CMPSCI 630 Systems **Fall 2019**

Lecture 19

Lecturer: Emery Berger *Scribe: Deepali Garg*

19.1 Profilers

The class began with a conclusion from the previous class which covered profiling.

Types of profiling

Instrumentation based - have a big probe effect, expensive, slow things down.

Sampling-based - like gProf which stop program periodically to find hotspots in program.

Performance counter-based - to find what's going on like branch prediction failures etc.

Causal profiling - experiment based, give actual causality, different from correlation of other programs.

They address problems at a single node. What about distributed systems? One of the challenges with distributed systems in a shared clock. There is currently no synchronized clock. If there was one, everything becomes trivial. E.g. Fischer lynch Paterson's result- it's impossible to have an agreement even on the value of one bit in a dist system. The consensus is impossible. Time is a huge problem. Drift happens even when clocks begin at the same time.

Network Time Protocol has known time servers. It helps maintain synchronization between servers. Another type is the Atomic Clock which is a perfect timekeeper. It has a radioactive substance that decays at a fixed rate and it is based on it. It provides a quasi-synch time. Google uses it, considering events happening in the same window as occurring at the same time.

A Lamport clock helps to maintain global order.

But time isn't the only problem. How do we get performance information? How do we handle network delays, caching, etc? There's a lot of variabilities. e.g. Memcached is used in Facebook for caching. The PHP API call resolution in the backend is intensive, so we like to keep the data in the cache.

Profiling dist systems is an open question because of these reasons.

Profiling mixed languages is complex. In JS compiler generates code. The problem is mapping compiled code with source code. JIT-compiler optimizes code over and over which makes it difficult. It moves from slow(snail) to fast(rabbit) territory. Python runs interpreters, C runs libraries. We don't want to go inside them and asking them to speed things up.

19.2 Correctness

Correctness is hard. There's no randomness in the occurrence of bugs later. It is monotonically increasing and compositional. If there's a bug or there's correctness, it accumulates and builds.

The program refines a specification which can lead to a subset of different programs. The challenge is that

specifying full correctness is very hard for programs. Partial correctness is with respect to some part of the program. E.g. a property that it doesn't dereference NULL. A bug checker goes to find if a program has a bug. an always error is something like dereferencing null or memory safety error.

Full correctness is called verification. It has been applied to domains like security and very narrow API where the specification is very simple. These domains are important and have small specs. E.g. device drivers in operating systems. Microkernels that have been verified. L4 in embedded systems.

OpenSSL is used for establishing secure connections. It has not been verified. It was vulnerable to the heartbleed bug. Microsoft embarked on the Everest project that wrote a formally verified implementation of OpenSSL. They wrote it in F* that compiles to C and runs it. Building high assurance software is expensive and thus only applies to things that really matter.

Correctness does not imply that cryptographic measures are correct. Testing can prove the presence of bugs but not their absence. The state space is so large that only after exhaustive testing you can prove correctness.

19.3 Verification

Verification is hard. The specification is something that is infinite and it is correct with respect to some other specification.

For most software, we can't verify things, so we do dynamic analysis, static analysis, and testing. The difference is that dynamic is when the program runs, static executes program symbolically. Dynamic is more precise with low false positives. It has a low recall, so it can miss out on bugs that are not exposed during execution. Static analysis is imprecise with high false positives since it works on approximations.

Sound - if there's any possibility of an error, report it. E.g. potential divide by zero occurrences is reported in soundness. Dynamic analysis already does this. In theory, you can't test everything so static analysis is way better. But false positives are a killer.

Coverity is a static analysis tool company. It has a free service but its not so good.

Heroic static analysis is when you throw everything at it.

How can we reduce false positives yet maintain soundness? It is a big research issue right now.

Dynamic analysis sounds bad but they tell you exactly where the problem is. There's a sanitizer by Google where you can plug in different things, it can be ASan, TSan, etc. It will insert checks for every single reference.

There's a tool from Intel called Pin that does dynamic analysis. Valgrind is the epitome of the heavyweight way of doing this. But it is very slow and is mostly used to check memory safety. ASan is a bit faster.

GC is a kind of dynamic analysis but it eliminates errors instead of reporting them. The Exterminator system detects memory errors and fixes the program. It injects something into the runtime system.