Continuous Profiling: Where Have All the Cycles Gone?

Jennifer M. Anderson, et al

digital
Systems Research Center
Palo Alto, California

Presented by Igor Pechtchanski

Background: Profiling

• Gathering information about dynamic execution of the program

• Mechanisms
  – simulation
  – instrumentation (compiler or binary)
  – sampling

Background: Profiling (continued)

• Granularity
  – function, basic block, instruction

• Can be used by
  – programmers directly, for hand-tuning code
  – optimization tools
    • compilers, linkers, post-linkers, run-time tools
  – analysis tools

Profiling: Example Use

• Trace construction
  – a trace is a “hot” path through a program
    • makes sense to optimize more
  – use basic block execution counts to determine “hot” blocks
  – profiling makes much more exact
  – especially useful in Java where blocks can have lots of side exits (exceptions)
DCPI: DIGITAL Continuous Profiling Infrastructure

1. Data Collection Subsystem
   - low overhead, system-wide, instruction-level
   - samples program counters and other info
   - records data in a database

2. Suite of Analysis Tools
   - generate histograms at different granularity
   - factor sample info into frequency and latency
     - with possible explanations for stalls

Data Collection System

- Samples PC (program counter) on each processor
- Very efficient
  - uses 1-3% of the CPU
- Relies on Alpha performance counters
  - services interrupts and captures interrupted instruction and context
- In time, provides accurate statistical picture for each instruction

Alpha Performance Counters

- Count events
  - CYCLES, IMISS, DMISS, BRANCHMP
- Generate interrupts when counter overflows
  - delivers PC of next instruction and event type
- Limited number of events can be monitored simultaneously
  - need to multiplex among events
Sample Information

- Frequency: ~ every 64K cycles (randomized on 21164)
- Latency: delivered in 6 cycles
  - if stall long enough, will get accurate PC
  - if not, useful info can be lost
- Blind Spots: deferred interrupts
  - non-interruptible or high-priority code
  - samples will show PC of next instruction

Kernel Device Driver

- Handles interrupts
- Provides `ioctl` interface for user program
- Must be efficient
  - High interrupt rate ⇒ fast handler
  - Limit memory bandwidth

Driver Efficiency Issues

- Associative counters (per sample)
- Minimize cache misses through data structure organization
- Reduce writes to shared cache and synchronization
- Specialize interrupt handler for run-time constants
Driver Data Structures

- Private per-processor
- Modified mostly by processor interrupt routine
- Can be read and modified by flush routines
  - need synchronization
  - use expensive IPI (inter-processor interrupt) during flush

Driver Data Structures (continued)

- Hash table for associative counters
  - four-way associative; array of buckets
    - each bucket = 1 cache line
  - evict capacity misses to overflow buffers

User-Mode Daemon

- Extracts samples from driver
  - user-controlled or time-based
- Associates samples with images
- Stores profiles on disk
Daemon Profile Database

- On-disk database
  - user-specified directory
  - may be shared
- Organized into non-overlapping epochs
  - one sub-directory per epoch
  - one file per image/EVENT combination
- Compact binary format
  - can be further compressed

Image Information Access

- Modified dynamic system loader
  - notifies daemon of image loads by processes
  - captures all dynamically loaded images
- Special recognizer routine for `exec`
  - captures information about all static images
- Mach-based process mapping system calls
  - daemon scans active processes
  - captures all processes already active at startup

Profiling Overhead: Time

- Device driver:
  - deliver interrupt (~214 cycles)
  - service interrupt (~400 cycles)
    - eviction rate
- User-mode daemon
  - read samples and merge into database
    - eviction rate ?!
Profiling Overhead: Space

- Device driver (per processor):
  - hash table (~16k samples @ 16 bytes)
  - 2 overflow buffers (~8k samples @ 16 bytes)
- User-mode daemon
  - buffer for hash table and 1 overflow buffer
  - data structures for active processes and images
  - overhead may be reduced by tuning

Performance Improvements

- Device driver (10-20%):
  - change hash table from 4-way to 6-way
  - use stack-based algorithm for eviction/hits
- User-mode daemon (~2x):
  - sort samples by PID and PC
  - map kernel buffers into daemon address space
  - generally optimize daemon code

Data Analysis Tools

- Given sample count $S_i$, produce
  - frequency $F_i$: number of times instruction executed
  - $cpi C_i$: average cycles at head of issue queue for each execution
  - set of culprits: possible explanations for stalls (wasted issue slots)
- Note: $S_i = F_i \times C_i$

Estimating frequency

1. Build control-flow graph (CFG) for procedure
2. Group basic blocks and edges of CFG into equivalence classes by frequency (known)
3. Estimate frequency of each equivalence class
4. Use local propagation based on flow constraints to propagate estimates around CFG
5. Use heuristic to predict accuracy of estimates
Building CFG

- Extract code for procedure from image
- Identify basic block boundaries
  - using branches and merges
- Attempt to predict targets for indirect branches
  - using preceding code
  - if can’t, note missing edges

Frequency Equivalence Classes

- Determining frequency equivalence
  - Identify sets of blocks guaranteed to execute same number of times
    - cycle equivalence algorithm extended to handle infinite loops
- Estimating frequency from sample counts
  - Schedule each block using processor model
    - static stall $M_i$ is 0 for most instructions
    - issue point is an instruction with $M_i > 0$
  - Average $S_i / M_i$ for some issue points in a class

Analyzing Estimates

- Propagating estimates through CFG
  - use flow constraints
    - similar to dataflow equations
    - use standard iterative algorithm used in compilers
- Predicting accuracy
  - use a heuristic to rate estimate low, medium or high confidence
    - standard statistical approaches

Instruction Frequency Errors
Continuous Profiling: Where Have All the Cycles Gone? Igor Pechtchanski  October 13, 1999

Edge Frequency Errors

Identifying Culprits

• Culprit: why the stall occurred
• Possibly multiple reasons:
  – Static stalls:
    • data dependences
  – Dynamic stalls:
    • I-cache, D-cache and TLB misses
    • branch mispredictions
    • write-buffer overflows
    • functional unit congestion

Future Directions

• Profile-driven optimizations
  – to exploit fine-grained analysis results
• “Continuous run-time optimization”
  – compile-time, link-time, binary rewriting
• Optimize and extend DCPI
  – instruction interpretation
  – double sampling
• Develop tool GUI and visualization
• Hardware support for out-of-order profiling