The Dicer

Differential profiling for micro-architectural performance optimization opportunities

Goals of performance analysis

1. Compiler optimizations
   a. deterministically compute optimal machine code
   b. accurately estimate quality of service
   c. evaluate new optimization techniques
      i. precisely compare against existing approaches
      ii. detect problematic special cases
2. Assist programmers with optimization and debugging
   a. detect poor utilization of hardware resources
   b. compare semantically equivalent versions of a program
3. Automated program transformations
   a. e.g., dynamic function specialization

Performance modeling: the ideal approach

1. Concise abstraction of a complete simulation
   a. Correlates code elements with hardware execution paths
   b. Indicates throughput and latency of hardware components
   c. Accounts for rules that preserve execution correctness
2. Limited effectiveness in practice:
   a. Requires detailed knowledge of hardware behaviors
      i. Rarely published, difficult to discover or infer
   b. Actual performance may depend on minute details:
      i. Component functionality and capacity are often CPU-specific
      ii. Circumstantial effects can become chained (domino effect)
         1. e.g., cache conflict may lead to a pipeline stall
   c. Special cases may have detrimental effects

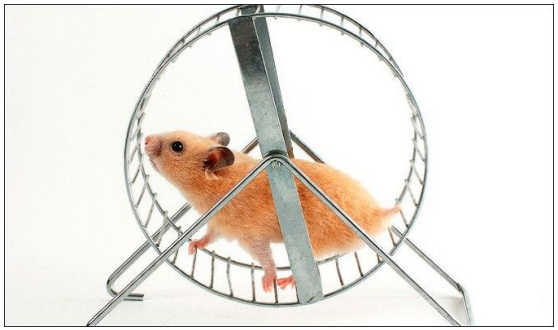
Performance profiling: the practical alternative

1. Precise: reports real statistics about real executions
    a. Does not rely on generalizations or speculative ideas
    b. Easily updated after hardware/firmware changes
    c. Often can be automated with simple, generic tools
2. Limitations of profiling in practice:
    a. Results can be highly specific to the profiled environment
    b. Detailed profiles may be difficult to interpret
        i. Often too verbose for human readers
        ii. Usually too vague for use in automated tools
    c. Offers no guidance for improving performance behaviors
        i. Essential problems may remain hidden
        ii. Requires complex additional steps to reach conclusions

A spectrum of performance analysis

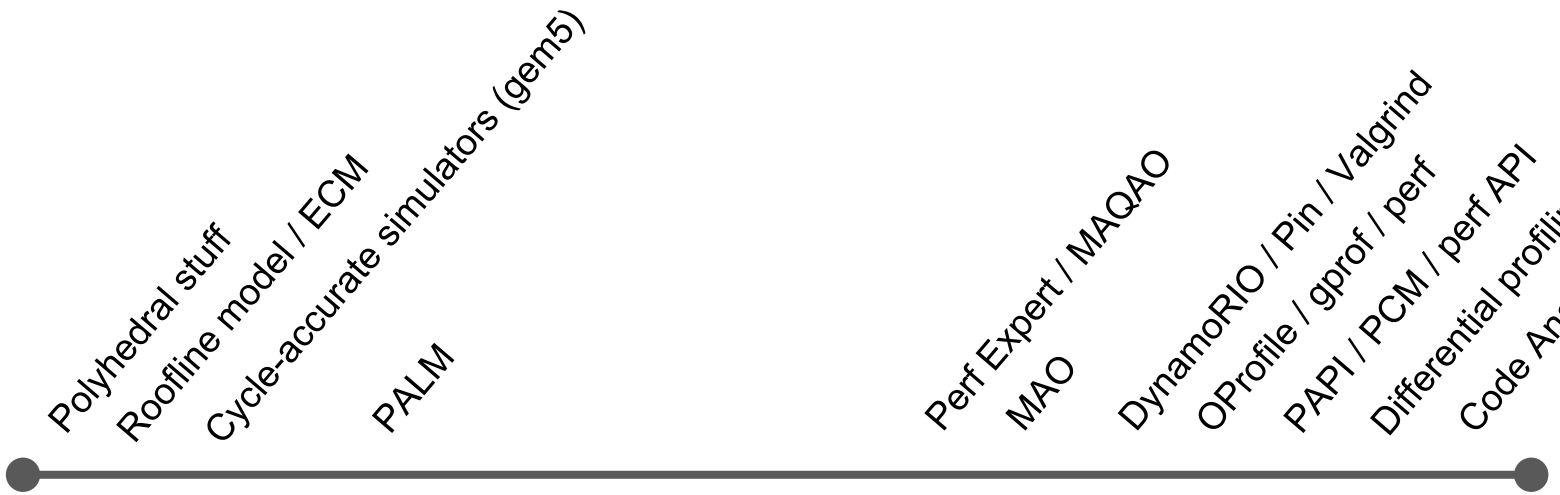
Modeling — Profiling

the holy grail



the routine workaround

Popular performance analysis tools



Modeling

Profiling

The Dicer: self-adapting hybrid

Polyhedral stuff
Roofline model / ECM
Cycle-accurate simulators (gem5)
PALM
MAO
Perf Expert / MAQAO
DynamoRIO / Pin / Valgrind
OProfile / gprof / perf
PAPI / PCM / perf API
Code Analyst

Modeling — Profiling

The Dicer

The Dicer: trivial machine model

1. Recognizes basic optimization functionality, e.g.:
   a. cache hierarcy with **`n`** levels, **`s`** sets and **`w`** ways
      i. some kind of eviction policy
      ii. a limited capacity per set, per way, and global
   b. instructions are decomposed into micro-ops for execution
      i. each micro-op is eligible for a specific set of dispatch ports
      ii. ports have capacity limits
2. Incomplete: cannot be applied without profile data
   a. evaluates resource conflicts on a case-by-case basis
      i. relies on hardware counters whenever possible
   b. generates tests to confirm hypothesized conflicts
      i. reports inconclusive analysis when results are vague

Two flavors of differential profiling

1. **Chunking**
   a. detect macro conflicts: e.g.: cache, branch predictor
   b. basic idea: measure performance of sub-programs
      i. chop the trace into equal-sized trace chunks
      ii. measure cache/branch predictor miss rate per chunk
      iii. iterate each chunk having a high miss rate (record/replay)
      iv. if miss rate drops, expand the chunk until miss rate elevates
2. **Splicing**
   a. detect micro conflicts: e.g.: pipeline, hardware loop
   b. basic idea: measure performance of program variations
      i. hypothesize missed optimization opportunities (e.g., stalls)
      ii. randomly transform representative chunk until it optimizes

Chunking example

startup	init	calculation loop			output
		analyze shapes	detect overlap	trim shapes	

whole program trace

Chunking example

startup	init	calculation loop			output
		analyze shapes	detect overlap	trim shapes	

whole program trace

cache miss ratio

45	20	74	43	35	12	22	30	64	43	53	65	74	59	4	8	68	73	31	29
1	1	1	40	8M	8M	8M	8M	8M	6M	6M	6M	6M	9M	9M	9M	9M	2K	2K	50

reuse potential (average # of loads per target address)

Chunking example

startup	init	calculation loop			output
		analyze shapes	detect overlap	trim shapes	

whole program trace

cache miss ratio

45	20	74	43	35	12	22	30	64	43	53	65	74	59	4	8	68	73	31	29
1	1	1	40	8M	8M	8M	8M	8M	6M	6M	6M	6M	9M	9M	9M	9M	2K	2K	50

reuse potential (average # of loads per target address)

	2	5	3	0	1	43	40	38	54	8	0	0	0		
	4			3						7		4			
15			18							68					
24															

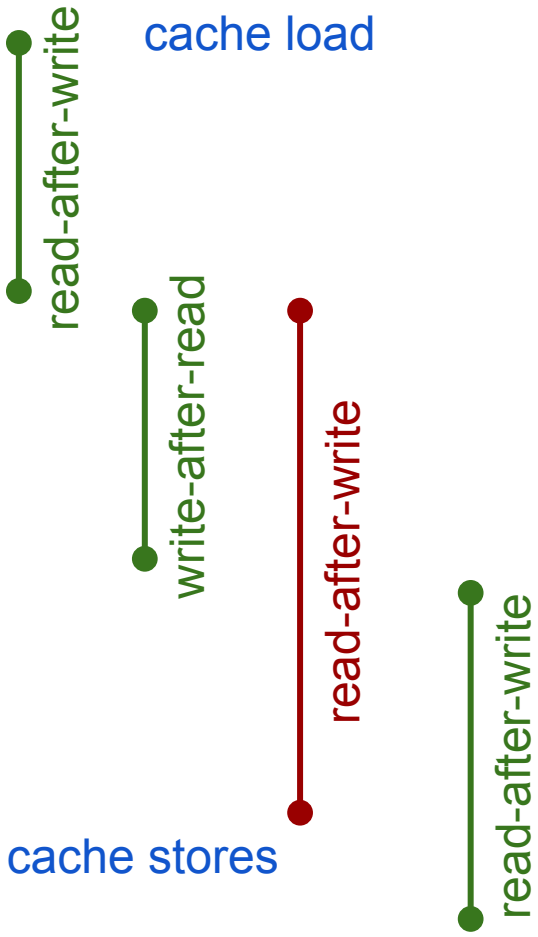
expanding chunk analysis

Two flavors of splicing

1. **By comparison:** isolate differences between executions
   a. Align traces and demarcate major performance differences
   b. Determine locality by iterating marked regions in isolation
      i. if slowdown disappears, go to macro conflict analysis
   c. Apply trivial machine model to detect possible causes
   d. Splice cause-relevant elements of the fast run into the slow
      i. Report changes having fast performance in the slow context
2. **By model:** search for typical optimization conflicts
   a. Select theoretically optimizable regions of iterative code
   b. Apply trivial machine model to hypothesize conflicts
   c. Randomly generate conflict resolutions and profile them
      i. Report resolutions that cross an optimization threshold

Splicing example

```
mov (%rbx), %rcx
... (4 instructions)
mul %rcx, %rsi
... (5 instructions)
mul %rbx, %rcx
... (3 instructions)
mov %rsi, (%r8)
mov %rcx, 0x8(%r8)
```



Case 1: some inputs run 50% slower

```
mov (%rbx), %rcx
```

Hypothesis: cache miss here...

```
... (4 instructions)
```

```
mul %rcx, %rsi
```

...causes a dependency stall...

```
... (5 instructions)
```

```
mul %rbx, %rcx
```

...increasing pressure on a busy micro-op port...

```
... (3 instructions)
```

```
mov %rsi, (%r8)
mov %rcx, 0x8(%r8)
```

...delaying loop iteration, and blocking instruction parallelism.

Case 1: some inputs run 50% slower

```
mov (%rbx), %rcx
... (4 instructions)
mul %rcx, %rsi
... (5 instructions)
mul %rbx, %rcx
... (3 instructions)
mov %rsi, (%r8)
mov %rcx, 0x8(%r8)
```

Splice test:

1. Generate a funclet to preload cache with the %rbx targets
2. Start exection in the funclet and continue into the loop
3. Measure execution time and relevant performance counters
4. If the slow inputs now run fast, the cache miss is the problem

Case 2: code change runs 50% slower

original version

```
mov (%rbx), %rcx

... (4 instructions)

mul %rcx, %rsi

... (5 instructions)

mul %rbx, %rcx

... (3 instructions)

mov %rsi, (%r8)
mov %rcx, 0x8(%r8)
```

new version

```
mov (%rbx), %rcx

... (4 instructions)

mul %rcx, %rsi

... (4 instructions)

mul %rbx, %rcx

... (3 instructions)

mov %rsi, (%r8)
mov %rcx, 0x8(%r8)
```

instruction has been removed

Case 2: code change runs 50% slower

original version

```
mov (%rbx), %rcx
```

new version

```
mov (%rbx), %rcx
```

Hypothesis: poor case for greedy parallelism (ILP)

1. micro-op port pressure delays the second mul
2. another instruction is ready before the second mul
3. CPU greedily dispatches the other instruction
4. the use of the mul result is now stalled
5. loop iterations are delayed by 3 cycles
6. hardware loop can no longer execute this code

```
mov %rcx, 0x8(%r8)
```

```
mov %rcx, 0x8(%r8)
```

Case 2: code change runs 50% slower

original version

```
mov (%rbx), %rcx
```

new version

```
mov (%rbx), %rcx
```

Splice test:

1. analyze the code for control dependencies
   - ➢ missing instruction is not control relevant:
     i. insert that instruction and verify it is retired
     ii. measure hardware loop utilization
   - ➢ missing instruction is control relevant:
     i. try similar instructions (latency, port usage, etc)
     ii. check consistency of other hardware counts
     iii. measure hardware loop utilization

```
mov %rcx, 0x8(%r8)
```

```
mov %rcx, 0x8(%r8)
```