JCOD

A Lightweight Modular Compilation Technology For Embedded Java

Bertrand Delsart : b.delsart@ri.silicomp.fr

Vania Joloboff : v.joloboff@ri.silicomp.fr

Eric Paire : e.paire@ri.silicomp.fr

Silicomp Research Institute

Plan

- Java and JIT compilers
- JCOD solution
- Dynamic profiling
- Experimental results

Java

- Advantages
  - Productivity gain (GC, object oriented)
  - Portability (CPU + OS)
  - Mobility (dynamic loading, security…)
  - Code size
- Drawback
  - **Performance**

Improving performance

- Use native methods for the critical parts
- Generate optimized bytecode
- Replace the interpreter by a dedicated hardware
- Transform the bytecode into native code
  - before installation : Ahead Of Time

  and / or
  - during execution : Just In Time

*SRI works on **both** compilation techniques*

Just In Time compilers

Compile the byte code just before its first execution

- Advantage
  - Preservation of dynamic loading and mobility
- Drawbacks
  - Compilation delay
  - Compilation resources
  - Code size expansion

JCOD solution

- Compilation delay
  ⇒ start in **interpreted** mode
- Compilation resources
  ⇒ compile on a **remote** compilation server
- Code size expansion
  ⇒ optimize the compiled **code size**
  ⇒ **select** the best compilation candidates

***J**ava **C**ompilation **O**n **D**emand*

Remote mechanism

- Support JVM independent compilers
- Use a downloadable JVM dependent plug-in
- Already handle :
  - remote administration
  - remote debugging
  - reconnection

100 % Java

- Extensions : secure protocol, code mobility …

Code size optimizations

- **Favor code size** over performance
- Use a **library** for complex bytecodes (64 bits, lookupswitch, tableswitch…)
- Share **prelude** and **postlude** (synchronization, call traces, stack handling, …)
- Share **wrappers** for external calls (exception delivery, call backs, …)
- …

Method selection

- At least test several policies during our research
- If possible after deployment, allow
  - other policies by third parties
  - tailoring of the policy for a kind application or platform

flexible, **programmable** but **efficient** policies

Profiling Java objet

- Profiling information **efficiently** gathered for one or more methods
- Notification mechanism for some events (enter, exit, "*interesting method*" …)
  - **Programmable** notification listeners
  - **Flexible** notification policy

Decision Maker (in Java)

- Control the profiling mechanism
  - **Associate** interpreted methods with a profiling object
  - Select the notification **policy** of each profiling object
- Handle the profiling information
  - **Asynchronously** parse the profiling objects
  - Provide the **notifications** listeners for each kind of profiling object

Example of policies

- Cruise mode

  select a method only when interpreting more than X bytecode/s

- Exit sampling

  "promote" the N^{th} executed method, or the current method when the interpreter has parsed X bytecodes

- Asynchronous focused profiling

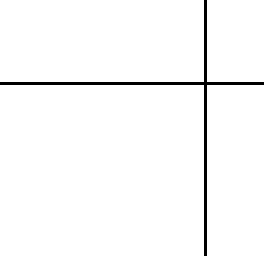
  1) starts with a **single** profiling object

  2) allocate a methodData for ***"interesting"*** methods

  3) **asynchronously** select the best profiling object

Cost of method calls

caller \ callee	interpreted	compiled
interpreted	1	2.5
compiled	1.8	0.2



Lion's share rule

90% of the time is spent in 10% of the code

Compile the "hotspots"

- bytecode inside a **loop**
- methods **called often** (…by a loop)

But detect them without a per method history to reduce the memory consumption

rely on cross calls to identify method called often

Profiling information and notifiers

- increase the considered set thanks to data in a global profiling object (**efficiently** updated on method **exit**) :
  - number of interaction with compiled methods
  - interpretation cost due to loops
  - interpretation cost (for sampling, …)
  - number of execution (for sampling, …)
- decide thanks to per method profiling object and :
  - global interpretation cost (to detect the cruise mode, …)
  - number of interaction with interpreted methods (to avoid cross calls)
- the notifiers receives the caller ID to let complex decision maker build a call graph

Notifiers

- **Global** interpretation cost **overflow**
- Exit notifier :
  - methodData interpretation cost **overflow**
  - After the **N**th executions :
    - NOTIFY_EXIT_**ALL**_METHODS
    - NOTIFY_EXIT_ON_**CROSSCALL**
    - NOTIFY_EXIT_ON_**LOOP**
- (Enter notifier)

Experimental results : overhead

- Small **support code** in ROM on the VM (≈ 20 K)
- Negligible **CPU profiling** overhead
- Small memory overhead for **profiling information** thanks to the focusing techniques
- Small memory overhead for the **compiled code**

- Huge **compilation delay** due to the remote features … but not critical since compilation is asynchronous

Experimental results : efficiency

- Taking the **worst case** for the compiler :
  - Acceleration factor : x4
  - Code size expansion : x3
- And the lion's share rule (90% CPU in 10% code)
  - Performance : **x3**

    (100s $\Rightarrow$ 10 + (90 / 4) = 32.5s)
  - Memory : **x1.2**

    (100K $\Rightarrow$ 90 + (10 x 3) = 120K)

Conclusion

- Efficient profiling
- Low cost solution
- Portable and extensible
  - JVM independent compiler in Java
  - CPU independent JVM resolver in Java
  - High level profiling in Java

Already repackaged to work in AOT mode !