Jupiter: A Modular and Extensible JVM Infrastructure

Patrick Doyle and Tarek Abdelrahman
Department of Electrical and Computer Engineering
University of Toronto
Motivation: Research into Scalable JVMs

- **Jupiter project research goal**: to investigate JVM architectures to deliver high performance on large-scale parallel systems.
  - 128-processor cluster of workstations with software-based SVM.
  - Single system image (SSI).

- To explore different approaches, we need a JVM infrastructure that is *Modular, Flexible, and Efficient*.

- Currently available JVMs:
  - Hard to modify (Kaffe, Sun’s JDK).
  - Designed to explore a specific portion of JVM design space (Jalapeño, OpenJIT, Joeq, etc.).

- Hence, we elected to build our own infrastructure: *The Jupiter JVM*.
  - Hard.
  - Rewarding.
Outline

- Philosophy and architecture.
- Design techniques.
  - Flexibility.
  - Performance.
- Experimental evaluation.
- Conclusions and future work.
The Jupiter JVM Framework

● Flexible design.
  – Modular components.
  – Object-oriented building-block architecture.

● Functional JVM.
  – Multi-threaded interpreter.
  – Executes the SPECjvm98 benchmarks.
  – Interfaces to Classpath library and Boehm garbage collector.
  – Currently no bytecode verifier and no JIT compiler.

● Competitive performance on SPECjvm98.
  – 2.65 times faster than Kaffe.
  – 2.20 times slower than Sun’s JDK.
  – Suitable for research on JVM scalability and performance issues.
Jupiter Overview

- **Resources**: items used by a running Java program.
- **Sources**: allocate and/or manage a resource.
An Incarnation of Jupiter

- Building-block architecture.
  - A JVM is assembled by interconnecting modules.
  - Nature of interconnections determines behaviour of JVM.
Object Allocator Example

- Extend for NUMA multiprocessors:
  
  ```
  void *nodeAlloc(int nodeNumber, int size);
  ```

- Multiple levels:
  - Memory allocation level.
  - Object allocation level.
  - Execution engine level.
Node-specific MemorySources

- MemorySource for each node.
  - Minimal change to the implementation of MemorySource.
  - Locality decisions made transparently in MuxMemorySource.
  - Same MemorySource interface.
Node-Specific ObjectSources

- ObjectSource for each node.
  - Enhances locality without altering ObjectSource interface.
  - Node choices still transparent to ExecutionEngine.
  - Same ObjectSource, MonitorSource, MemorySource\_i as before.
A Locality-Aware ExecutionEngine

- Locality-aware ExecutionEngine.
  - ExecutionEngine directly manages locality.
  - Same ObjectSource, MonitorSource, MemorySource_i as before.
  - Modifications confined to ExecutionEngine.
Design for Flexibility

- Flexibility achieved by applying many design techniques.
  - Interface coding conventions
  - Design by Contract
  - Splitting over-constrained interfaces
  - Modularity by maintenance characteristics
  - Pervasive error handling

- Example: *Splitting over-constrained interfaces.*
Over-constrained Interfaces

- **Interface**: the means by which pairs of modules communicate.
- Modules impose requirements on interfaces.
- **Over-constrained interface**: no design meets all requirements.
  - Requires modules to compromise.
  - Trade-offs impair flexibility.
- **Solution**: split the over-constrained interface into two.
- Insulating layer relieves each interface from the other’s constraints.
Example: Call Stack Bookkeeping

- One frame for each method pending execution.
  - A frame is “pushed” when a method is invoked.
  - A frame is “popped” when a method returns.

- Bookkeeping:
  - Keeping track of current top frame.
  - Propagating return value.
  - Synchronization.

- Whose responsibility?
**Example: Call Stack Bookkeeping**

**Method invocation:**

```c
    cs_pushFrame(cs, method);
    ...
    cs_popFrame(cs);
```

**Complicated!**

**cs_pushFrame(cs, method):**

```c
    nextFrame = curFrame + methodFrameSize(method);
    nextFrame->prev = cs->curFrame;
    cs->curFrame = nextFrame;
    if(isSynchronized(method))
        mn_enter(mbMonitor(method, curFrame));
```

**cs_popFrame(cs):**

```c
    if(isSynchronized(method))
        mn_enter(mbMonitor(method, curFrame));
    Value result = fr_pop(cs->curFrame);
    cs->curFrame -= methodFrameSize(fr_method(curFrame));
    fr_push(cs->curFrame, result);
```

**Complicated!**

**Method invocation:**

```c
    nextFrame = cs_getFrame(cs, method, curFrame);
    nextFrame->prev = curFrame;
    curFrame = nextFrame;
    if(isSynchronized(method))
        mn_enter(mbMonitor(method, curFrame));
    ...
    if(isSynchronized(method))
        mn_exit(mbMonitor(method, curFrame));
    Value result = fr_pop(curFrame);
    curFrame = curFrame->prev;
    fr_push(curFrame, result);
```

**Complicated!**

**cs_getFrame(cs, method, curFrame):**

```c
    return curFrame + methodFrameSize(method);
```
Example: Call Stack Bookkeeping

Method invocation:
  \texttt{cx\_pushFrame(cx, method);} \\
  \texttt{\ldots} \\
  \texttt{cx\_popFrame(cx);} \\

\texttt{frs\_getFrame(frs, method, curFrame):} \\
  \texttt{return curFrame + methodFrameSize(method);} \\

- Insulating layer makes both sides of the interface simpler.
- Small flexibility improvements accumulate.
Design for Performance

- Performance achieved by applying many design techniques.
  - Design by Contract
  - Lazy computation
  - Reducing arithmetic implied by interfaces
  - Promoting function inlining
  - Exploiting immutability

- Example: *Exploiting immutability.*
Exploiting Immutability

- **Immutable data**: never changes after construction.
  - Easily cached.
  - Freely duplicated.
  - Passed by value or reference.
  - No consistency problems among multiple copies.

- Therefore, Jupiter’s data is immutable whenever practical.

- Immutable data provides **opportunities for efficient implementation**.
Two problems:
- **Call overhead**: must make calls through the object hierarchy.
- **Object proliferation**: MemorySource\_i for each node i.
Example: Locality-aware Memory Allocation

- Custom declarations exploiting immutability of node number:

```c
typedef int MemorySource;

inline MemorySource ms_new(int nodeNumber){
    return nodeNumber;
}

inline MemorySource ms_newMux(){ return -1; }

inline void *ms_getMemory(MemorySource this, int size){
    if(this == ms_newMux())
        return nodeAlloc(/* Heuristic */, size);
    else
        return nodeAlloc(this, size);
}
```
Example: Locality-aware Memory Allocation

```c
void *ptr = ms_getMemory(obs_memorySource(), size);
```

```
void *ptr = ms_getMemory(ms_newMux(), size);
```

```
void *ptr = ms_getMemory(-1, size);
```

```
void *ptr = nodeAlloc(/* Heuristic */, size);
```
Performance Evaluation

- Single-threaded SPECjvm98 benchmarks.
- Jupiter’s interpreter is competitive with other interpreters:
  - 2.65 times faster than Kaffe.
  - 2.20 times slower than Sun’s JDK.
- Once a JIT compiler is added, interpreter performance is less significant.
  - Typical results: less than 10% of time spent in interpreter.

\[
\text{JIT + Jitted Code: 90%}
\]
Opportunities for optimization exist.

- Object Access: slow!
Impact of Successive Optimizations

- Jupiter’s flexibility facilitated modifications.
  - Minimal effort.
  - Independent.
Object access time greatly improved.
Opportunities for optimization remain.
## Multithreaded Performance

4-Threads/4-Processors

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JDK</th>
<th>Jupiter</th>
<th>Kaffe</th>
<th>Jupiter/JDK</th>
<th>Kaffe/Jupiter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 205 raytrace</td>
<td>51s</td>
<td>140s</td>
<td>640s</td>
<td>2.72:1</td>
<td>4.58:1</td>
</tr>
<tr>
<td>2 Series</td>
<td>103s</td>
<td>127s</td>
<td>313s</td>
<td>1.23:1</td>
<td>2.47:1</td>
</tr>
<tr>
<td>3 LUFact</td>
<td>26s</td>
<td>50s</td>
<td>66s</td>
<td>1.95:1</td>
<td>1.30:1</td>
</tr>
<tr>
<td>4 Crypt</td>
<td>29s</td>
<td>53s</td>
<td>118s</td>
<td>1.82:1</td>
<td>2.23:1</td>
</tr>
<tr>
<td>5 SOR</td>
<td>123s</td>
<td>234s</td>
<td>289s</td>
<td>1.90:1</td>
<td>1.23:1</td>
</tr>
<tr>
<td>6 SparseMatmult</td>
<td>46s</td>
<td>85s</td>
<td>294s</td>
<td>1.83:1</td>
<td>3.47:1</td>
</tr>
<tr>
<td>7 MolDyn</td>
<td>296s</td>
<td>657s</td>
<td>1578s</td>
<td>2.22:1</td>
<td>2.40:1</td>
</tr>
<tr>
<td>8 MonteCarlo</td>
<td>105s</td>
<td>223s</td>
<td>565s</td>
<td>2.21:1</td>
<td>2.43:1</td>
</tr>
<tr>
<td>9 RayTracer</td>
<td>373s</td>
<td>886s</td>
<td>3233s</td>
<td>2.37:1</td>
<td>3.65:1</td>
</tr>
</tbody>
</table>

Geometric Mean 1.99:1 2.43:1
Conclusions

- Presented a flexible JVM framework.
  - Object-oriented building-block architecture.
  - Carefully-designed module interfaces.

- Competitive early performance.
  - 2.65 times faster than Kaffe.
  - 2.20 times slower than Sun’s JDK.

- Suitable for research on JVM scalability and performance issues.
Future Work

- Further interpreter optimizations.
- JIT compiler.
- Parallel scalability and performance research.

Internet resources:
- Project Home Page:
  - [http://www.eecg.toronto.edu/jupiter](http://www.eecg.toronto.edu/jupiter)
  - Source code download.
- Patrick Doyle’s Jupiter page:
  - [http://www.eecg.toronto.edu/~doylep/jupiter](http://www.eecg.toronto.edu/~doylep/jupiter)
  - Publications.