

Achieving accurate & context-sensitive timing for code optimization

R. Clint Whaley

`whaley@cs.utsa.edu`

`www.cs.utsa.edu/~whaley`

Anthony M. Castaldo

`castaldo@cs.utsa.edu`

University of Texas at San Antonio
Department of Computer Science

April 3, 2008

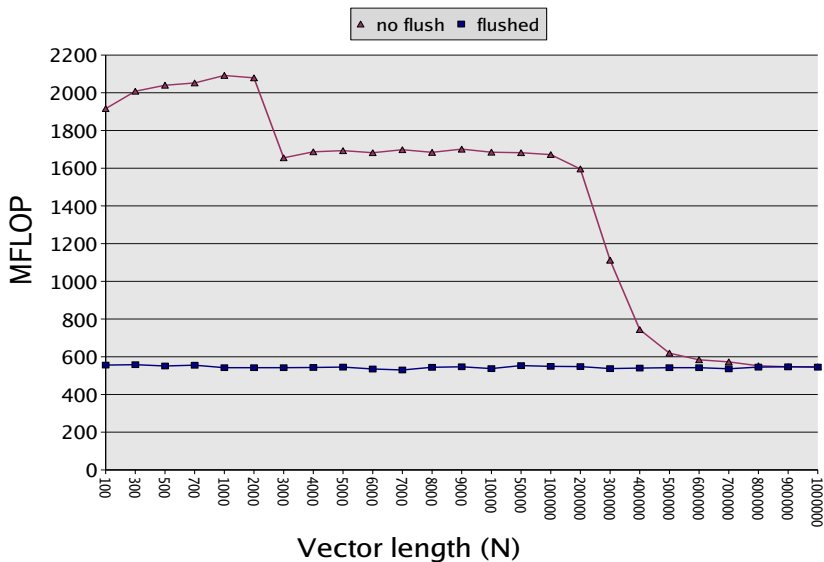
- I. Motivation
- II. Introduction: Naïve kernel and timer implementations
- III. Flushing caches when calling kernel once per sample
- IV. Flushing caches when calling kernel multiple times per sample
- V. Timer refinements/misc timing techniques
- VI. Further Information

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

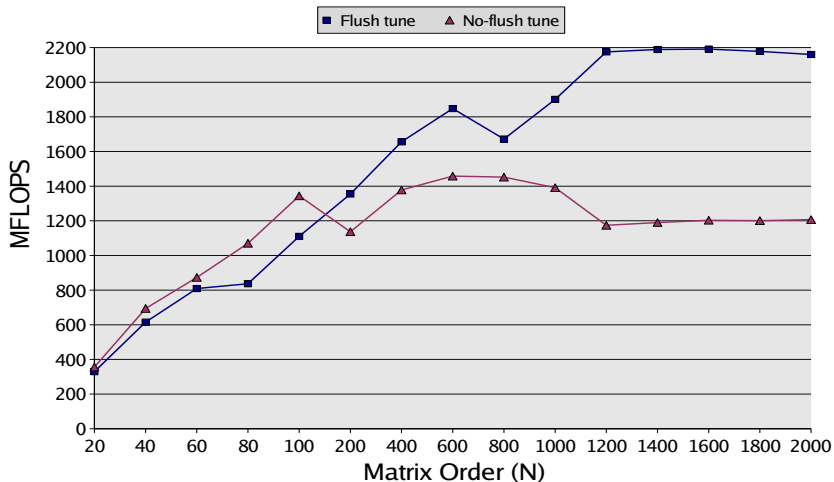


Demonstrated strong effect of in- vs. out-of-cache timing on all considered optimizations in:

- R. Clint Whaley and David B. Whalley, "Tuning High Performance Kernels through Empirical Compilation", In *The 2005 International Conference on Parallel Processing*, June 2005.

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)?



DDOT kernel

```
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

```
    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

LRU-based cache flush

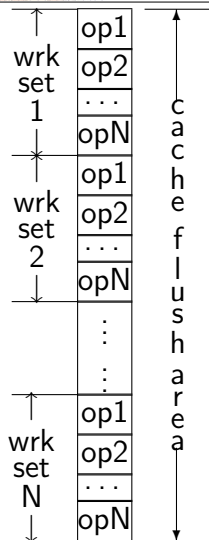
```
dsz = sizeof(double)
cs = cacheKB*1024/dsz;
flush = calloc(cs,dsz);
for (i=0; i < N; i++) {
    X[i] = rand(); // Init
    Y[i] = rand();
}
for (i=0; i < cs; i++)
    tmp += flush[i]; //flsh
assert(tmp < 10.0);
t0 = my_time();
dot = dotprod(N, X, Y);
t1 = my_time();
```

OneCallFlushLRU Notes

- ⇒ Access unrelated area \geq cache size to force flush
- Relies on LRU for flush
 - For non-LRU caches, increase cacheKB
- Vary flush level wt cacheKB
- Allow specific ops in-cache by initing after flush
- Paper has x86-specific method using explicit cache-flush instructions

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)



Multiple call dot product timer

```

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

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.

- **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
- **iFKO paper:** R. Clint Whaley and David B. Whalley, “Tuning High Performance Kernels through Empirical Compilation”, In *The 2005 International Conference on Parallel Processing*, June 2005.
→ www.cs.utsa.edu/~whaley/papers/icpp05_8.ps

Choosing a system timer

- Use CPU time if machine heavily loaded and no I/O or parallelism
 - Low resolution: both over- and under- reports time
- Otherwise, use (cycle-accurate) wall time
 - High resolution, but includes other processes

Timings have strong variance

Can improve by taking multiple timing samples and:

- return median for CPU time
 - Return minimal time for walltime
- Paper shows how to safely use cycle-accurate wall timers, and gives assembly code for x86 & SPARC

x86 GetCycleCount.S

```
.text
.global GetCycleCount
GetCycleCount:
    #ifdef __LP64__
        xorq %rax, %rax
        .byte 0x0f
        .byte 0x31
        shlq $32, %rdx
        orq  %rdx, %rax
    #else
        .byte 0x0f
        .byte 0x31
    #endif
    ret
```

Avoiding roundoff for floating pt

```
long long GetCycleCount();
double GetWallTime()
{
    static long long start=0;
    static const double SPC = 1.0/(Arc
    long long t0;

    if (start) {
        t0 = GetCycleCount() - start;
        return(SPC * t0);
    }
    start = GetCycleCount();
    return(0.0);
}
```

x86-specific cache flush

```
#define flCacheLn(mem) \  
    __asm__ __volatile__ \  
    ("clflush %0" :: "m" \  
     (*((char *) (mem))))  
for (i=0; i < N; i++) {  
    X[i] = rand(); // Init  
    Y[i] = rand();  
}  
for (i=0; i < N; i++){  
    flCacheLn(X+i); // flsh  
    flCacheLn(Y+i);  
}  
t0 = my_time();
```