The Problem

A Program is taking too long to execute, we want to know:

- **Where** in the program most of the time is being spent (also known as *Hot Spots*)
  - functions
  - loop nests

- **Why** these parts of the program are taking too long to execute.
  - too many instructions
  - too many cache misses
  - too many TLB misses
  - too many page faults
  - too many I/O requests
Potential Approaches

- Static Program Analysis
- Manual Profiling
- Automatic Profiling

Static Program Analysis

- Try to detect hotspots by looking at the code
- Challenges
  - Complexity
    - Large scale software systems often consist of hundreds of components, only a small number of which are potential hot spots.
  - Insufficient Information
    - The actual behavior of the programs depend on user input (examples: executed code paths, the number of loop iterations, the size of the data structures).
Manual Profiling

- Insert timing code around potential candidates for hot spots

```
Count1 = Read (CYCLE_COUNTER);
for (i = 0; i < N; i++) {
    compute();
}
Count2 = Read (CYCLE_COUNTER);
CyclesSpent = Count2 - Count1;
```

- Challenges:
  - You need to understand the entire program before coming up with a set of candidate hotspots
  - Too many potential candidates

Automatic Profiling

- Automatic Code Instrumentation
  - *Static*
  - *Dynamic*

- Program Counter (PC) Sampling
  - Time-based
  - Event-based
Automatic Code Instrumentation

- Automatically instrument regions of programs:
  - the beginning and the end the functions (or methods)
  - function (or method) invocations (to build a call graph)
  - loop nests
  - other programming constructs
    - OpenMP
    - MPI

Time-based PC Sampling

- **Basic Operation**
  - Set up a timer
  - Upon every triggering of the timer, interrupt program execution (e.g., send a signal)
  - At the interrupt handler, record the current PC (with some context)
  - Use program symbol table to map collected PCs to program constructs (e.g., modules, functions, loops, etc.)

- **Basic Rationale:** The more PC samples you collect for a function, the more time you must have spent in there.

- **Key Advantage:** It doesn’t require any change in the executed source code or binary.

- **Example Tool:** Linux’s `gprof`
PC Sampling Example

source: AMD

Performance Modeling and Evaluation, Shiraz University, Fall 1388, by Reza Azimi
Event-based Sampling

- Similar to Time-Based Sampling, but the PC Samples are collected at every N occurrence of an event:
  - Cache Miss
  - TLB Miss
  - Branch Misprediction

- Basic Rationale
  - The number of PC samples you collect on a piece of code for an event is proportional to the number of events that code is causing to occur.

- Key Advantage
  - It enables CPU bottleneck analysis
  - For more info look at: J. Anderson et al., Continuous Profiling: Where Have All the Cycles Gone?, in SOSP 1997
Instruction-based Sampling

- **Problem:**
  - In a super-scalar processor Event-based PC Sampling can be slightly inaccurate
    - Multiple instructions are in flight at a time
    - The instruction that has caused the event counter to overflow cannot be tracked at the interrupt handler.

- **Solution:**
  - Monitor individual instructions as they pass through the pipeline and count the events they’re causing
  - Requires extra support from hardware performance monitoring unit (PMU)

- For more info, look at:

---

Profiling for Just-in-Time Compilation

- For further info, look at:
Profiling Multithreaded Software

- **Per-thread Profiling**
  - show the CPU spent in each individual thread
  - show the number of performance event each thread is causing

- **Profiling Semi-Busy Time**
  - time spent to execute lock acquiring/release
  - time spent spinning on locks
  - time spent to send/receive messages to other threads

- **Profiling Threads Wait Time**
  - time spent for each thread to wait for the others
  - time spent for each thread to (re)fetch shared data that is invalidated by the others into cache.

Some Famous Code Profiling Tools

- OProfile
- Perfmon2
- Intel’s VTune
- AMD’s CodeAnalyst
- PAPI
Profiling OS Activities

- Tracing all system calls
  - `strace`

- Page Faults, Interrupts, I/O
  - `vmstat`

- Network Activities
  - Packets sent/received, number of open connections
  - `netstat`

- Other Statistics
  - Linux’s `/proc` File System

Performance Improvement Techniques

- I/O Optimization Techniques
  - Use parallel I/O (example: database file and its log file on two different storage devices)
  - Changing File Access Method
    - Sequential accesses are faster because of lower seek time (on magnetic disks) and prefetching opportunity
  - Use asynchronous I/O
    - free up CPU for other tasks while the I/O request is being processes
Performance Improvement Techniques

- Code Optimization Techniques
  - tiling
    - divide multi-dimensional arrays into smaller tiles so that each tile fits in cache
  - prefetching
    - insert prefetch instructions ahead of certain load/store instructions
  - inlining
  - loop coalescing
  - code straightening

Further Reading

- J. Anderson et al., Continuous Profiling: Where Have All the Cycles Gone?, in SOSP 1997
- E. Duesterwald et al., Characterizing and Predicting Program Behavior and Its Variability, in PACT 2003.
- T. Sherwood et al., Phase Tracking and Prediction, in ISCA 2003