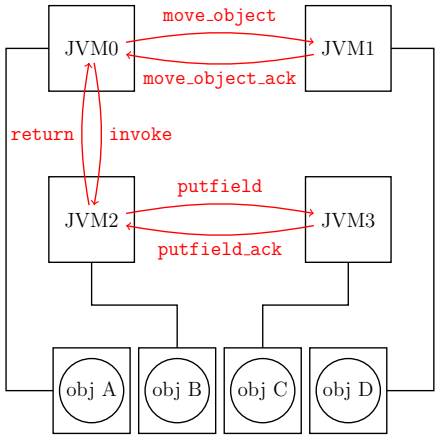

Figure 3: The Distributed approach

The JVM was mostly written in C and runs on Linux and Barrelfish. On Linux, it uses `glibc` and `pthreads` to implement most of its functionality. On Barrelfish, it runs as a service on one of the cores and provides an RPC interface to run a class's `main()` function. This can then be called from the Barrelfish shell. Our version of Barrelfish did not provide a file system. Hence, all Java class files were packaged with the JVM's executable and loaded by the boot loader.

While this JVM can spawn threads on multiple cores on Linux (and on a single core on Barrelfish), more work was required to run it on multiple cores on Barrelfish. As explained earlier, we investigated two different approaches:

5.1 The Shared-Memory JVM

This approach is similar to the one used on a traditional operating system: The JVM is first launched on a single core. As explained in Section 4, it then spans a domain to different cores, using a Barrelfish API that enables it to spawn new dispatchers. Once this has been done, the JVM can create threads on the remote core, similar to creating local pthreads (Figure 2). New threads are assigned to the different cores in a round-robin manner and spin locks are used for synchronization, as they gave significantly better performance than mutexes on Barrelfish. When operating in user-mode, this system is essentially the same as on Linux.

5.2 The Distributed JVM

This approach avoids the need for shared memory by running an instance of the JVM on every core and communicating solely via message passing (Figure 3). Our implementation is naïve, with many opportunities for optimisation. It is inspired by dJVM [14]: Each object has a *home node* where it resides. When a core performs an operation on an object, it sends a message to the object's home node, which executes the operation and returns an acknowledgement.

Multiple server instances are launched on startup, each on a different core and with a unique name (e.g. `jvm-node0`). Each of the nodes provides and manages its own set of components, including loader, linker and heap. The policy for choosing home nodes is simple: an object's home node is the node that executed the `new` instruction that created it (future work could investigate more elaborate schemes).

5.2.1 *Inter-core communication*

All JVM nodes are completely independent. To communicate with each other, they have to set up point-to-point connections between cores. Each node runs a message-handler thread that never blocks. This thread handles incoming messages and spawns new threads as necessary. All other threads may block at any time. For this purpose, they have a semaphore associated with them that blocks while e.g. waiting for a reply from another node. The reply handler then unblocks the thread and potentially stores the return value in a structure. To identify the thread that needs to be unblocked, a pointer to the structure is sent with each message and returned with the reply.

5.2.2 *Object relocation*

Moving objects between cores is essential to the distributed approach. While the JVM currently only relocates objects during the creation of a new thread (which consists of moving the `Thread` object to a different core and performing a remote method call on its `run()` function), it could be extended to relocate objects during the program's execution, similar to O^2 scheduling [5]. During relocation, the source node sends the object's data to the destination node, which adds it to its heap and replies with an acknowledgement. Heap references do not change when their entry is moved.

Once the transfer has finished, the source node removes the objects from its heap and adds an entry to a *core table*, a mapping from references to cores. This table represents the node's knowledge about the homes of different objects. To keep references unique across nodes, the first 8 bit of each reference are set to the id of the node that created it.

While our implementation simply nulls out entries that are removed from the heap (making it easy to determine whether an object is on the heap or not), a JVM with garbage collection could e.g. use 24 reference bits (out of 64) as a UID that is also stored in the object's header - if the bits do not match, it means that the object had been removed.

5.2.3 *Object lookup*

When a node tries to access an object whose location is unknown to it, the node needs to perform a lookup to determine the object's home node. There are two basic ap-

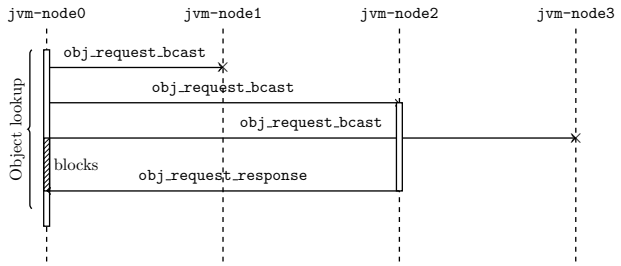

Figure 4: Object Lookup

proaches to this problem: a central directory of all object-location mappings or a broadcasting approach. We use the latter, to keep the JVM decentralised and avoid bottlenecks.

Whenever the interpreter tries to access an object, it first checks the local heap. If the object cannot be found on the heap, it tries to look it up in the core table. If the core table does not contain the object either, the node broadcasts a request to all other nodes and blocks the current thread. When a core receives such a request, it checks its local heap for the object in question and, if the object can be found, replies to the sender. Upon receipt of the reply, the core unblocks the thread and enters the new object-location mapping into the core table (Figure 4).

5.2.4 *Remote operations and method invocations*

Remote operations are method calls as well as `getfield`, `putfield`, `aload` and `astore` instructions on an object or array located at a different node (static fields/methods are all handled by core 0). Once the object's location has been determined (using the core-table and the lookup mechanism), the JVM sends a message to the remote core and blocks the current thread. When the remote core receives the message, it performs the operation and sends a reply back to the client, which then unblocks the thread and passes on the result. For method invocations, the remote core has to spawn a new thread to execute the method, since method invocations may block and therefore cannot be executed in the message-handler thread.

5.2.5 *Synchronization*

The `monitorenter/exit` instructions are implemented as messages to the home node of the object they apply to. The home node stores a queue of nodes that are blocking on a particular object. Incoming `monitorexit` messages trigger a reply to the front element of the queue while `monitorenter` messages are added to the end of it. Local threads are added to the queue as well, but no messages are sent in this case. This is similar to queue-based locks, such as MCS locks [10].

6. EVALUATION

All experiments were conducted on a 48-core AMD Magny-Cours (Opteron 6168) NUMA system, running both Linux and Barrelfish. The system contains four 2×6-core processors running at 1.9GHz with each processor accessing 2×8GB of RAM. Each core has a 512KB L2-cache and groups of six cores share a 6MB L3-cache. The system has 8 NUMA nodes, each with 8GB of memory. To ensure a low variance of results on Linux, threads were pinned to cores and memory affinity was ensured using the `numactl` tool.

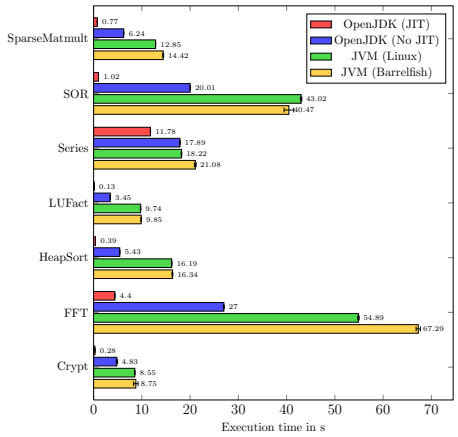

Figure 5: Single-core performance (for the single-core Java Grande, Section 2A benchmarks)

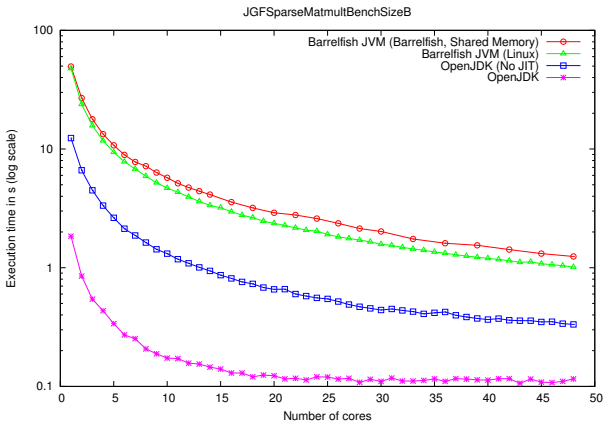

Figure 6: Multi-core performance (for the parallel Java Grande JGFSparseMatmultB benchmark)

6.1 JVM Performance

To understand how our JVM performs when compared to an industry-standard implementation, we compared its performance with the HotSpot JVM (OpenJDK 1.6.0) on Linux. For better comparison, we also ran this JVM with JIT disabled. Our first series of tests only used a single core and are based on the Java Grande benchmarks [6]. The results are shown in Figure 5 and show that while the JVM is up to a factor 2-3x slower than OpenJDK without JIT, this range remained relatively stable (`Series` being an exception), indicating that qualitative insights gained from our JVM can be used to make conclusions about other JVMs as well.

6.2 Shared-Memory Approach

To evaluate the feasibility of the shared-memory approach on Barrelfish, we measured the parallel *sparse matrix multiplication* Java Grande benchmark with different core counts on Linux and Barrelfish, as well as OpenJDK (Figure 6).

The results on Barrelfish and Linux scale as expected and seem almost identical. However, when looking at the speed-up compared to execution on a single core (Figure 7), it can be seen that Barrelfish introduces an additional overhead as the number of cores grows, which is most likely introduced

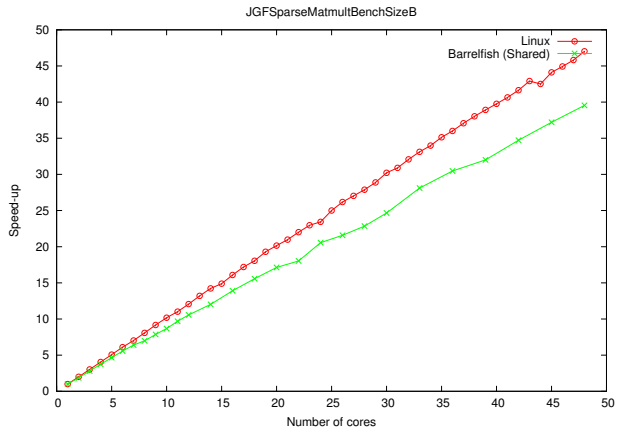

Figure 7: Shared-memory speed-up

by the OS consistency protocols. Note that the almost ideal speed-up of our JVM is arguably caused by the large overhead of the interpreter, which is expected to scale perfectly as it does not require inter-core communication. The impact of the overhead on Barrelfish might therefore be more significant for a JVM that uses a JIT compiler.

Nonetheless, these results indicate that the shared-memory approach works well and could be used as a basis for a JVM on Barrelfish. However, it would only work on systems that support shared memory, while systems based on message-passing need to use the distributed approach.

6.3 Distributed Approach

While we expected the run-times for the distributed JVM to be longer than for the shared-memory approach, the measurements reveal that the overhead is much larger than expected. On two cores, the distributed JVM was about 300 times slower than the shared-memory approach. The following run-times are for 1/10th of the iterations from JGFSparseMatmultBenchSizeA (compare to Figure 6):

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)

The results show that, as is, the distributed approach is orders of magnitude 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 the 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 8). The diagram also confirms that communication with cores on other chips (#6 and #7) is significantly more expensive than on-chip communication.

For this particular benchmark, the distributed JVM has to

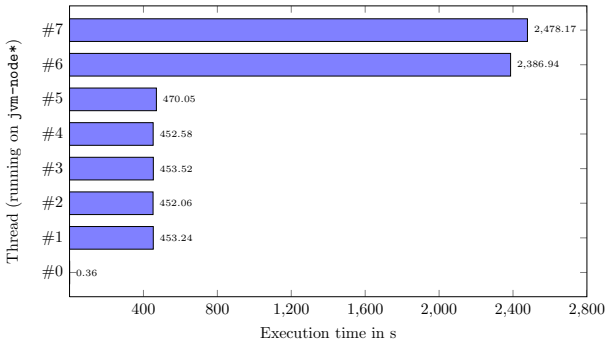

Figure 8: Run-times of individual threads

exchange 7 pairs of messages for each iteration of the benchmark's kernel, 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 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 [3].
- **Message handling**: The client has to yield the interpreter thread, poll for messages, execute the message handler and unblock the interpreter thread. This involves two context switches and a time interval during which the core is polling for messages. Running multiple threads on the same core exacerbates this problem.

The fact that the JVM has a lower run-time on three cores than on two and four cores may indicate that the run-time on two cores is limited by the latency of inter-core communication and message handling at the client, while the performance on four cores seems to be limited by the message handling on node 0. The high variance for six and more cores may be introduced by a saturation of the bus through the cache coherence protocol.

7. DISCUSSION

The results show that a naïve implementation of the distributed approach is infeasible on the given hardware and imply that the following steps need to be taken when implementing a JVM on Barrelfish using the distributed approach:

- **Reducing the number of messages**: This could be achieved by caching objects and arrays, using a mechanism similar to a directory-based MSI cache coherence protocol. For example, the number of messages in the `JGFSparseMatmult` benchmark could be reduced to almost zero, as most of the arrays are read-only and threads access different ranges of the output array.
- **Reducing the latency of messages**: Different approaches could be used to reduce the latency. The Barrelfish team discusses this topic in the documentation of the March 2011 release and proposes the use of inter-processor interrupts (IPI).

8. RELATED WORK

Distributed JVMs.
There has been an extensive amount of publications on running Java programs distributed across machines: One of the first solutions was Sun's own RMI framework, which implemented RPCs directly in Java. Other approaches include a custom JVM running distributed across multiple servers (cJVM [2]) and a monolithic JVM running on a distributed computing platform with distributed shared memory (Kaffemik [1]). While this work is very relevant for the Barrelfish JVM, the trade-offs on a cluster are different to those in a multi-core machine. For example, message passing is orders of magnitude cheaper and a lower error-tolerance is required. The distributed approach chosen by the Barrelfish JVM resembles cJVM while the shared memory version is similar to a JVM using distributed shared memory.

Java on many-core systems.
There has been research on making Java more effective on many-core systems. One example is Kilim [13], a lightweight framework that allows Java threads on a single machine to communicate via message-passing rather than shared memory, an approach similar to that of Barrelfish. However, this system runs within Java on a traditional operating system while the Barrelfish JVM is effectively running on a distributed system with unusual characteristics.

JVMs for heterogeneous systems.
Hera-JVM [9] implements a JVM for the heterogeneous Cell microprocessor, providing a single-system image and transparently migrating threads between cores. Our work differs in that it uses the underlying operating system to handle heterogeneity and communication between cores.

9. CONCLUSIONS AND FUTURE WORK

The presented results indicate that more work will be necessary when bringing a production-grade JVM to Barrelfish using the distributed approach. As discussed previously, a core improvement could be a fine-grained caching protocol: Each array is split into chunks of equal size and the array's home node stores a set of sharers for each chunk. Cores can request read access (in which case they will be added to the set of sharers) or write access (in which case the home node sends an invalidate message to all sharers before marking the chunk as modified and returning it to the requester). When a node tries to access a modified chunk, the chunk's home node prompts the holder of the chunk to write it back.

Additional future work may include the following:

- **Notifications and IPI**: Using Barrelfish's notification features to reduce the latency of messages. According to the Barrelfish documentation, latencies of 4,000 cycles per message could be achievable (instead of 25,000 cycles in the current version).
- **Object relocation** The current JVM only relocates objects when a new thread is created. This could be changed to make placement decisions at run-time and move objects between cores.
- **Running on the SCC**: Running Barrelfish on the Intel SCC [8] would allow to evaluate whether performance improves with message passing in hardware.

We conclude that while it is possible to build a JVM for a system like Barrelfish, the overhead of message-passing makes the naïve approach infeasible. However, this overhead may be reduced by optimisations such as caching.

Acknowledgements: *Thanks to Tim Harris for the original idea and supporting this project throughout; and to John Kubiatowicz and the anonymous reviewers for their feedback.*

10. REFERENCES

[1] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik - A distributed JVM on a single address space architecture. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *icpp*, page 4, 1999.

[3] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, volume 9, pages 29–44, 2009.

[4] A. A. Blackburn, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. Mckinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005.

[5] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *HotOS-XII*, May 2009.

[6] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Proceedings of the ACM Java Grande Conference*, pages 81—88, 1999.

[7] T. G. Mattson, R. V. der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. SC '08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

[8] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, et al. The 48-core SCC processor: the programmer's view. In *SC'2010*, pages 1–11, 2010.

[9] R. McIlroy and J. Sventek. Hera-jvm: a runtime system for heterogeneous multi-core architectures. In *OOPSLA'10*, pages 205–222, 2010.

[10] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *SIGARCH Comput. Archit. News*, 19:269–278, April 1991.

[11] K. Modzelewski, J. Miller, A. Belay, N. Beckmann, C. Gruenwald, D. Wentzlaff, L. Youseff, and A. Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation.

[12] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP*, 2009.

[13] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. *ECOOP'08*, pages 104–128, 2008.

[14] J. N. Zigman and R. Sankaranarayana. dJVM - A distributed JVM on a cluster. Technical report, Australian National University, 2002.