CMP SCI 635: Modern Computer Architecture

Professor Charles Weems

Computer Performance Microscopy with SHIM

Name: Kunjal Panchal

Student ID: 32126469

Date: 12th Sept, 2019

Paper: 3 – Methodology [Yang15]

Strengths:

1. SHIM does not address observer effects due to invasive instrumentation. SHIM induces secondary effects by contending for hardware resources with the application. This effect is largest when SHIM shares a single core with the application using Simultaneous Multithreading (SMT) and contends for instruction decoding resources, local caches, etc. But the SHIM observer thread offers a constant load, which does not obscure application behaviour and makes it possible to reason about SHIM's effect on application measurements. When SHIM executes on a separate core, it interferes much less. A no caching read simply transfers the value without invoking the cache coherence protocol.

2. If a software is running on multiprocessor environment. While one core being totally unused by that particular software, SHIM overhead is just 2%, despite the sampling rate being ~1200cycles, it is much finer than the fastest rate of 30K cycles of most of the profiling tools, which instrument software events and reads hardware counters.

3. The hardware is naturally inducing large amounts of randomness in SHIM's sampling, and thus SHIM avoids the sampling bias problem due to regular intervals.

4. Compared to hardware interrupts (which throttles throughput greatly), direct measurements (increases observer effects while perturbing the code), software profilers (observer effect plus they do not corelate hardware and software events), simulators and emulators (Instead of measuring hardware performance counters, they instrument code and emulate hardware. They are therefore unsuitable for correlating fine-grain hardware and software events); SHIM use periodic sampling to identify and then target hot code for aggressive dynamic optimization. SHIM provides a high resolution, low overhead profiling mechanism that lends itself to feedback directed optimization.

Weaknesses:

1. SHIM also suffers from not having the rudimentary hardware intrinsic on which it will profile the software. This is the same problem most of the simulator designers face, not being able to utilize most of the hardware means the researchers can't verify SHIM's fine-grained accuracy.

2. Adding a bounded buffer might add some overhead (of checking the bounds every time) to manage the classic producer-consumer problem. {Yes, this is proven in Section 6.2}

3. It's worth noting that sampling, at any frequency, is going to miss waiting on (for example) software locks. One recommendation for this is to timestamp based on wall clock (not CPU clock), and then try to find the underlying causes of unusually long waits.

Questions/Assertions:

1. How exactly can the profiling data derived from SHIM, improve the way the software and/or hardware is designed? Do coders/designers know how to handle such fine-grained performance analysis? Does it really help them to study the behaviour of their system at such scale which might not be in their power to manipulate?

2. According to Figure 1(a), how can some methods be sampled more frequently at lower(slowest) frequencies? In other words, how can the red/blue bars go higher than green curve's y-dimension at that x-coordinate?

3. Support for Jikes RVM no longer being available.

4. Does there exist a way or some new research that can optimize SHIM observer overheads? Separate CMP cores seem to work? Maybe some new hardware which has a new take on SMT?

5. What can be the downside of revealing one core's private hardware performance counter to another cores?

6. From an excerpt by Dan Luu, a former Google TPU and Microsoft Bing engineer:
   "*I think SHIM approach is really neat, but considering the current state of things, you can get a pretty substantial improvement without much cleverness. Fundamentally, all you really need is some way to inject your*

tracing code at the appropriate points, some number of bits for a timestamp, plus a handful of bits to store the event.

Say you want trace transitions between user mode and kernel mode. The transitions between waiting and running will tell you what the thread was waiting on (e.g., disk, timer, IPI, etc.). There are maybe 200k transitions per second per core on a busy node. 200k events with a 1% overhead is 50ns per event per core. A cache miss is well over 100 cycles, so our budget is less than one cache miss per event, meaning that each record must fit within a fraction of a cache line. If we have 20 bits of timestamp (RDTSC >> 8 bits, giving ~100ns resolution and 100ms range) and 12 bits of event, that's 4 bytes, or 16 events per cache line. Each core has to have its own buffer to avoid cache contention. To map RDTSC times back to wall clock times, calling gettimeofday along with RDTSC at least every 100ms is sufficient.

*Now, say the machine is serving 2000 QPS. That's 20 99%-ile tail events per second and 2 99.9% tail events per second. Since those events are, by definition, unusually long, Dick Sites recommends a window of 30s to 120s to catch those events. If we have 4 bytes per event * 200k events per second * 40 cores, that's about 32MB/s of data. Writing to disk while we're logging is hopeless, so you'll want to store the entire log while tracing, which will be in the range of 1GB to 4GB. That's probably fine for a typical machine in a datacenter, which will have between 128GB and 256GB of RAM.*"