A Cluster-Aware Distributed Java Virtual Machine written in Java

Peter Strazdins,
(with John Zigman, Ramesh Sankaranarayana and James Sinnamon),
Department of Computer Science,
The Australian National University
peter@cs.anu.edu.au djvm@upside.anu.edu.au

August 2003
Talk Outline

1. Motivation
2. Related Work
3. Approach
4. Issues
5. Performance
6. Current Status
7. Future Plans
8. Conclusion
Motivation

- Investigate distributed Java Virtual Machine (dJVM) focusing on long running low contention server side applications.
- Target commodity hardware, e.g. Bunyip—a 96 node cluster of PCs.
- dJVM platform enables the investigation of techniques and algorithms for supporting distribution, including:
  - methods for determining object placement, caching and migration, and
  - the evaluation of runtime support algorithms, such as distributed garbage collection.
Related Work

There are broadly three approaches to the effective implementation of distributed JVM:

1. Provide a solution built on top of a set of standard JVMs, either by:
   - static transformation—ahead of time transformation, or
   - dynamic transformation—just in time transformation.

2. Building a JVM on cluster enabled infrastructure, such as a DSM.

3. Developing a cluster aware JVM.
Approach—Design

● Maintain the standard Java programming environment—Single System Image (SSI).

● Master-slave model of class management.

● Distribution model:
  – Execution normally based on object location.
  – Distribution by placement of objects (including threads) and subsequent migration.

● Independent memory management on nodes (so far).
Approach—Implementation

- Effect distribution through transformation of code:
  - Ideally—$f(JVM) \mapsto dJVM$.
  - Practically—combination of $f$ and infrastructure changes.

- Use the Jikes RVM developed by IBM Research:
  - Written in Java.
  - Compiles to native code.
  - Integrates runtime and application code.
Issues—Globally Visible Data

- JVM runtime structures have global scope in a non-distributed implementation.

- Initialization must only occur once.

- Booting is a two phase process, before and after joining a cluster:
  - Initialization locally.
  - Coalesce global data.
  - Unify type information identity.
Issues—Remote Objects

- Objects are owned by a particular node, and given a global identity on demand.

- References to remote objects are distinguished from those to local objects by their form. Consequently, software faulting is based on the form of a reference.

- Type information is always locally available even for references to remote objects.
Issues—Transformations

● Define transformations that apply to:
  – the construction of the JVM image,
  – the transformation code, and
  – the application code.

● Transformations need to be incorporated in-between the JVM load and link sequence phases.

● Annotations to guide or direct transforms extend the Jikes RVM annotation model:
  – Implement interfaces.
  – Method throw lists.
  – Try/Catch blocks.
Issues—Magic

Reference faulting requires:

- Hooks for defining the form of references to local and to remote objects.

- The JVM to allow dynamic type resolution through faulting references.

- GC must allow mechanisms to handle faulting references.
Issues—Threading Support

- A thread resource is local to a node.
- Threads may take on the identity of a remote thread, acting as a proxy.
  - invoking a method on a remote object thus involves a remote method call (RMC)
  - class loading on slave nodes is done similarly by an RMC to the master node
- Local thread resources reused for correctness & performance.

- As well as threads corresponding to the application, there are threads associated with the dJVM itself
  - these include a communication thread, activated whenever a message is received
Issues—Thread Reuse

Node 1

VM_Thread A

Node 2

VM_Thread B

a.m() c.y() b.x()
Performance Evaluation

- Platform used: a 20 node Beowulf cluster (550 MHz dual PIII, 10Mb/s network)
- Application: numerical integration program, 1 master and \( p \) slave thread
  - synchronized using standard `wait()`/`notify()` methods
  - communication of input / results via RMC parameters / return values
Performance Evaluation – Results

- Overhead of an RMC $40 - 80 \times$ that of MPI round-trip message (260\,$\mu$s)

- Overhead of remote class loading & thread creation $5 - 10 \times$ greater still

- Overheads high even when master and slave threads were run on same node

- Parallel speedup at $p = 20$ of 5.2 (with 0.22s computation & 0.04s for 2 RMC’s per integration)

- Major problem: all JVM-level threads were run on a single kernel-level thread: communication overheads were a multiple of a timeslice!
Current Status

● Prototype based on Jikes RVM 2.0.2 using the baseline compiler running on a 20 node cluster.

● Testing the prototype revealed two serious issues:
  – Thread switching bug.
  – Adverse scheduling behaviour.

● The port to the latest version of the Jikes RVM is near completion. In conjunction with the port we are:
  – minimizing the invasiveness of the changes, raising as many alterations to the bytecode level as practical, and
  – incorporating modifications to enable the optimizing compiler to be used.
Future Work

- Expect (2nd) release based on Jikes 2.2.0 under GPL 11/03
  - preliminary tests indicate much better performance

- Investigate:
  - Placement, migration and caching policies and techniques.
  - Tuning underlying infrastructure layers.
  - Incorporating Fault-tolerance for long-running applications.
  - Distributed Garbage Collection algorithms and inter-node contracts.

and their interactions and synergies with each other.
Conclusions

- The dynamic bytecode transformation approach combined with a JVM written in Java provides a high level of flexibility, allowing transformation techniques to be applied at build and runtime.
  - does have drawbacks; more scope for very subtle bugs

- Analysis and transform techniques can be re-executed at runtime as further information comes to light.

- Identifying and addressing global data issues can be non-trivial.

- Low level mechanisms are JVM dependent.
We believe that:

- Interoperability can be addressed with appropriate inter-node interfaces and contracts.
- Portability issues can be reduced to a small number of low level JVM specific modifications.
Acknowledgements

- Fujitsu Labs Pty Ltd (sponsor)
  - beginning of dJVM project (2001–2002)
  - reliability extensions to the dJVM (2002-)