Motivation

- Runtime profile information is important for better optimizations in Just-In-Time compilers
- Low overhead is important for online profiling techniques

We exploit the hardware performance monitor (HPM) of the processor to minimize the profiling overhead

- We study following two important profiles
  - Object creation profile
  - Lock contention profile
    - what class? where in source code?
Hardware Performance Monitor (HPM) of POWER6

- HPM of the POWER6 processor has four programmable performance counters, each of which:
  - **counts** the number of hardware events, such as
    - executed instructions
    - memory accesses
    - cache misses
  - **generates an interrupt** when the counter overflows
    - to sample events with a specified sampling interval
    - two special purpose registers (SIAR, SDAR) to provide the instruction address and the data address that caused the interrupt
HPM-based profiler vs. software-based profiler

- HPM-based profilers (e.g. Oprofile in Linux, Intel’s VTune)
  - to generate profiles for **hardware events** (e.g. cache miss)

- Software-based profilers (e.g. HPROF in JDK)
  - to generate profiles for **Java-level events** (e.g. object creation)

→ We propose HPM-based profiling techniques for **Java-level events** (object creation and lock contention)
  - HPM-based techniques are easy to balance the accuracy and the overhead by controlling the sampling interval of the HPM
Overview of our HPM-based profiler in JVM

- **JVM**
  - (IBM JDK for Java6 SR2)
  - HPM profiler

- **OS**
  - (RedHat Enterprise Linux 5.2)
  - device driver for HPM

- **CPU**
  - (POWER6)
  - Hardware Performance Monitor
How our HPM-based profiler interact with the HPM

1) configure the HPM via the device driver

- set events to count
- set sampling intervals
How our HPM-based profiler interact with the HPM

1) configure the HPM via the device driver

2) record the instruction address (SIAR) and the data address (SDAR) in a buffer

when an interrupt occurs (e.g. once per 10,000 L1D cache misses)
How our HPM-based profiler interact with the HPM

1) configure the HPM via the device driver
3) translate the hardware addresses into Java-level information

2) record the instruction address (SIAR) and the data address (SDAR) in a buffer

- instruction address ➔ Java method
- data address ➔ Java class, field and location (nursery, tenure, or stack)

when the buffer becomes full
How our HPM-based profiler interact with the HPM

1. JVM
   - HPM profiler

2. OS
   - device driver for HPM

3. CPU
   - Hardware Performance Monitor

4) configure the HPM again
Agenda

- Motivation
- Overview
- Object creation profiling using the HPM
- Lock contention profiling using the HPM
- Context-sensitive profiling
- Summary
How we can profile Java-level events using the HPM

Our approach

- **To correlate** Java-level events with the hardware events directly supported by the HPM

- We propose two different approaches
  - for the object creation profile
  - for the lock contention profile
## Existing techniques for object creation profiling

<table>
<thead>
<tr>
<th>Inserting a hook in allocation site</th>
<th>large</th>
<th>classes of created objects and allocation sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting objects in garbage collector</td>
<td>medium</td>
<td>classes of created objects</td>
</tr>
</tbody>
</table>
Object creation profiling using the HPM

Observation

*The virtual function table pointer in the Java object header is modified only at the object creation time*

store for the virtual function table pointer = object creation

three-word object header

object format in our JVM

- VFT
- flags
- lock
- fields
Steps of the object creation profiling

1. To generate a *memory store event profile* using the HPM
   - each sampled event includes the instruction address and the data address
2. To translate the data address of each sampled store event into Java class and the offset in the object
3. To filter out events whose offset is not for the virtual function table pointer in the object header
4. The *memory store event profile* has became the *object creation profile* after the filtering
## Comparing object creation profiling techniques

<table>
<thead>
<tr>
<th>Method</th>
<th>Overhead</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inserting a hook in allocation site</td>
<td>large</td>
<td>class of created objects and allocation sites</td>
</tr>
<tr>
<td>Counting objects in garbage collector</td>
<td>medium</td>
<td>class of created objects only</td>
</tr>
<tr>
<td><strong>Our HPM-based technique</strong></td>
<td>small (depend on sampling interval)</td>
<td>class of created objects and allocation sites <em>for sampled events</em></td>
</tr>
</tbody>
</table>

*no overhead by stopping the HPM to generate interrupts!*
Examples of the object creation profiles

breakdown by the number of objects

<table>
<thead>
<tr>
<th>0%</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counted in GC</td>
<td>Counted in GC</td>
<td>Our profiler</td>
<td>Our profiler</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECjbb2005</td>
<td>SPECjbb2005</td>
<td>compiler.compiler (SPECjvm2008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

software:  RedHat Enterprise Linux 5.2, IBM JDK for Java6 SR2 32 bit

hardware:  4 cores of POWER6 4.0GHz, 16 GB memory (IBM BladeCenter JS22)

See the paper for quantitative evaluations
Examples of the object creation profiles

- java/math/BigDecimal.multiply: 34.8%
- spec/jbb/Orderline.process: 15.5%
- java/math/BigDecimal.setScale: 11.6%
- java/math/BigDecimal.movePointRight: 10.0%
- other methods: 28.1%

- spec/jbb/infra/Util/XMLTransactionLog.populateXML: 52.9%
- java/math/BigDecimal.longString1: 35.6%
- other methods: 11.5%

- spec/jbb/infra/Util/XMLTransactionLog.populateXML: 39.0%
- java/math/BigDecimal.longString1: 55.2%
- spec/jbb/Company.getCustomerByLastName: 2.8%
- other methods: 3.0%

See the paper for quantitative evaluations
Agenda

- Motivation
- Overview
- Object creation profiling using the HPM
- Lock contention profiling using the HPM
- Context-sensitive profiling
- Summary
## Comparing lock contention profiling techniques

<table>
<thead>
<tr>
<th></th>
<th>suitable for</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>software-based profiling</td>
<td>blocking lock</td>
<td>large</td>
</tr>
<tr>
<td>our HPM-based profiling</td>
<td>spin lock</td>
<td>small</td>
</tr>
</tbody>
</table>

- Two profiling techniques are complementary

Smaller overhead is important for lock contention profiling because the overhead may change the lock contentions.
Instrument with the special NOP instruction

Our approach

*insert a special NOP instruction, called ProbeNOP, in the code of interest to correlate with the hardware event*

where,

✓ the ProbeNOP instruction does not affect the program meaning
✓ the HPM can count the number of ProbeNOP execution
Our ProbeNOP Implementation

- *No existing processor has the ProbeNOP instruction*

- We need to find an existing instruction that can be used as the ProbeNOP
  - The instruction does not change the program behavior
  - The HPM can count the execution of the instruction
  - The JVM does not use the instruction

- We used the *vector permute instruction (vperm)* of the VMX (Altivec) instruction on POWER6
  - The HPM can count the execution of the VMX instructions
  - The current JVM does not use the VMX
Profiling lock contentions

```java
while (true) {
    if (tryLock(obj) == SUCCESS) break;
    ProbeNOP;
}
// critical section begins
```

- The profiler can identify the location of the contended locks based on the HPM interrupts
- Additionally, we want to know which object causes the contention

Note: This pseudocode is radically simplified. See the paper for more realistic implementation.
Value profiling with the ProbeNOP

```
monitor enter for obj

while (true) {
    if (tryLock(obj) == SUCCESS) break;
    ProbeNOP(obj);
}
// critical section begins
```

- We encode information on which register or memory location to profile in the unused bits of the ProbeNOP

The interrupt handler decodes the information encoded in the ProbeNOP instruction and collects the values of the specified target.

Note: This pseudocode is radically simplified.
See the paper for more realistic implementation.
### Example of the lock contention profile

<table>
<thead>
<tr>
<th>SPIN LOOP</th>
<th>METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>% samples</td>
<td></td>
</tr>
<tr>
<td>36.9%</td>
<td>com/ibm/ejs/ras/Tr.register(Ljava/lang/Class;Ljava</td>
</tr>
<tr>
<td>14.3%</td>
<td>org/apache/openjpa/meta/MetaDataRepository.getMeta</td>
</tr>
<tr>
<td>6.4%</td>
<td>com/ibm/io/async/ResultHandler.runEventProcessingL</td>
</tr>
<tr>
<td>3.0%</td>
<td>com/ibm/ws/persistence/EntityManagerImpl.createNam</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPIN LOOP</th>
<th>LOCATION</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>% samples</td>
<td>tenure</td>
<td>com/ibm/ws/bootstrap/WsLogManager</td>
</tr>
<tr>
<td>36.9%</td>
<td>(75)</td>
<td></td>
</tr>
</tbody>
</table>

For DayTrader benchmark
Agenda

- Motivation
- Overview
- Object creation profiling using the HPM
- Lock contention profiling using the HPM
- Context-sensitive profiling
- Summary
Context-sensitive profiling for the HPM: Motivation

Motivation

- Methods in collection classes (e.g. HashMap) often cause many cache misses
- just the code location of the cache misses are not informative enough for optimization

→ Calling context for such methods are important

Challenge:
To obtain a calling context information with minimum overhead
An example of the context-sensitive profiling
An example of the context-sensitive profiling

We collect the value of the stack frame pointer in the HPM interrupt handler

See the paper for more detail
Accuracy of the calling context information

L1 cache miss events whose callers were uniquely identified for at least three levels

- SPECjbb2005
- compiler.compiler
- derby
- sunflow
- xml.validation
- serial
- mpegaudio
- DayTrader
Summary

- We showed how our profiler in JVM generates the object creation profile and the lock contention profile using the HPM
  - We correlate the object creations with the store instructions for Java object headers
  - We correlate the lock contentions with the special NOP instruction, called *ProbeNOP*

- We proposed a technique to identify the calling context for each sample using the call stack depth
  - This technique works surprisingly well with many tested benchmarks including a large application server workload
Thank you!
Backup slides
How our HPM-based profiler interact with the HPM

L1D cache miss occurs at 0x1234 (SIAR) when accessing 0xAAAA (SDAR)

buffered in the OS-space buffer

{ 0x1234, 0xAAAA }{ 0x1000, 0xBBBB }{ 0x1000, 0xCCCC }....

method = hashMap.get(),
class = java.lang.String,
field = 1st field,
location = nursery heap

method = hashMap.get,
....
Accuracy of the object creation profiling

![Graph showing accuracy of object creation profiling over different numbers of samples and periods.

- SPECjbb2005
- compiler.compiler
- derby
- sunflow
- xml.validation
- serial
- mpegaudio
- daytrader

The graph plots the overlap metric against the number of samples for different durations:
- 30 seconds, 2,000 samples/sec
- 30 seconds, 8,000 samples/sec
- 240 seconds, 8,000 samples/sec

The accuracy is high, with values close to 1, indicating a good match between the monitored and actual object creation.
Overhead of the profiler

![Graph showing relative throughput with and without profiler](image)

- **Sampling rates**:
  - 8,000 samples/sec
  - 1 sample/msec/HW thread