CMPSCI 630 Systems **Spring 2017**

Lecture 17

Lecturer: Emery Berger *Scribe: Jayanth Parameshwar Hegde,Daniel Sam Pete Thiyagu*

GProf

Terminology:

Static call graph: Generated using static analysis. This graph depicts all possible function calls in an execution of the program and the arrows indicate the flow of control.

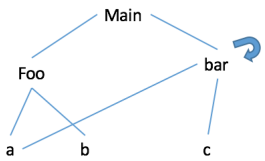
Consider the following program snippet:

```
void foo(){
    a();
    b();
}

void bar(int n){
    if(n!=0){
        c();
        a();
        bar(n-1);
    }
}

int main(){
    foo();
    bar(10);
    return 0;
}
```

Static call graph for the above program:



Dynamic call graph: Generated from an execution of the program. Nodes represent only the function calls observed during this particular execution and the arrows indicate the flow of control. The profiling is more detailed.

Dynamic call graph $\subseteq$ *staticcallgraph*

Probe effect: Distortion in measurement due to the act of measuring. Example: In order to profile a function, one might copy all parameters to a storage. When the size of parameters is large or the number of parameters is large, this copying might evict lot of information of caches. This leads to cache misses worsening the performance of the program under profiling.

Solution to probe effect: Arnold-Ryder approach (based on sampling) The program counter values are sampled at regular intervals of time. Using the distribution of these sampled values, execution times are inferred. This mitigates probe effect and cost of profiling.

Problems with sampling: Applications with heartbeat messages or alternate busy and idle intervals can lead to wrong profiling. The idle times may coincide with the sampling time and cause the profiler to capture nothing. This is solved using Nyquist's theorem where you should always sample double the frequency.

Perf is a Unix profiling tool based on sampling. **Perf Tool** It is an on-chip ARM counter and can count cache misses at every level, branch mispredictions, instructions retired. It records the call stack.

For example let us consider call stacks like

main() $->$ foo() $->$ a()

main() $->$ foo() $->$ b()

the sequence of calls are that a is called first and b is called 99 times , and a is then called again once and b is called 99 times again and so on. Say if we have the countdown timer that checks every 100 times, then our profiler would say that a is the most frequently called code and you would need to optimize the function a , whereas the function b is the most frequently called. Only if it is uniformly distributed, this(countdown timer approach) works well. So sampling should be random.

Gprof profiling:

Gprof produces flat profiles. A flat profile makes measurements of a function execution independent of the callee. Thus, it is path/context insensitive. Alternatively a path sensitive profiling breaks down measurements with respect to the path taken. Depending on the path taken in a particular execution, the measurements can vary. Also flat profiling looks at every function independently and it doesn't account for hierarchy.

Some other notes:

- Gprof cant profile multi-threaded code
- Probe effect is seen with Gprof
- The calls that arise to the kernel space from the user space is not profiled.
- Sampling based profilers were state of the art before coz.
- In multithreaded profilers, we have critical path, longest path. Making improvements on the critical path may not be worth doing if improvements dont reflect in increase in performance.

Notes on stabilizer, randomizations

This was a continuation of some of the topics discussed in the previous lecture.

Experimental Setup Randomization :

Let us consider link object ordering, and say there are 4 object files and we have 4! possible orderings and a program like google chrome has around 10000 object files and it is impossible to do such testing. So experimental setup randomization helps us to get an approximation based on average sampling.

Randomization hurts locality. The objects wont be stored contiguously. But inside an object/page, locality wont be impacted. The overhead of stabilizer arises from the fact that TLB(mapping from virtual to physical pages) is impacted and TLB misses are costly and degrades performance.

Byzantine Problem

The need for Distributed systems started with the creation of the World Wide Web in 1993

In an asynchronous distributed consensus system, there are no leaders and everyone is equal.

For distinguishing between message failure and delay we need to have a bounded delay and this can also involve getting repeated messages. Things can fail and there could be traitors who could also decide to be evil. The most possible general failure model includes forging messages. If there are m number of malicious actors and 3m+1 loyal good guys then it is impossible to forge messages. In general you dont know the number of malicious users. Byzantine generals is used for security and reliability.

Byzantine Fault Tolerance Handling independent random faults