CMPSCI 691ST Graduate Systems **Fall 2011**

Lecture 20

Lecturer: Emery Berger *Scribe: John Vilk*

20.1 Performance Analysis with Profilers

We read the paper for `gprof`, which is a sampling-based performance profiler that takes samples at regular intervals to determine the average amount of time spent in each function. However, there is a more accurate profiler called `oprofile` that uses performance counters on the CPU. These counters can be configured to trigger interrupts, which lessens the observer affect.

20.1.1 Detour: Java Profilers

At a recent talk, Peter Sweeney discussed the implementations of various Java profilers, and described how they produce inaccurate results. Java has these things called "yield points" where garbage collection is safe to perform. All of the tested profilers in his paper, "Evaluating the Accuracy of Java Profilers", would only collect samples at these yield points, which heavily biased their output.

In order for a sample-based profiler to produce accurate output, it must:

- Collect a large number of samples to get statistically significant results.
- Sample in a way such that all points in a program have an equal probability of being sampled. Not doing so will bias profiling certain points in the program over others.

The Java profilers do not satisfy the second requirement. The second requirement can be fulfilled by sampling at random intervals. Note that `gprof` does not satisfy the second requirement either since it samples at regular intervals, which could bias the result if it synchronizes with an activity in the program or the system.

As a result, the paper found that the most popular Java profilers did not produce the same output, and would even disagree on the 'hottest' method in some cases.

In class, we joked about Java culture, 'abstracting away abstractions' and how it makes analysing performance difficult, and all other sorts of things.[1]

20.1.2 Profiling Object Oriented Programs

In object-oriented programs, calling a function on an object requires a lookup at runtime in a function table to find the correct function to call with the same signature, since it could be in that object's class or a parent class, which adds overhead during runtime. C++ explicitly requires the `virtual` keyword when defining functions that can be overridden in child classes. Using a simple cache that remembers the last version of the method called can greatly speed up this lookup, and is what most JVMs do during runtime.

[1] See `http://www.classnamer.com/` for more fun.

20.1.3 Concluding Discussion on `gprof`

Essentially, profilers are a tool used to improve performance in programs. It is most important that they produce output useful to this end, rather than 100% accurate output.

`gprof`, like most profilers, suffers from the probe effect – which is the effect instrumentation has on program behavior. The probe effect can cause cache misses, which can destroy performance.

One huge limitation of `gprof` is that it doesn't work on multithreaded programs.

20.2 Path Profiling

The path profiling paper does not make a decent argument for path profiles in the first place. They tried to benchmark the accuracy of their tool by comparing it to an algorithm that constructs path profiles from edge profiles, which does not make a decent argument for path profiling. Aside from that, it is amazing that they were able to get the cost of path profiling down as much as they did.

One interesting application of path profiling is using it during runtime to guide JIT optimizations. Mozilla's JavaScript engine constructs traces of JavaScript programs, and uses that information to determine optimizations, such as what functions to inline. This strategy does not work well for big traces, which actually do occur in JavaScript.

Essentially, there are two types of procedural analysis: intra-procedural analysis, which analyses functions independently, and inter-procedural analysis, which analyses the interrelations among functions (of which there are usually an exponential amount). Inter-procedural analysis is sensitive to the context and the call site. Both types of analyses are flow sensitive.

20.3 Profiling Multithreaded Programs

As mentioned before, `gprof` does not work with multithreaded programs. It emits a call to `penter` at the start of each function and `pexit` at each function's end. These are globally entered functions – which does not work for multithreaded programs.

The type of profiling described in "Effective Performance Measurement and Analysis of Multithreaded Applications" is different from `gprof`, but programmers need *something* since `gprof` does not work for multithreaded programs.

One issue with the approach in this paper is that it atomically decrements a counter for the number of working / nonworking cores. While this operation should not happen too often (since if the workload is too fine-grained, the startup cost will kill performance), it can still cause *false-sharing* problems.

20.3.1 False Sharing

If you have multiple cores that share a cache, it becomes very difficult to reason about the performance of multithreaded programs. The problem is exacerbated by false sharing. When a core requests something from memory (e.g. 'give me object Foo'), it actually receives a *cache line* of data (e.g. 64/128 bytes of data). The cache coherency protocol marks this cache line as *exclusive* if it is only in the cache of one core. If another core requests this cache line (which could be for Foo or something entirely different that is adjacent to Foo in memory), it becomes *shared.* If one core updates a shared cache line, it must notify the other cores to *invalidate* the data; those cores must fetch the modified version from memory when the cache line is needed next. Intel calls this MESI – Modified Exclusive Shared Invalid, the four states that a cache line can be in. False sharing can cause multithreaded programs to thrash if multiple threads are accessing and modifying data that are close together in memory.

Hardware implementation details are still kept secret, which frustrates systems researchers. The implementation of things like prefetchers in chips can even secretly change within chip families, which can make reproducing others' results very difficult.

20.3.2 Back to the Paper

Going back to the "Effective Performance Measurement and Analysis of Multithreaded Applications" paper, there are some issues with the metrics suggested in the paper. Charlie raised the point that if a core is stalled because it has no work to do, it could just be stalled due to I/O rather than waiting for a lock acquisition. Also, according to the metrics, if spin locks are used the program has 100% utilization of the parallel power of the CPU. He also doubts that the implementation generalizes to `pthreads`, and seems more Cilk-specific.

Regardless, what we did agree is that this tool is not the next `gprof`, and that a new `gprof` is needed for parallel programs.