Measurement: Profiling
Performance Evaluation of Computer Systems

Vojtěch Horký     Peter Libič     Petr Tůma

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University

2010 – 2021
Outline

1 Overview

2 Profiling With Sampling

3 Native Program Location

4 Managed Program Location

5 Profiling With Instrumentation

6 Visualizing Profiling Output
Profiling

Purpose

Profiling collects information about system execution connected (typically) with individual program locations, making it possible to associate performance anomalies with code.

Examples of profiling output:

- List of all executed functions annotated with percentual share of execution time
- A calling context tree depicting all function calls annotated with call counts in each context
- A map of all program locations annotated with the likelihood that a memory access in that location causes a cache miss
Collecting Profile Data

Profiler can collect profile data in one of two ways

- Sampling on asynchronous events
- Instrumentation in locations of interest

**Sampling**

Interrupts program execution at (ideally) random locations and records metric of interest together with location.

- Enough random samples should provide representative information
- Overhead depends on sampling frequency (hence can be regulated)

**Instrumentation**

Inserts probes for collecting metric of interest in important locations such as function entry and exit points or basic block boundaries.

- Accuracy determined only by probe location (no sampling involved)
- Overhead depends on execution patterns (cannot be helped)
Profiling With Sampling

System configured to interrupt the application when some event occurs. The sample is collected on each interrupt.

The event can be generated for example by:
- Periodic timers (hardware interrupt)
- Hardware counter overflow (hardware interrupt)
- Software callback (JVMTI, kernel signal, network stack, ...)

Robustness

For sampling to be useful, the event occurrence should be independent of the metric sampled (unless the events are the metric).

The sample typically consists of:
- Sampled program location
- Possibly sampled metric
Outline

1 Overview

2 Profiling With Sampling

3 Native Program Location

4 Managed Program Location

5 Profiling With Instrumentation

6 Visualizing Profiling Output
Sampling Program Location

Program Counter

Program counter value is typically available on interrupt (because that is where the program execution will resume).

The uses of program counter in profiling:

- Identifies (binary) instruction as sample location 
  (but interrupt can be delayed after sampled event)
- Debug information can convert address to source code location 
  (but debug info not always available and mapping not trivial)
- Stack trace information can provide calling context information 
  (but stack frames are not always in standard format)

Sampling location alone already already gives *hotness profile*. 
Example: Delayed Branch Event Samples

> perf record -e branch-misses <file>
...
> perf annotate --no-source --stdio
...

0.00 : 4010cd: cmpb $0x0,0x404180(%rdx)
6.79 : 4010d4: je 4010c0 <main+0x20>
0.11 : 4010d6: lea (%rdx,%rdx,1),%eax
44.23 : 4010d9: add $0x1,%ebp
 7.95 : 4010dc: cmp $0x7ffffffff,%eax
0.00 : 4010e1: ja 4010c0 <main+0x20>
32.12 : 4010e3: lea (%rdx,%rdx,1),%rax
0.00 : 4010e7: nopw 0x0(%rax,%rax,1)
 2.61 : 4010f0: movb $0x0,0x404180(%rax)
 4.85 : 4010f7: add %rdx,%rax
0.06 : 4010fa: cmp $0x7ffffffff,%eax
0.00 : 4010ff: jle 4010f0 <main+0x50>
...

The only branch instructions in the listing are ja, je and jle, yet most samples are recorded for add and lea.
Example: DWARF Debug Information

> readelf --debug-dump=rawline <file>
...

Line Number Statements:

- [0x0000031d] Extended opcode 2: set Address to 0x4008c6
- [0x00000328] Advance Line by 17 to 18
- [0x0000032a] Copy
- [0x0000032b] Special opcode 103: advance Address by 7 to 0x4008cd and Line by 0 to
- [0x0000032c] Special opcode 230: advance Address by 16 to 0x4008dd and Line by 1 to
- [0x0000032d] Special opcode 103: advance Address by 7 to 0x4008e4 and Line by 0 to
- [0x0000032e] Advance PC by constant 17 to 0x4008f5
...

> readelf --debug-dump=decodedline <file>
Decoded dump of debug contents of section .debug_line:

CU: <file>.cc:

<table>
<thead>
<tr>
<th>File name</th>
<th>Line number</th>
<th>Starting address</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;file&gt;.cc</td>
<td>18</td>
<td>0x4008cd</td>
</tr>
<tr>
<td>&lt;file&gt;.cc</td>
<td>19</td>
<td>0x4008dd</td>
</tr>
<tr>
<td>&lt;file&gt;.cc</td>
<td>19</td>
<td>0x4008e4</td>
</tr>
<tr>
<td>&lt;file&gt;.cc</td>
<td>21</td>
<td>0x400901</td>
</tr>
<tr>
<td>&lt;file&gt;.cc</td>
<td>22</td>
<td>0x40090c</td>
</tr>
<tr>
<td>&lt;file&gt;.cc</td>
<td>23</td>
<td>0x400919</td>
</tr>
</tbody>
</table>
...
## Source Line Mapping

### Ideal

Each source code line corresponds to distinct code block

- Each code block is characterized by program counter range
- Direct mapping between program counter and source trivial

### Real

Source code line is not an execution unit
Optimizations create complex mapping

- Matching to lines with multiple statements loses information
- Macros expanded before compilation not visible to compiler
- Optimizations can change execution order
  (for example invariant code motion)
- Optimizations can break one to one mapping
  (for example common subexpression elimination)

... how does all this impact profiling?
Example: Standard Stack Frame

\[
x = f(1,2);
\]

... 

int f(int a, int b) {
    int c = a + b;
    ... 
}

... 

push 2
push 1
call f

... 

f: push ebp
mov ebp, esp
sub esp, 16
mov eax, [ebp+8]
add eax, [ebp+12]
mov [ebp-4], eax

...
Example: libunwind Stack Walk

```c
#define UNW_LOCAL_ONLY
#include <libunwind.h>

void print_stack () {
    unw_context_t context; // Machine register context.
    unw_cursor_t cursor; // Stack frame reference.
    unw_word_t ip, sp, offset;
    char name[256];

    unw_getcontext (&context);
    unw_init_local (&cursor, &context);

    while (unw_step (&cursor) > 0) {
        unw_get_reg (&cursor, UNW_REG_IP, &ip);
        unw_get_reg (&cursor, UNW_REG_SP, &sp);
        unw_get_proc_name (&cursor, name, sizeof(name), &offset);
        printf ("%s+%lx IP+%lx SP+%lx\n", name, offset, ip, sp);
    }
}
```

Based on code from libunwind documentation
### Stack Walk

#### Ideal
- Location sample includes top stack frame address
- All stack frames in standard format

  - Frame walk as simple as linked list traversal

#### Real
- Top stack frame address not always available
- Some stack frames not in standard format
- Some calls do not have stack frames

  - Samples close to function entry and exit may not have frame pointer ready
  - Handwritten assembly code may use stack in many inventive ways
  - Tail call optimizations do not create frames
  - Stack may not be continuous

... how does all this impact profiling?
Stack Frame Information

Compiler Information

Compilers know how the code uses stack, can they help?

> cat <file.c>
void function () { ... }

> objdump --disassemble <file.o>
...
0000000000000000 <function>:
  0:  55 push %rbp
  1:  48 89 e5 mov %rsp,%rbp
...

> readelf --debug-dump=frames-interp <file.o>
...
  LOC    CFA    rbp    ra
0000000000000000  rsp+8   u    c-8
0000000000000001  rsp+16  c-16  c-8
0000000000000004  rbp+16  c-16  c-8
...
Stack Frame Information II

0000000000000000 <function>:
  0:  55                  push   %rbp
  1:  48 89 e5           mov    %rsp,%rbp

DWARF

<table>
<thead>
<tr>
<th>LOC</th>
<th>CFA</th>
<th>rbp</th>
<th>ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>rsp+8</td>
<td>u</td>
<td>c-8</td>
</tr>
<tr>
<td>0000000000000001</td>
<td>rsp+16</td>
<td>c-16</td>
<td>c-8</td>
</tr>
<tr>
<td>0000000000000004</td>
<td>rbp+16</td>
<td>c-16</td>
<td>c-8</td>
</tr>
</tbody>
</table>

- On function entry
  - Return address at top of stack
  - Registers not yet saved
- First instruction saves RBP on stack
- Second instruction sets RBP to provide canonical frame address
- Rest of function uses canonical frame address in RBP instead of RSP
Dynamic Stack Frame Analysis

Doing More?

What if the frame information is incomplete or incorrect?
We can try dynamic program analysis …

- Exported symbols point to function addresses
- More function boundaries located with heuristic analysis
  - Non conditional control transfers may terminate function
  - Conditional jumps do not cross function boundaries
  - Look for typical frame pointer manipulation
  - Look for symmetrical stack manipulation
  - …

- Linear scan to find stack and frame pointer manipulation
- Stack frame information deducted using more heuristic
- Shown to be over 95% accurate on optimized x86 code

Based on Tallent et al.: Binary Analysis ... doi:10.1145/1543135.1542526
Outline

1. Overview
2. Profiling With Sampling
3. Native Program Location
4. Managed Program Location
5. Profiling With Instrumentation
6. Visualizing Profiling Output
Sampling Managed Languages

What happens when we are sampling program location in a managed language environment (imagine JavaScript, Python, Java)?
Sampling Managed Languages

What happens when we are sampling program location in a managed language environment (imagine JavaScript, Python, Java)?

**Interpreter**

Naive sampling will profile the interpreter rather than the application.
What happens when we are sampling program location in a managed language environment (imagine JavaScript, Python, Java)?

**Interpreter**
Naive sampling will profile the interpreter rather than the application.

**Just-In-Time Compiler**
Naive sampling will profile the compiler together with the application.
Sampling Location in Java

**JVMTI**

Use JVMTI to query high level language program position (used by hprof)?

```c
jvmtiFrameInfo frames[5];
jint count;
jvmtiError err;

err = (*jvmti)->GetStackTrace (jvmti, aThread, 0, 5, &frames, &count);
if (err == JVMTI_ERROR_NONE && count >= 1) {
    char *methodName;
    err = (*jvmti)->GetMethodName (jvmti,
        frames[0].method, &methodName, NULL);

    if (err == JVMTI_ERROR_NONE) {
        printf ("Method %s.
", methodName);
    }
}
```

Based on code from JVMTI documentation
Sampling and Safepoints

Safepoint

Program location where information needed for garbage collection is available.

This interferes with JVMTI program location sampling:

- Stack trace query waits until all threads reach safepoints
- Safepoints at hot locations typically avoided if possible
- Samples therefore biased

Based on Mytkowicz et al.: Evaluating the Accuracy ... doi:10.1145/1809028.1806618
Sampling Location in Java

**JVMTI**

Use JVMTI to know about dynamically compiled code?

JVMTI reports JIT compilation
- What method was compiled
- Where the compiled code is stored

By recording this information, we can do sampling in compiled methods. Not compiled methods are not hot anyway.
Outline

1. Overview
2. Profiling With Sampling
3. Native Program Location
4. Managed Program Location
5. Profiling With Instrumentation
6. Visualizing Profiling Output
Hotness Profiling With Instrumentation

**Basic Block Counting**

- Counter increment inserted at the start of each basic block
- Provides exact execution count for every instruction in the code

This comes with significant overhead. Typical basic block units to tens of instructions long, hence overhead factors of 2 or more are easily possible.
Tool: gprof

Native code profiler integrated with the GCC compiler.
  * Instrumentation at function entry.
  * Periodic sampling with interval timers.

```bash
> g++ -pg <file.c> -o <file>
> ./<file>
...
> gprof <file>
Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>time</th>
<th>seconds</th>
<th>calls</th>
<th>ns/call</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.33</td>
<td>3.33</td>
<td>3.33</td>
<td>1234567</td>
<td>123.45</td>
<td>123.45</td>
<td>frequent_function(int)</td>
<td></td>
</tr>
<tr>
<td>9.99</td>
<td>0.99</td>
<td>0.99</td>
<td>123456</td>
<td>12.34</td>
<td>12.34</td>
<td>less_frequent_function(int)</td>
<td></td>
</tr>
</tbody>
</table>
...```
Tool: OProfile

Native code profiler with legacy kernel support.

- **operf** Collect program profile.
  Remotely similar to `perf record`.

- **ocount** Collect counter values.
  Remotely similar to `perf stat`.

- **ophelp** Display available counter information.

- **opreport** Interactively examines profiling results.

- **opannotate** Annotates source code and disassembly with profile.

Also provides JVMTI agent for profiling JIT compiled code.

```
operf java -agentlib:jvmti_oprofile <command>
```
Outline

1. Overview
2. Profiling With Sampling
3. Native Program Location
4. Managed Program Location
5. Profiling With Instrumentation
6. Visualizing Profiling Output