Efficient and Flexible Value Sampling

Mike Burrows, Ulfar Erlingson,
Shun-Tak Leung, Mark Vandevoorde,
Carl Waldspurger, Kip Walker, Bill Weihl

Compaq Computer Corporation
Systems Research Center

Goal: Value profiling

Record values during program execution.

Find “semi-invariant” values for prefetching, specialization, speculation.

For example:

> A load reads from 0x3c8 95% of the time.
>
> A function is always evaluated at zero.

Possible techniques

Instrument program with a binary editor (Calder *et al.*, 97).

Interpret, and record values generated.

Sample values using periodic timer interrupts.

We explored the last approach.

It's potentially far less intrusive.

DCPI review

DCPI profiles all address spaces.

Generates periodic interrupts.

Interrupt routine records process ID and PC.

User-space daemon maps to offset/executable, and aggregates samples in files.

Tools report data for executables, procedures, and instructions.

Value sampling with DCPI

On each interrupt, collect values.

Somehow associate values with PC,PID.

Summarize the values, and aggregate in files.

Tools analyse summaries to find optimization opportunities.

Inherited properties of DCPI

It's a sampling technique.

It has modest overhead.
Low enough for production use.

It's transparent to the user.

It can be used on the whole system.

Associating values with instructions

Which instructions generated which values?

On interrupt, we don't know:

which instruction was last executed.

which register was last written.

First try—“bounce back” interrupt

Alpha 21164 can interrupt after k user-mode cycles.

On a periodic interrupt:

```
    record PC;
    set $k$ to be small;
    return.
```

On “bounce-back” interrupt, match executed instructions against register contents.

Bounce-back is tricky

Works only on user-mode on the 21164.

Hard to set k because timing is unpredictable.
e.g. Chip interrupts 6 cycles *after* event.

Occasionally confused by tight loops.

—

Evict the i-cache line to improve predictability.

Start by setting k small, and increase.

Collecting values with an interpreter

On each interrupt, interpret a few instructions.

Save values associated with each instruction.

Should interpreter have side-effects?

No side-effects $\Rightarrow$ correctness less critical.

But we want to value-profile the kernel, and loads have side-effects in device drivers.

So interpreter must affect process state.

Fortunately, testing is merely tedious.

Interpreter advantages

It's easy to associate values with instructions.

We can gather other values, e.g. load latency, PC of calling procedure.

User can configure what data to gather.

We can interpret in user mode via an up-call.

Interpreter limitations

Can’t interpret when interrupts are disabled.

Can’t interpret through an OS trap.

Can’t interpret for too long.

Hotlists: Gibbons & Matias' algorithm

One algorithm instance per PC.

Counts each value seen with probability p.

p is decreased so counts fit in constant space.

Probabilistically yields most common values & frequency estimates.

It's a great simplification over *ad hoc* schemes.

A value profile

```
cycles instruction
  39   ldq ra, -16(t12)  ra:(98.94% 0xff...ff) (0.53% ...
   0   and a1, s1, v0    v0:(4.76% 0x55...00) (3.17% ...
   0   and a1, s3, a1    a1:(100.00% 0x0)
   0   eqv v0, s2, v0    v0:(4.23% 0x55...00) (2.65% ...
   0   xor a1, s4, a1    a1:(100.00% 0x0)
9748   bic ra, v0, v0    v0:(100.00% 0x55...1c)
```

Load latencies

Measured using CPU's cycle counter.

```
cycles instruction      latencies
  0.0  ldt  $f17, 8(t6) (94.3% D) (3.6% M) (2.1% B)
  ...
  0.0  ldt  $f11, 0(t2) (84.9% M) (15.1% D) (0.0% B)
  ...
102.3  mult $f11,$f17,$f17
```

Are latency values meaningful?

Usually, yes.

We displace a few percent of d-cache lines.

—

Can’t get i-cache fill latencies with interpreter.

Nor mispredict penalties.

21264 replay traps

Reordering can violate memory semantics.
e.g. a load of L reordered before a store of L

A *replay trap* replays the offending instruction.

Expensive: all later instructions are replayed.

Hardware counters say where the trap occurred, but not why.

Identifying replay trap cause: vreplay

Interpret > 100 instructions at a time.

Interpreter compares load/store addresses.

Records which instructions could conflict.

Later, combine results and hardware counts.

vreplay output

```
replays                                 vcount
...
 0  0x...2a0  stt  $f8, 104(sp)      5 (100.0% 0x...4f8)
 0  0x...2a4  bis  a0, a0, s5        0
 0  0x...2a8  bis  a1, a1, s6        0
...
 0  0x...2c8  bis  v0, v0, s2        0
43  0x...2cc  ldq  at, 0(a0)        25 (100.0% 0x...0d0)
 0  0x...2d0  bsr  ra, 0x20027a50   0
...
```

Overhead
CPU2000 integer benchmarks

DCPI option	Average Slowdown %
no vprof	3.9
vprof	10.7
vreplay	5.9

vprof interprets 4 instructions per 124k

vreplay interprets 128 instructions per 8M

Summary

Periodic interpretation: flexible, about 10% overhead.

Provides value profiles and data not provided by hardware counters.

Gibbons and Matias' algorithm is useful.

—

Download from

`http://www.tru64unix.compaq.com/dcpi/`

Related work

Gabbay and Mendelson, 1996.

Calder *et al*, 1997, 1999.

Feller, 1998.

Deaver *et al*, 1999.

Bala *et al*, 1999.

Chambers and Ungar, 1989.