JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support

Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
{wzzhu+clwang+fcmlau}@csis.hku.hk
Outline

- Motivations
- Related works
- JESSICA2 features
- Experimental results
- Conclusion & Future works
**Motivation**

- **Why Java?**
  - The dominant language for server-side programming
  - Platform independent
  - Built-in multithreading support at language level
  - High-performance with Just-in-Time compilation

- **Why cluster?**
  - A cluster provides a scalable parallel hardware platform for high performance computing
Parallel/Distributed Computing using Java

- RMI, Cobra?
  - Application level
  - Complex programming model
  - Can’t take advantage of Java’s multithreading features

- Java Multithreading
  - Running a multithreaded Java application on a cluster
  - A Distributed Java Virtual Machine (DJVM) Approach
A distributed Java Virtual Machine (DJVM) spanning multiple cluster nodes can provide a true parallel execution environment for multithreaded Java applications with a Single System Image illusion to Java threads.
An abstract view of (Distributed) JVM

TEM Model

T: Thread System
E: Execution Engine
M: Memory Space
Design issues of DJVM

- Extend TEM to distributed environment
  - T -> thread creation and migration mechanisms
  - E -> execution engine should be aware of the cluster environments
  - M -> provide a global object space in a distributed environment
Related works

- cJVM (IBM Hafia Research): Remote Creation, Embedded OO-based DSM (Proxy)
- JESSICA (HKU): Transparent Migration, Page-based DSM
- Java/DSM (Rice): Manual Distribution, Page-based DSM
- Hyperion (NHU), Jackal (Vrije U): Remote Creation
- Others: Intr/JIT, Page-based DSM

Intr = Interpreter

T E M
Problems in existing DJVM’s

- Can’t preserve Java’s merits
  - Static compilation (Hyperion, Jackal) => No dynamic class loading
  - Interpreters (cJVM, Java/DSM, JESSICA) => Can not support JIT compilation
  - Manual distribution (Java/DSM) => Need to re-write programs

- Layered design using DSM can’t be tightly coupled with JVM
  - JVM runtime information can’t be channeled to DSM
  - False sharing problem if page-based DSM is employed
Our strategies

- Preemptive transparent Java thread migration in JIT mode
  - No source code modification or bytecode instrumenting
  - Runtime Capturing and Restoring of thread execution context at bytecode boundary
  - Able to be executed in JIT compilation mode
  - Enable dynamic load balancing on clusters

- Embedded Global Object Space layer
  - Take advantage of JVM runtime supports to reduce object access overheads
JESSICA2 Architecture

Transparent migration

JIT

GOS

Portable Java Frames

Migration

Load monitor

OS

Hardware

Master JVM

worker JVM

JITEE

Global Object Space

Global Object Space

Host Manager

threads

threads

JITEE

Host Manager

JITEE

Host Manager

Migration order

Communication Network

Migration

Migration

Threads

Threads

Threads
Transparent thread migration in JIT mode?

- Simple for interpreters (e.g. JESSICA)
  - Interpreter sits in the bytecode decoding loop which can be stopped upon a migration flag checking
  - The full state of a thread is available in the data structure of interpreter
  - No register allocation

- JIT mode execution makes things complex (JESSICA2)
  - No clear bytecode boundary
  - How to deal with machine registers?
  - How to organize the stack frames?
  - How to restore an execution of native codes?
What are those functions?

**Migration Points Selection**
- At the head of loop basic block + method

**Register Context Handler**
- **Nondestructive register spilling**: spill dirty registers at migration point without invalidation so that native codes can continue the use of registers
- **Register rebuild**: use register recovering stub at restoring phase

**Variable Type Deducing**
- Spill type in stacks using compression

**Java Frames Detection**
- Discover consecutive Java frames
Details of Transparent Java thread migration inside JIT compiler

1. migration checking
2. Non-destructive register spilling
3. Object checking
4. Type spilling for variable type deducing

Variables
(Restore)
Register allocation
Register rebuild

Bytecode verifier
control flow graph

Bytecode translation
 Intermediate Code

invoke

1. migration point selection

Native Code

code generation

Linking & Constant Resolution

Global Object Space

Java frame detection
thread stack

reg var

mov var1->reg1
mov var2->reg2 ...

191 Variables

Java frame
C frame

Raw stack
Global Object Space (GOS)

- Provide global heap abstraction for DJVM
- Home-based object coherence protocol, compliant with JVM Memory Model
  - OO-based to reduce false sharing
- Non-blocking communication
  - Use threaded I/O interface inside JVM for communication to hide the latency
- Adaptive object home migration mechanism
  - Take advantage of JVM runtime information for optimization
Overview of GOS

Global Heap Abstraction
Adaptive object home migration

Definition
- “home” of an object = the JVM that holds the master copy of an object

Problems
- cache objects need to be flushed and re-fetched from the home whenever synchronization happens

Adaptive object home migration
- if # of accesses from a thread dominates the total # of accesses to an object, the object home will be migrated to the node where the thread is running
Experimental Setting

- Pentium II 540MHz, 128MB
- Linux 2.2.1 kernel
- Connected by Fast Ethernet
- Kaffe 1.0.6
Microbenchmarks(I)

CPI breakdown

- Capture time
- Parsing time
- Resolution of methods
- Frame setup time

<table>
<thead>
<tr>
<th>Frame number</th>
<th>1 frame (475Bytes)</th>
<th>2 frames (482Bytes)</th>
<th>11 frames (3,049Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack capturing</td>
<td>232</td>
<td>437</td>
<td>12,993</td>
</tr>
<tr>
<td>Frame parsing</td>
<td>166</td>
<td>328</td>
<td>1,383</td>
</tr>
<tr>
<td>Resolution</td>
<td>3,431</td>
<td>13,747</td>
<td>227,587</td>
</tr>
<tr>
<td>Frame setup</td>
<td>9</td>
<td>13</td>
<td>49</td>
</tr>
<tr>
<td>Overall time</td>
<td>3,838</td>
<td>14,525</td>
<td>242,012</td>
</tr>
</tbody>
</table>
Microbenchmark(II)
(Execution time in microseconds)

Java Granda benchmark result
(Single node)

<table>
<thead>
<tr>
<th>Task</th>
<th>Kaffe 1.0.6 JIT</th>
<th>JESSICA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>ForkJoin</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Sync</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>Crypt</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>LUFact</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>SOR</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>Series</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>SparseMatmul</td>
<td>20</td>
<td>15</td>
</tr>
</tbody>
</table>
JESSICA2 vs JESSICA (CPI)

CPI(50,000,000 iterations)

Number of nodes

Time (ms)

JESSICA

JESSICA2
Application benchmark

![Graph showing speedup vs. node number for different applications]

- Linear speedup
- CPI
- TSP
- Raytracer
- nBody
Parallel Ray Tracing on JESSICA2
(Running at 8-node P-III cluster)

Execution time: 900 seconds (15 minutes)
Take more than 10 hours to run on single node

800x600 image size, 114 objects
Effect of Adaptive object home migration (SOR)
Conclusions

- Transparent Java thread migration in JIT compiler enables the high-performance execution of multithreaded Java application on clusters
- An embedded GOS layer can take advantage of the JVM runtime information to reduce communication overhead
Works in Progress

- Exploit new optimization techniques on GOS
- Incremental Distributed GC
- Add load balancing module
- Enhanced Single I/O Space to benefit more real-life applications
- Parallel I/O Support
Thanks

- Q & A