The VMKit project: Java and .Net on top of LLVM

Nicolas Geoffray
Université Pierre et Marie Curie, France
nicolas.geoffray@lip6.fr
What is VMKit?

- Glue between existing VM components
- A drop-in replacement to Java and .Net
- Usage:
  - vmkit -java HelloWorld
  - vmkit -net HelloWorld.exe
Scientific Goals

• Build VMs with existing components
  • JIT, GC, class libraries, threads

• VM Interoperability
  • Execution in the same address space
  • Isolation

• VM communications
This talk: Trying to bring back the VM in LL VM

1. The design of VMKit
2. VMKit's performance
3. LL: "VM where are you?"
The design of VMKit
VMKit

Glue between existing VM components

- JIT: LLVM
- GC: Boehm, Mmap2
- Libraries: Classpath, Mono, Pnetlib
- Threads: Posix threads
- Exception handling: GCC
VMKit runtime

- Internal representation of classes (assemblies)
- Convert bytecode (ILASM) to LLVM IR
- Method (field) lookup
- VM runtime
  - Threads, reflection, exceptions
Execution overview (java, .net)

- Load .class and .jar (.exe)
- JIT main()
  - Insert stubs for methods (lazy compilation) and fields
- Run main()
  - Stubs call JIT
  - Load class (assembly) dynamically
JIT Interface

- Compile-only approach
- Custom memory manager
  - Stack unwinding
- Custom module (function) provider
  - Constant pool (Assembly) lookup
From bytecode to LLVM IR

• All JVM bytecode (MSIL) is expressible in LLVM
  • One-to-one translations (e.g. add, sub, div, local loads and stores, static calls ...)
  • One-to-many translations (e.g. array stores, virtual calls, field loads and store ...)
  • Runtime calls (e.g. exceptions, inheritance, synchronizations)
• Type resolution in .Net
  • LLVM Opaque types (for recursive types)
Useful LLVM optimizations

- Mem2reg
  - Move local variables to registers
- Global Value Numbering
  - Load constant values only once
- Predicate Simplifier
  - Removal of array bounds checks
- Loop Invariant Code Motion
  - Load constant values out of the loop
Performance
JVM Benchmarks

- Athlon XP 1800+, 512M, Linux
- 4 JVMs
  - OpenJDK, IBM J9, Jikes RVM, Cacao
- Java Grande Forum Benchmark
  - Section1: low-level operations
  - Section2: scientific benchmarks
- SPEC JVM98
  - Real-world applications
Relative performance to VMKit (higher is better)
No exception optimization in LLVM

GCC's unwinding code

Boehm GC allocator
Garbage Collection? (and synchronizations?)
.Net Benchmarks

- Athlon XP 1800+, 512M, Linux
- 2 .Net
  - Mono, Pnet
- No comparison with Microsoft
- PNetMark
  - Scientific applications
Losing the VM in LL VM
Compilation time

- VMKit uses a compilation-only approach
  - No mixed-mode in LLVM
  - No dynamic optimizations in LLVM

How does that affect application startup?
Tomcat startup (OpenJDK)

INFO: Initializing Coyote HTTP/1.1 on http-8080
Jul 31, 2008 7:17:39 PM org.apache.catalina.startup.Catalina start
INFO: Initialization processed in 1367 ms
Jul 31, 2008 7:17:40 PM org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
Jul 31, 2008 7:17:40 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.16
INFO: Starting Coyote HTTP/1.1 on http-8080
Jul 31, 2008 7:17:40 PM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
Jul 31, 2008 7:17:40 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/44 config=null
Jul 31, 2008 7:17:40 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 1142 ms
Tomcat startup (VMKit w/ Opt)

INFO: Initializing Coyote HTTP/1.1 on http-8080
INFO: Initialization processed in 15020 ms
Jul 31, 2008 6:35:54 PM org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
Jul 31, 2008 6:35:54 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.16
INFO: Starting Coyote HTTP/1.1 on http-8080
Jul 31, 2008 6:36:13 PM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
Jul 31, 2008 6:36:13 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=19/738 config=null
INFO: Server startup in 22660 ms
Tomcat startup (VMKit w/o Opt)

INFO: Initializing Coyote HTTP/1.1 on http-8080
INFO: Initialization processed in 10219 ms
INFO: Starting service Catalina
INFO: Starting Servlet Engine: Apache Tomcat/6.0.16
INFO: Starting Coyote HTTP/1.1 on http-8080
INFO: JK: ajp13 listening on /0.0.0.0:8009
Jul 31, 2008 6:52:00 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=16/679 config=null
Jul 31, 2008 6:52:00 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 17729 ms
<table>
<thead>
<tr>
<th>---User Time---</th>
<th>---System Time---</th>
<th>---User+System---</th>
<th>---Wall Time---</th>
<th>--- Name ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2840 ( 8.0%)</td>
<td>17.5490 ( 52.8%)</td>
<td>18.8331 ( 38%)</td>
<td>19.6628 ( 39%)</td>
<td>X86 DAG-&gt;DAG Instruction Selection</td>
</tr>
<tr>
<td>3.2201 (20.2%)</td>
<td>0.7200 ( 2.1%)</td>
<td>3.9402 ( 8.0%)</td>
<td>4.2831 ( 8.5%)</td>
<td>Unswitch loops</td>
</tr>
<tr>
<td>3.3322 (20.9%)</td>
<td>0.7800 ( 2.3%)</td>
<td>4.1122 ( 8.3%)</td>
<td>4.1940 ( 8.3%)</td>
<td>Predicate Simplifier</td>
</tr>
<tr>
<td>1.1120 ( 7.0%)</td>
<td>0.8240 ( 2.4%)</td>
<td>1.9361 ( 3.9%)</td>
<td>1.9569 ( 3.8%)</td>
<td>Linear Scan Register Allocator</td>
</tr>
<tr>
<td>1.0640 ( 6.6%)</td>
<td>0.8800 ( 2.6%)</td>
<td>1.9441 ( 3.9%)</td>
<td>1.9235 ( 3.8%)</td>
<td>Live Variable Analysis</td>
</tr>
<tr>
<td>0.8200 ( 5.1%)</td>
<td>0.9280 ( 2.7%)</td>
<td>1.7481 ( 3.5%)</td>
<td>1.5910 ( 3.1%)</td>
<td>Live Interval Analysis</td>
</tr>
<tr>
<td>0.6920 ( 4.3%)</td>
<td>0.6760 ( 2.0%)</td>
<td>1.3680 ( 2.7%)</td>
<td>1.3666 ( 2.7%)</td>
<td>Global Value Numbering</td>
</tr>
<tr>
<td>0.6640 ( 4.1%)</td>
<td>0.5880 ( 1.7%)</td>
<td>1.2520 ( 2.5%)</td>
<td>1.2098 ( 2.4%)</td>
<td>Simple Register Coalescing</td>
</tr>
<tr>
<td>0.3480 ( 2.1%)</td>
<td>0.4520 ( 1.3%)</td>
<td>0.8000 ( 1.6%)</td>
<td>0.7449 ( 1.4%)</td>
<td>Combine redundant instructions</td>
</tr>
<tr>
<td>0.1640 ( 1.0%)</td>
<td>0.3400 ( 1.0%)</td>
<td>0.5040 ( 1.0%)</td>
<td>0.5229 ( 1.0%)</td>
<td>Combine redundant instructions</td>
</tr>
</tbody>
</table>

15.8850 (100.0%) 33.1980 (100.0%) 49.0830 (100.0%) 50.2414 (100.0%) TOTAL
## VMKit: Compilation time w/o Opt

<table>
<thead>
<tr>
<th>---User Time---</th>
<th>System Time---</th>
<th>--User+System--</th>
<th>---Wall Time---</th>
<th>--- Name ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0081 (24.9%)</td>
<td>24.4535 (75.4%)</td>
<td>26.4616 (68%)</td>
<td>27.2189 (68%)</td>
<td>X86 DAG-&gt;DAG Instruction Selection</td>
</tr>
<tr>
<td>1.6041 (19.9%)</td>
<td>0.9960 (3.2%)</td>
<td>2.6001 (6.7%)</td>
<td>2.5746 (6.5%)</td>
<td>Live Variable Analysis</td>
</tr>
<tr>
<td>1.2680 (15.7%)</td>
<td>0.9920 (3.2%)</td>
<td>2.2601 (5.8%)</td>
<td>2.3452 (5.9%)</td>
<td>Live Interval Analysis</td>
</tr>
<tr>
<td>1.2320 (15.3%)</td>
<td>0.8440 (2.7%)</td>
<td>2.0761 (5.3%)</td>
<td>2.0690 (5.2%)</td>
<td>Linear Scan Register Allocator</td>
</tr>
<tr>
<td>1.0000 (12.4%)</td>
<td>0.6280 (2.0%)</td>
<td>1.6280 (4.1%)</td>
<td>1.5355 (3.8%)</td>
<td>Simple Register Coalescing</td>
</tr>
<tr>
<td>0.3680 (4.5%)</td>
<td>0.4320 (1.4%)</td>
<td>0.8000 (2.0%)</td>
<td>0.7780 (1.9%)</td>
<td>Control Flow Optimizer</td>
</tr>
</tbody>
</table>

8.0444 (100.0%)  30.7618 (100.0%)  38.8063 (100.0%)  39.5548 (100.0%)  TOTAL
No Parallel compilation

One big lock for JIT (and shared tables)

- JVM (MSIL) to LLVM IR
- Generating LLVM types
- Applying optimization passes
- Code generation
Missing features

- Non-calls exceptions
  - Null and div/0 runtime checks
- Arithmetic overflow
  - Runtime checks
- Bytecode checking
- Type-based alias analysis
Conclusion:
VMKit needs your participation!

• VMKit work
  • Thread optimizations
  • Generational GC with LLVM

• LLVM work
  • Hot-spotting LLVM
  • VM specific optimization passes
  • Non-calls exceptions
  • Compilation times
For more information

http://vmkit.llvm.org

Thank you: Tanya, Ted, ADC France, Apple Inc.