Achieving accurate & context-sensitive timing for code optimization

or

How do we measure success for tuning & performance?

R. Clint Whaley
whaley@cs.utsa.edu
www.cs.utsa.edu/~whaley

Anthony M. Castaldo
castaldo@cs.utsa.edu
I. Motivation
II. Introduction: Naïve kernel and timer implementations
III. Flushing caches when calling kernel once per sample
IV. Flushing caches when calling kernel multiple times per sample
V. Timer refinements/misc timing techniques
VI. Further Information
VII. Response to mandatory questions/red meat to dogs
I(a). Problem Motivation

Why do context-sensitive timings matter?

Problem Definition

- Literature contains much discussion of optimizations, little discussion on how to measure transformation results
- Performance of optimization usually measured by home-grown timer
- If timer does not replicate the calling context found in target application(s), timer results are often misleading
  - Most important context is probably cache state

Does Lack of Context Sensitivity Matter?

- Changes magnitude of speedup enormously (next slide)
- Changes best parameters for most optimizations
- Changes viability of many optimizations altogether
I(b). Impact of Timer Method. on Speedup
Performance of DDOT using flushed & non-flushed timers

Vector length (N) vs. MFLOP

- **no flush**
- **flushed**

R. Clint Whaley (UTSA-CS)

July 9, 2008
Demonstrated strong effect of in- vs. out-of-cache timing on all considered optimizations in:


Less formally, consider:

- Optimizations like: load/use pipelining, data prefetch, tiling
  - All may show no benefit, slowdown, get wrong param value when timed in-cache, but used out-of-cache
  - All may be critical for out-of-cache performance
- Does this actually occur (yes, next slide)?
I(d). Wrong Timer’s Impact on Autotuning
ATLAS DGEMM performance when installed with/without flushing

Matrix Order (N) vs. MFLOPS

Flush tune vs. No-flush tune

R. Clint Whaley (UTSA-CS)
II. Naïve Kernel and Timer

**DDOT kernel**

```c
double dotprod(
    const int N,
    const double *X,
    const double *Y)
{
    int i;
    double dot=0.0;
    for (i=0; i<N; i++)
        dot += X[i] * Y[i];
    return(dot);
}
```

**Naïve timer**

```c
for (i=0; i < N; i++)
    { // Init operands
        X[i] = rand();
        Y[i] = rand();
    }
    
    // Perform timing
    
    t0 = my_time();
    dot = dotprod(N, X, Y);
    t1 = my_time();
```

- Init preloads operands to any cache large enough to hold them
III. Portable Cache Flushing for One Call

Cache flushing when kernel is called only once per sample

**LRU-based cache flush**

\[
dsz = \text{sizeof(double)} \\
cs = \text{cacheKB} \times 1024 / dsz; \\
\text{flush} = \text{calloc}(cs, dsz); \\
\text{for} \ (i=0; \ i < N; \ i++) \ { \\
   \text{X}[i] = \text{rand}(); \ // \ \text{Init} \\
   \text{Y}[i] = \text{rand}(); \\
}\)

\[
\text{for} \ (i=0; \ i < cs; \ i++) \\
   \text{tmp} += \text{flush}[i]; \ // \text{flsh} \\
\text{assert} (\text{tmp} < 10.0); \\
\text{t0} = \text{my\_time}(); \\
\text{dot} = \text{dotprod}(N, \ X, \ Y); \\
\text{t1} = \text{my\_time}();
\]

**OneCallFlushLRU Notes**

⇒ Access unrelated area \( \geq \) cache size to force flush

- Relies on LRU for flush
  → For non-LRU caches, increase \text{cacheKB}

- Vary flush level \text{wt} \text{cacheKB}

- Allow specific ops in-cache by initing after flush

- Paper has x86-specific method using explicit cache-flush instructions
IV(a). Cache Flushing for Multiple Calls
Timing when each measurement contains multiple kernel invocations

When kernel call below repeatable clock resolution, can time loop that invokes kernel $nrep$ times to get timing interval above resolution:

- Cannot start & stop timers inside loop
  - each interval below resolution, so timing mostly error
  - adding them up gives erroneous time

⇒ Must start timer before $nrep$ loop, stop after:
- If you call with same operands, will be in-cache
- If you use prior technique, last $nrep - 1$ calls in-cache
- If you put flush inside loop, flush time added to kernel time
  - Cannot time flush only loop and subtract, since flush time may vary strongly depending on external access

⇒ Must lay out operands in mem, and move so that each kernel invocation uses out-of-cache data (next slide)
IV(b). What not to do
Timing when each measurement contains multiple kernel invocations

**bad idea1 - no resolution**

```c
dsz = sizeof(double);
flush = calloc(cs, dsz);
for (j=0; j < nrep; j++)
{
    for (i=0; i < cs; i++)
        tmp += flush[i];
    assert(tmp < 10.0);
    t0 = my_time();
dot = dotprod(N, X, Y);
t1 += my_time() - t0;
}
```

**bad idea2 - flush prob**

```c
t0 = my_time();
for (j=0; j < nrep; j++) {
    for (i=0; i < cs; i++)
        tmp += flush[i];
    assert(tmp < 10.0);
dot = dotprod(N, X, Y);
}
t1 = my_time() - t0;
t0 = my_time();
for (j=0; j < nrep; j++) {
    for (i=0; i < cs; i++)
        tmp += flush[i];
}
t1 -= my_time() - t0;
```
IV(c). Cache Flushing for Multiple Calls

Cache flushing when kernel called multiple times in one sample

Multiple call dot product timer

```c
\[ cs = \text{cacheKB} \times (1024/\text{sizeof(double)}); \]
\[ \text{setsz} = N + N; \quad // \text{2 N-length ops in wrk set} \]
\[ \text{nset} = (cs + \text{setsz}-1)/\text{setsz}; \]
\[ \text{if (nset < 1) nset=1}; \]
\[ \text{Nt} = \text{nset} \times \text{setsz}; \]
\[ \text{X} = \text{vp} = \text{malloc} (\text{sizeof(double)}*\text{Nt}); \]
\[ \text{X} += \text{Nt} - \text{setsz}; \text{Y} = \text{X} + \text{N}; \]
\[ \text{for} (x=\text{vp},i=\text{Nt}-1; i \geq 0; i--) \]
\[ \quad \text{x[i]} = \text{my_drand}(); \]
\[ \text{x=X}; \; \text{y=Y}; \; k=0; \; \text{alpha} = 1.0; \]
\[ \text{t0} = \text{my_time}(); \]
\[ \text{for} (i=0; i < \text{nrep}; i++) \{ \]
\[ \quad \text{dot} += \alpha \times \text{dotprod}(N, X, Y) \]
\[ \quad \text{if (++k != nset) \{x -= \text{setsz}; y -= \text{setsz};\}} \]
\[ \quad \text{else} \{x=X;y=Y;k=0;alpha = -alpha;\} \]
\[ \} \]
\[ \text{time} = (\text{my_time}()-\text{t0})/((\text{double})\text{nrep}); \]
```
V. Timer Refinements

List of additional timing techniques/tips covered in paper

Paper provides techniques for avoiding:

- Floating point over/under-flow,
- Lazy page zeroing,
- Virtual memory instruction load,
- Incorrect timings due to CPU throttling.

Paper discusses methods for:

- Choosing best system timer
- Getting more repeatable results using both CPU and WALL timers,
- Varying type and thoroughness of flush,
- Enforcing memory (mis)alignment,
- Adapting cache flushing for parallel timings.
VI. Further Information

- **Presenter homepage**: www.cs.utsa.edu/~whaley/
- **Timing paper**: Clint Whaley and Anthony M. Castaldo, “Achieving accurate and context-sensitive timing for code optimization”, accepted for publication in *Software: Practice & Experience*
  
  → www.cs.utsa.edu/~whaley/papers/timing_SPE08.pdf
- **ATLAS homepage**: math-atlas.sourceforge.net
  
  → www.cs.utsa.edu/~whaley/papers/icpp05_8.ps
In theory, no, in practice, yes.

Probably will compete on selected benchmarks, but be crushed for actual use.
Picking losing fights
Self-tuned libraries will always outperform compiler-generated code

⇒ In theory, no, in practice, yes.
→ Probably will compete on selected benchmarks, but be crushed for actual use.

Why: Three anti-HPC Compiler Traditions

1. My assumptions trump your experimental results
   - Libraries eventually have users wt. applications
     → keeps them honest to some degree

2. All problems solved 20 years ago → nothing works today
   - HPC weak, but does reward raw performance improvement
   - We haven’t solved this prob in serial:
     ⇒ Let’s solve it on heterogeneous massively parallel machine!

3. 10,000 front-ends, 0 HPC backends
   - CISC compaction, front-end (arch) optimization, inst alignment, inst selection & sched