Achieving accurate and context-sensitive timing for code optimization *

R. Clint Whaley and Anthony M. Castaldo [†]

January 18, 2008

Abstract

Key computational kernels must run near their peak efficiency for most high performance computing (HPC) applications. Getting this level of efficiency has always required extensive tuning of the kernel on a particular platform of interest. The success or failure of an optimization is usually measured by invoking a timer. Understanding how to build reliable and context-sensitive timers is one of the most neglected areas in HPC, and this results in a host of HPC software that looks good when reported in papers, but which delivers only a fraction of the reported performance when used by actual HPC applications. In this paper we motivate the importance of timer design, and then discuss the techniques and methodologies we have developed in order to accurately time HPC kernel routines for our well-known empirical tuning framework, ATLAS.

Contents

List of Figures

1   Introduction

In high performance computing (HPC), there are many applications for which no amount of compute power is "enough". A key example of this is in scientific simulation, where an increase in compute speed will allow the scientist to increase the accuracy of the model, rather than solving the same problem in less time. Many HPC applications share this characteristic: the time that the application runs is always as long as the scientist can afford, and so extra speed translates to more detailed or accurate problem solving.

Such applications are written so that their computational needs can be serviced to the greatest possible degree by building-block computational libraries. This reduces ongoing tuning task is to tuning a modest number of widely used operations, rather than tuning each HPC application individually. Some of the operations of interest include matrix multiply (main kernel for dense linear algebra), sparse matrix-vector multiply (sparse linear algebra), and fast Fourier transforms (FFTs).

Therefore, there are a variety of commercial and academic libraries dedicated to optimizing these types of kernels, and the literature contains many publications which discuss them. More particularly, there are a host of papers each year devoted to code transformations for performance optimization, both in compiler and HPC journals. As far as we can determine, however, there is little or no discussion (beyond a few scanty details available in textbooks) of how to measure the performance improvements that have actually been achieved.

This is unfortunate, and represents an ongoing problem for both researchers and users of HPC libraries. The reason is that most timers used by researchers are extremely naïve, and fail to take account of important information such as cache state, so that often the "tuned" code produced using a naïve timer for tuning is no faster than untuned code when used in the application, despite the timer having shown a large speedup.

This problem has recently become acute, with the rise of packages that adapt critical performance kernels to given architectures using an automated timing/tuning step. Such efforts include domain-specific optimizers, which include PHiPAC [1], FFTW [2, 4, 3], ATLAS [13, 14, 15, 17, 16], SPIRAL [9, 7], and OSKI [12], as well as research on iterative compilation [6, 8, 11, 10, 18]. This type of research is all typified by making code transformation decisions based on timings taken on a platform of interest. Since optimization choices are based on these timings, it becomes critical that the timers are calling the computational kernels in the way in which the applications call them. In many cases, unfortunately, this is clearly not the case, and so it is not uncommon to see performance numbers in publications that are never realized by any actual application.

1.1   Understanding the scope of the problem

Figure 1 shows the performance of ATLAS's dot product kernel on a 2.4Ghz Core2Duo processor, timed using two different methods. The blue line with squares shows the results reported by a naïve timer (such a timer can be seen in Figure 3(b)) which preloads the cache, whereas the red line with triangles shows the results as reported by a timer which ensures that the operands are not cache-resident. At the beginning of the curve, the unflushed version runs more than 3.5 times faster than the flushed version. The unflushed timings within this region time performance when running the kernel with all the operands preloaded to the L1 data cache. After $N = 2048$ the operands will no longer fit in the 32Kb L1 data cache, and so $N = 2048$ starts a new plateau, with the unflushed version running roughly three times faster than the flushed version. This second plateau corresponds to running the kernel with all operands preloaded to the L2 cache. However, once N reaches roughly $200,000$, the operands begin to exceed the L2 cache size and so the unflushed timings exhibit a precipitous drop-off in performance, until at the end of the timing range the flushed and unflushed timers produce nearly identical results. We stress we are timing exactly the same kernel throughout, so this graph demonstrates the magnitude of the problem on even a simple kernel like dot-product: If there is no good reason to assume operands will be in L1, a timer that preloads the operands to L1 (or any level of cache) can report extremely misleading timings.

These results are fairly typical when contrasting flushed and unflushed timers: the unflushed timers show large performance losses on cache boundaries, which results in small problems achieving greater performance than large problems. Flushed results typically show a pattern of steady performance improvement until an asymptotic speed is reached. This reflects what we *expect* to see; that larger problems tend to get better performance, as transformations with overheads (eg., unrolling) amortize their startup costs more completely, and large-

Figure 1. Dot product results from cache flushed and unflushed timers

scale optimizations such as blocking kick in. In the case of dot product we see a flat flushed curve, because dot product, which is a simple operation with only a few key optimizations, has essentially already fully amortized its optimizations at our starting problem size.

Our dot product kernel is not particularly well optimized for this platform, and the unflushed/flushed ratio can grow even more extreme for other kernels/implementations/ architectures. Therefore, we see that the results reported can be significantly different
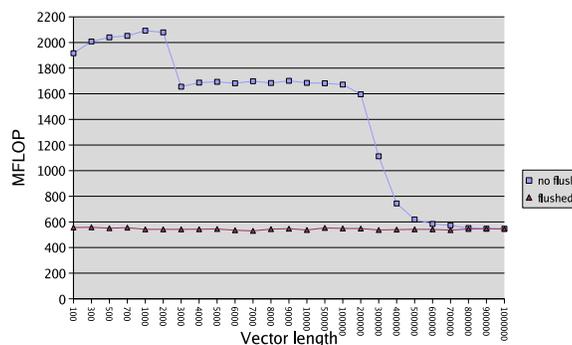
depending on how the timer is implemented. This leads to the question of whether tuning using the wrong timing methodology will produce a differently optimized kernel, or if they will in fact be the same.
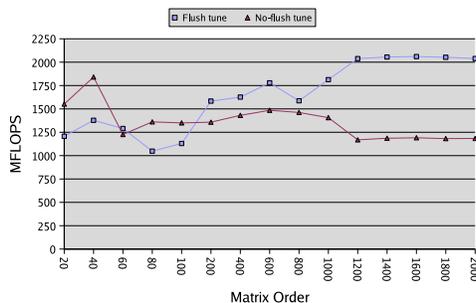
The answer is that indeed, the timing methodology has a strong effect on what the best optimization parameters are. In [18] we showed that having the operands in- or out- of cache strongly changed both the type and degree of beneficial transformations for even the simplest of kernels. Less formally, consider two simple optimizations: data prefetch and load/use software pipelining. When data is in the cache, neither one of these optimizations may give any advantage at all, and indeed due to overheads, might cause a slowdown. If the data is out-of-cache, and the operation is bus-bound, then these may be the most important optimizations that can be applied. More generally, in-cache timings will stress the importance of computational optimizations, and out-of-cache timings will stress the importance of memory optimizations. Unfortunately, memory is several orders of magnitude slower than modern processors, and so a kernel that is completely optimal computationally may run no faster than an unoptimized kernel when called with operands that are not preloaded to the cache. Therefore, we see that if we empirically tune the kernel using the in-cache numbers, we will be able to report massive speedups in a paper, but any user calling our kernel with out-of-cache data may experience no speedup at all over unoptimized code. Even worse, by tuning for the in-cache case, we may have produced a kernel that is far less efficient than we could have produced using our current tuning framework, merely because our naïve timer showed that crucial memory optimizations gave no benefit.

To put some teeth behind the idea that timing with the wrong context can cause an automatic tuning framework to underperform its potential, we installed ATLAS's double precision matrix multiply (DGEMM) twice on a 1.35Ghz UltraSPARC IV. For both installs, we refused architectural defaults (which allow ATLAS to skip parts of the automatic tuning by using previously saved values), and ran a complete automatic tuning of DGEMM from the ground up. In the first install, ATLAS was allowed to flush the cache as usual in all timings, and in the second install, we turned off cache flushing completely. Figure 2 shows the performance of the resulting automatically-tuned kernel. In figure 2(a), we measure the performance using a timer which flushes the cache, and in Figure 2(b) we measure the same two kernels using a timer which does no cache flushing. These figures have similar asymptotic peaks, since large enough operands overflow the cache, but as we have seen before, the timer without flushing reports inflated numbers when the operands are cache-contained.

We see that with the cache flush timer (Figure 2(a)), the performance curve of the DGEMM tuned with cache flushing on (blue squares) behaves as expected: a relatively smooth rise in performance until an asymptotic peak is reached (the performance drop at N=800 results from a poor matrix partitioning brought on by our L2 cache blocking). The install without flushing (red triangles) does not. The primary difference in these two differently-tuned kernels is that, without flushing, the ATLAS framework picks bad blocking
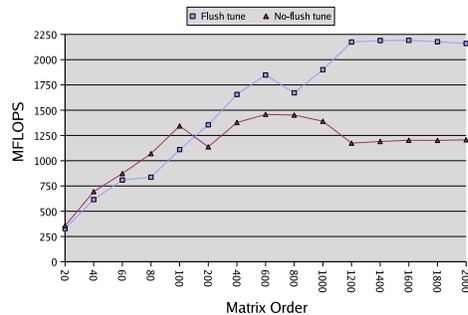
(a) Using a timer with flushing   **(b) Using a timer without flushing**

Figure 2: Performance of resulting ATLAS DGEMM on an UltraSPARC IV when the automatic tuning step uses cache flushing (squares), and when cache flushing is turned off (triangles)

factors for the L1 and L2 caches, and thus as the cache is exceeded performance drops off. This results in the DGEMM tuned without flushing having its asymptotic peak reduced by almost half when compared with DGEMM tuned with cache flushing.

The timings are much more volatile when measured with a timer without flushing, as in Figure 2(b). These early drops in performance represent the kicking in of new optimizations that have yet to amortize their cost (eg., the performance drop between N=40 and N=60 for the no-flush-tuned DGEMM is due to the matrix size exceeding the L1 blocking factor for the first time). These optimizations cause a performance loss on in-cache data (where they did not when our timer flushed the cache), since blocking, for instance, does not improve in-cache performance. This provides a further demonstration of the problem of timing things in-cache: it shows blocking, which is critical for out-of-cache performance, to be a performance *loss* unless you are lucky enough to time a problem large enough to do the appropriate amount of self-flushing.

One interesting thing to note is that the DGEMM tuned using timers without flushing actually wins for small problems. This is because smaller L1 blocking factors provide better performance for smaller problems, and tuning in-cache causes ATLAS to choose a blocking factor that is ill-tuned for large problems (where blocking is truly important). This is why the asymptotic performance of the DGEMM tuned without flushing is only a little more than half of the DGEMM tuned with flushing. Note that ATLAS's main DGEMM algorithm is presently designed so that we must use only one L1 blocking factor regardless of problem size, and so the install chooses to optimize for asymptotic performance at a slight cost to small problems. We plan to extend our DGEMM to handle multiple blocking factors, and with this in place, the cache-flushed DGEMM will win across the entire range, since the smaller blocking factor shows up as an advantage for small problems regardless of

the timing methodology used during tuning. Further, DGEMM is a kernel that reuses the operands in the cache if they fit; in other kernels that don't have this native reuse, we would find that for out-of-cache operands (the usual case), the cache-flush-tuned kernel would win throughout the entire range.

The performance loss due to failure to time appropriately when tuning will vary by architecture, kernel, and tuning framework. The present case is not very extreme, in that the two installs differ mainly on what blocking factors are used. The ATLAS framework always blocks; another framework might apply blocking only when it shows a win. In-cache timings might not show any win at all from blocking, which would result in a calamitous asymptotic performance loss. Since some computational optimizations also have memory optimization impact (eg., instruction scheduling), it is possible for the differing installs to choose completely different computational kernels, which could further impact true performance (as a matter of fact, software prefetch, which this kernel has, is usually a performance loss on in-cache data but performance critical on out-of-cache data).

Given these results, it should be clear that timer design has a profound effect on the ultimate performance of the kernel being tuned. Thus it is critical that automated tuning frameworks, which cannot rely on human judgment to bridge the gap between timer and application usage, pay particular attention to proper timer design.

1.2   Addressing the problem

We have seen that the timing methodology used can have a strong effect on the best way to optimize a kernel, particularly in regards to handling the cache. Since differing methodologies can lead to widely varying results, understanding whether users typically call the kernel with in- or out-of-cache data becomes overwhelmingly important. In practice, it need not be all-or-nothing: a kernel may be typically called with one operand in the L2 cache and another in main memory, etc. What is important is to be sure to tune the library to the important case, and this demands that timers need to be sophisticated enough to recreate all important calling contexts. Having flexible timers can have other benefits as well. For instance, we often measure many computational transforms using in-cache timings, and then begin memory transform tuning starting from this computationally optimized code using out-of-cache timings.

Therefore, in this paper we describe the timing methodologies we developed in order to support ATLAS's empirical tuning. Our approach is to choose a default timing method that we believe represents the majority of our users (out-of-cache timings with paged-in kernel code), but build our timers flexibly enough that a user with a different context could tune them for that as well. Therefore, this paper should serve as a tutorial on how to build a timer that is accurate and adaptable enough to be used to make optimization decisions in the real world.

Concrete examples often provide the best mechanism for understanding such applied

```
double dotprod(                              for (i=0; i < N; i++)
   const int N,                              {                // Init operands
   const double *X,                             X[i] = rand();
   const double *Y)                             Y[i] = rand();
{                                             }
   int i;                            //
   double dot=0.0;                   // Perform timing
   for (i=0; i<N; i++)               //
      dot += X[i] * Y[i]                t0 = my_time();
   return(dot);                         dot = dotprod(N, X, Y);
}                                       t1 = my_time();
```

(a) Simple dot product   **(b) Naïve timer**

Figure 3: Naïve kernel and timer implementations

concepts, and so we will show actual code that utilizes these methods to time a simple dot product timer. In order to get started, Figure 3(a) shows the simple dot product kernel, while Figure 3(b) shows the type of naïve dot product timer that a typical programmer might write. In examining this timer, notice that we initialize the operands, which will bring them into any cache large enough to hold them. We then immediately start the timing, which will therefore be with in-cache data (assuming the vectors fit in some level of cache). In §3 and §4 we will discuss how to control what operands are allowed to be in various levels of the cache. The second thing to notice about this naïve timer is that the kernel is called only once, which may may prevent accurate timing. In particular, if the kernel does not take long to run, the elapsed time may be below timer resolution, so that our timing result will vary widely and bear little or no relation to the actual speed of the kernel. In §2 we will discuss the various timers with an eye towards allowing the use of the highest resolution timer available, as well as presenting code that allows for cycle-accurate wall times on x86 and SPARC machines. Further, §4 will discuss how to call kernels multiple times while maintaining the desired cache flushing. Finally, in §5 we will cover a variety of small pitfalls that bedevil real-world timings and the workarounds that we use.

2   CPU VS. Wall Time

In building a high-quality kernel timer, the first thing that must be considered is what system timer to use. Timers are divided into two categories depending on whether they measure *CPU time* or *wall time* [5].

   Wall time is the most straightforward measure. Wall timers attempt to measure the actual elapsed time between two calls to the timer, and get their name because they should deliver the same value as would be obtained if a clock on the wall was consulted when the

timed operations were begun, and this value was subtracted from the time showing when the operations complete. This time is straightforward because it requires only that the processor have some mechanism for measuring the passage of time reliably. Wall time is usually very accurate, with resolution that can go as high as the clock rate of the processor (eg., many 2Ghz processors can return wall times that are accurate to within half of a nanosecond). The problem with wall times is that they include total elapsed time, including time spent doing other users' tasks, time spent doing unrelated OS operations, etc. This means wall time provides a very accurate measure of elapsed time, but that the timer will be unsure how much of that time came from the kernel whose performance is being measured.

This is the reason for the existence of *CPU time*. In CPU time, the OS attempts to quantify how much work the <u>CPU</u> undertook for a particular process. Therefore CPU time does not include time when other (non-child) processes are running, or when performing I/O. In order to measure CPU time, the OS assigns given time slices to the CPU time of the appropriate process (or to no process, as in the case unrelated OS routines). The time slices themselves must be measured in terms of wall time, of course, and so we see that CPU time can never have greater resolution than the system's most accurate wall timer. In fact, due to overhead concerns, the difficulty of assigning all time to the appropriate task/user, and because it is impossible to completely sort out all unrelated costs, all CPU timers have much lower resolution than wall timers. Therefore, we see that CPU time is less affected by machine load, but has much poorer resolution.

So, the first question that must be answered is what type of timer to use. If the kernel being tuned includes I/O costs or involves parallelism, then wall time must be used (parallel codes measured with CPU time will nearly always show perfectly parallel speedup, since the time a process spends waiting on another process will not be included). If optimizing a serial compute-bound kernel, then either class of timer can be used. If the timings are running on a relatively unloaded machine, then wall time will usually provide more accurate timings. However, if the load of the machine is unknown (as is the case in a package such as ATLAS, where a user can choose to perform the installation under unknown conditions), then CPU time may be the only way to get even vaguely repeatable timings. Therefore, in ATLAS we default to using CPU time for all non-threaded timings, on the assumption that the installation machine could be experiencing heavy load. However, we encourage users to throw a flag indicating that the more accurate wall timer should be used if they are able to get access to a relatively unloaded machine. Regardless of the type of timer chosen, the reliability of the timing can be improved by timing the same problem multiple times, and reporting the most appropriate time, as outlined in §2.1.

2.1 Handling multiple timing samples

The accuracy of both CPU and wall times can be improved by using multiple samples (i.e., time the problem n times, and then report the time in an appropriate way, as discussed

below). In ATLAS, we typically use a modest number of samples, say three at minimum and seven at most. When using wall time, and particularly cycle-accurate wall time, then the only real source of inaccuracy is that other processes' are included in the timing. Therefore, for wall time the most accurate timing in a series will be the minimum timing.

CPU time, on the other hand, is highly inaccurate in both directions: it can be very much larger than it should be, and very much smaller. By experience we can verify that this innaccuracy can be substantial in both directions, and so we see that returning the maximum or minimum timing will not be helpful. Originally, ATLAS used the average of all samples, until an ATLAS user (Carl Staelin) pointed on the ATLAS mailing list that averaging timings that include such widely varying data is extremely unlikely to lead to reliable timings. Therefore, for CPU time, ATLAS returns the median value (middle of sorted samples), which results in throwing out both the over and under estimation outliers that CPU time is prone to.

2.2   Avoiding round-off using cycle-accurate timers

Most cycle-accurate timers return their values as a 64-bit integer. Even the less accurate timers often return their results in a way that more than 32 bits of integer accuracy are returned. We usually find that it is more convenient to return timing results as a floating point number, so that all timer methodologies can return a time measured in seconds (which must obviously be fractional), regardless of underlying accuracy. During this conversion, care must be taken so that the integral results are not truncated to fit into floating point storage, as this would have the effect of losing the most rapidly changing digits in the integer, and thus drastically lowering the resolution of the timer. It is obvious therefore that returning a 32-bit `float` is not wise. Even a 64-bit `double` can store only 53 bits of mantissa, and so we must guard against roundoff here as well. Figure 4 shows one way of doing this.

Figure 4(a) shows the assembly code necessary to get cycle accurate timings on both x86 and SPARC architectures, where CPP macros (`x86Mhz` and `SparcMhz`, defined to the megahertz of the x86 or SPARC machine, respectively) select which code to assemble. The `__LP64__` macro is automatically set by the gnu C preprocessor when compiling for 64-bit assembly, where the reading of the time stamp counter instruction must be handled differently than when in 32-bit mode. Figure 4(b) shows the wall timer that returns this value as a `double`, written to minimize timer overhead and guard against unnecessary roundoff when converting from 64-bit integer to `double`. First, we see that that we calculate the seconds-per-clock as a `static const double` so that the compiler can replace this value with a compile-time constant if possible.

In order to avoid int/real conversion rounding, the first time the routine is called we set the `long long start` variable to whatever the current cycle count is, and return a timer start time of `0.0`. For all other calls, we subtract `start` from the current cycle count, which

```
#ifdef x86Mhz
   .text                                #ifdef x86Mhz
.global GetCycleCount                      #define ArchMhz x86Mhz
GetCycleCount:                          #elif defined(SparcMhz)
   #ifdef __LP64__                          #define ArchMhz SparcMhz
      xorq %rax, %rax                   #endif
      .byte 0x0f; .byte 0x31           long long GetCycleCount();
      shlq  $32, %rdx
      orq   %rdx, %rax                 double GetWallTime()
   #else                               {
      .byte 0x0f; .byte 0x31             static long long start=0;
   #endif                                static const double SPC = 1.0/(ArchMhz*1.0E6);
    ret                                  long long t0;
#elif defined(SparcMhz)
   .section       ".text"                if (start)
   .global GetCycleCount                 {
GetCycleCount:                              t0 = GetCycleCount() - start;
   rd      %tick, %o0                      return(SPC * t0);
   clruw   %o0, %o1                      }
   retl                                  start = GetCycleCount();
   srlx    %o0, 32, %o0                  return(0.0);
#endif                                  }
```

(a) x86/SPARC `GetCycleCount.S`   **(b) Avoiding roundoff for floating point**

Figure 4: Avoiding round-off using cycle-accurate timers

should have the effect of reducing the number of binary digits in the cycle count so that it can be stored in a `double` without any rounding.

3   Cache Flushing Methods When Timing One Invocation

When using a cycle accurate timer, or when timing a long-running kernel, one invocation of the kernel may possess enough granularity that it can be reliably repeated. In such a case, we are aware of two basic methods to flush the cache.

Figure 5(a) shows the most general technique, which should work on almost all systems. In this approach we simply access a series of contiguous memory addresses that are not used by our kernel, and if this flush space is large enough, conflict and capacity misses should evict all of the previously-initialized operands from the cache. To do this, we allocate a workspace of size `cacheKB` kilobytes in addition to the space for the operands (vectors X and Y in this case). We then initialize the operands, which in this timer is done using a random number generator. In writing X and Y we will have brought them into the cache, and they

```
                                     #define flCacheLn(mem)__asm__ __volatile__\
cs = cacheKB*1024/sizeof(double);       ("clflush %0" :: "m" (*((char *)(mem))))
flush = calloc(cs,sizeof(double));      for (i=0; i < N; i++) {
for (i=0; i < N; i++) {                     X[i] = rand();   // Init operands
    X[i] = rand();   // Init operands      Y[i] = rand();
    Y[i] = rand();                      }
}                                       for (i=0; i < N; i++){// flush cache
for (i=0; i < cs; i++) // flush cache        flCacheLn(X+i);
    tmp += flush[i];                         flCacheLn(Y+i);
assert(tmp < 10.0);                     }
t0 = my_time();                         t0 = my_time();
dot = dotprod(N, X, Y);                 dot = dotprod(N, X, Y);
t1 = my_time();                         t1 = my_time();
```

(a) Generic LRU-based cache flush   **(b) x86-specific cache flush**

Figure 5: Flushing cache when calling the kernel only once

will remain in any cache of at least size $2 * N * 8$ bytes. Therefore, we now read our entire flush area by summing all its double precision elements. Since our call to calloc has filled the flush area with zeros, the sum will also be zero. We assert that the summation is small, so that the compiler won't figure out that this whole flush procedure is dead code (tmp value never used) and remove it. A similar guard to avoid losing code of this type is to print the result (code producing output is never dead code). Note that a particularly sophisticated compiler which understands the semantics of calloc could replace this loop with an assignment of tmp = 0.0, and then eliminate the whole loop, but we have never found one that does so. If this kind of things occurs, the solution is often to put the initialization of the flush area into a separate file and disallow interprocedural analysis, so that the compiler can no longer figure out that the summation must produce zero.

The only thing we must determine to apply this general cache flushing is how big a space to allocate as flush, or in other words, what we should set cacheKB to. Since we access cacheKB KB of data, we can simply set this to the size of the largest cache that we want to flush, assuming least-recently used (LRU) replacement. Unfortunately, most associative caches do not use LRU replacement due to its expense. Instead, it is common to use random or pseudo-random replacement. In this case, there is no size of the cacheKB that is guaranteed to force a complete eviction of the operands, since the most-recently used way could theoretically be selected at each random replacement step. However, as cacheKB is enlarged, less and less elements of X and Y will be retained in the cache. If a few elements are still present, the timing should not be strongly affected, so our main goal is get a large proportion of the operand elements evicted. For relatively low associativity (say 4-way or less), a rule of thumb for adequate flushing might be to set the cache flush area to roughly

`sizeof(cache)*associativity(cache)`, but as the associativity is increased, it becomes increasingly difficult to fully flush a cache with random replacement policy. In practice we simply continue increasing `cacheKB` until performance stops dropping, suggesting the cache is fully flushed.

In Figure 5(b), we see code that can accomplish the cache flush without needing extra workspace. However, this code relies on the x86-specific assembly instruction `clflush`, which flushes the cacheline holding a particular address from all caches. In this example, we use `gcc`'s inline assembly support to invoke `clflush`. Note that `clflush` flushes an entire cache line from the cache. If you know the size of your smallest cache line you can reduce the number of `clflush` calls you must make, but for simplicity we flush every element striding by 8 bytes, which is certainly a minimal cache line size in modern architectures. Finally, notice that as long as the `clflush` instruction is correctly implemented, this method can ensure full operand eviction, unlike the first method when used with random replacement caches. Our own timings show that the two methods produce equivalent results on both AMD Athlon-64 and Intel Core2Duo.

3.1 Modifications for partial cache flush

We have shown a very simplified timer, but in practice we would like the timer to have a great deal of flexibility so that we can tune the kernel for a given usage pattern. For instance, if we typically call the kernel with operands in the L2 cache but not in the L1, we can simulate this using the Figure 5(a) approach by setting `cacheKB` based on L1 size, rather than L2. The x86-specific cache flush always flushes all caches, and so does not have this flexibility.

It is probably uncommon to have both operands in any level of the cache, but a fair number of applications might have some operands in cache. For instance, consider a series of filters (each of which has its uncached data) being applied to a single vector. In such a scenario, the output vector might well stay in the cache, while the input vectors would probably need to be loaded from a slower layer of the memory hierarchy. Both of these methods can be easily adapted to cover such situations. For example, if we wished X to be cache contained, but Y not, then in the Figure 5(a) we would simply initialize X <u>after</u> the flush loop, while initializing Y before it. In the Figure 5(b), we would simply flush only Y's addresses.
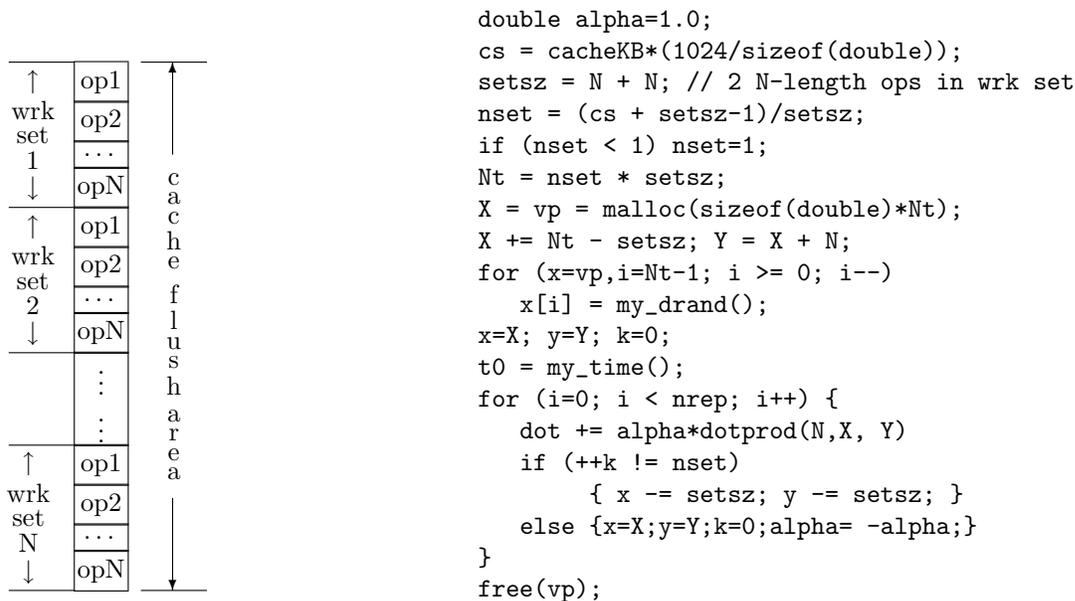
4 Cache Flushing Methods When Timing Multiple Invocations

We have seen how to handle cache flushing if a kernel need only be called once per timing interval. However, if a cycle-accurate timer is not available, then most kernels will need to

be called multiple times in order to get the timing above clock resolution. This means that instead of a single call to the kernel, we have a loop that calls it some number of times; call this the repeat loop. In this case, neither of the methods shown in Figure 5 will work reliably. The straightforward adaptation of these methods is to do the cache flush within the repeat loop, but then of course our timing includes the flush costs. The obvious answer is then to have a second loop which does only cache flush, and subtract the two timings to get our kernel timing. Unfortunately, this obvious extension does not work reliably. To see how this could be the case, imagine that the cache flush area alone fits into some level of the memory hierarchy, but that the cache flush area plus the operands of the kernel together do not. Then, the second loop over cache flush will considerably underreport how long the cache flushing operations took from within the loop.

Another common pitfall when trying to do this is to have a repeat loop, but perform all timings within the loop, so that on each iteration you start the timer, call the kernel, stop the timer, do flush, and repeat. The problem with this approach is that the main reason for doing multiple kernel invocations is to get the timing interval comfortably above clock resolution. In timing each loop invocation separately, this advantage is lost: each call is still below resolution, so that each timing result is mostly junk. When these results are added up and averaged it may yield a more stable result than when one call is made, but there is little reason to believe it actually represents how long the kernel takes to run, particularly for measurements done with CPU time, which has a variable error built into each call. For cycle accurate walltime, it may be possible for this method to work, assuming the number of timings performed is very large (say at least in the hundreds). In this case, since the timing measurement has no error within its resolution, the only problem is that the operation being timed is not an even multiple of clock ticks, and there are not enough clock ticks to make the remainder negligible. In this case, performing hundreds of timings can lead to a statistically valid result.

To see how this could be the case, imagine timing an operation which takes 2.75 clock ticks one hundred times. If each timing begins at a random location in the clock cycle (for some systems and codes, this might not be true, but assume it is for now), we would expect to get 75 results of 3, and 25 results of 2, which when averaged would provide the correct answer of 2.75 clock ticks. This method is prone to error if the timings are not truly random; for example if the kernel timing always begins a fixed amount of time after the last clock tick, then exactly the same tick count will be measured for each timing. A modern multitasking OS typically handles thousands of interrupts per second and services hundreds of tasks, and this noise may well swamp the small timing that you are attempting to measure. CPU time, which is supposed to exclude other tasks, is not a solution: We have observed that CPU time at small scales can be vary unpredictably, particularly when the kernel execution time approaches the clock resolution. Thus we consider this approach to be useful only in well-controlled environments (such as embedded systems, or OS privileged level code), and will not discuss it further here.

```
double alpha=1.0;
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;
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;}
}
free(vp);
```

(a) Cache flushing area diagram   (b) Mult. call dot product timer

Figure 6: Cache flushing for multiple kernel invocations

Therefore, our approach to timing a problem that is below timer resolution is to make the multiple kernel invocations (where the number of kernel invocations is selected so that the interval being timed is comfortably above clock resolution) that are self flushing. To do this, an area of memory that is large enough to overflow the chosen cache level to the appropriate degree (as discussed in §3) is allocated. This memory is subdivided into working sets required by the kernel, as shown in Figure 6(a), where a working set is all the operands required by the kernel (in dot product the working set would be the two input vectors X and Y, plus the scalar dot, which would have no effect on performance and is therefore ignored). After a given kernel invocation, we simply move to a new working set before making the second call, and since this cache flush area is large enough to flush the desired level of the cache, by the time we must reuse a working set, it has been evicted from the cache due to conflict and capacity misses caused the accessing the other working sets.

In practice we refine this simple concept somewhat in order to minimize the effects of hardware prefetch. We initialize the entire cache flush area in reverse order, and then start timing with work set N. We know that our dot product kernel accesses the memory in least-to-greatest address order, and so we move amongst working sets in the opposite direction. Since the operation being timed accesses the working set in a loop, all but the shortest vector lengths will allow any hardware prefetch units to detect the least-to-greatest access

pattern, and so any hardware prefetch will fetch the beginning portion of the working set that we used last time through the repetition loop, rather than what we will use on the next iteration.

Given enough working sets, it might be possible to traverse the working sets in a random walk to more fully guard against smart prefetch units figuring out the pattern. In practice, we typically don't need to allocate that many working sets, and the algorithm given above seems to produce the desired result on the machines that ATLAS has used over the years.

We show this 'backwards-traversal-of-sets' timer for dot product in Figure 6(b). As before, we see that we allocate the flush area, and we make sure that its length is a multiple of our working set size (the working set for dot product is two N-length vectors, which we store as `setsz`). We then initialize the data in reverse order (greatest-to-least), and since this area's size has been chosen to overflow the cache, working set N's space should be evicted by the time we have initialized work set 1. We then set our pointers to point to the appropriate areas in working set N, and begin the repetition loop. It is important to keep all the non-kernel code in this loop as efficient as possible, since the time for these additional instructions is being added to the time that is reported for kernel calls. Therefore, we we have only one if, which is computationally efficient and optimized for the frequent case so that normally we simply decrement the vector pointers by the set size, and then reset them when we have used all working sets. Note that `alpha` is used to guard against overflow, which if it occurred, could completely invalidate the timings. This is discussed in more detail in §5.

4.1 Modifications for partial cache flush

Since this self-flushing timer is based on the same cache concepts as the general single-invocation timer, it also can be used to flush only certain caches by the appropriate setting of the `cacheKB` variable. At the cost of a slight complication, this timing methodology can also allow for the flushing of only certain of the operands in the working set.

The easiest way to do this is to allocate each operand for which we wish to be able to change the flush characteristics in its own cache flush area. We can then use a variable (compile- or run-time) to control whether we move through a particular cache flush area, or leave the pointer unchanged across iterations of the repetition loop. This may complicate our repetition loop slightly, as the operand pointers must be updated individually. When we split the cache flush area so that each operand has its own, we may be able to reduce the individual size due to the access pattern within the loop. If this is the case, initializing each operand in turn may no longer completely flush the cache, so that we must either initialize all the used flush areas in the appropriate order (i.e. in the order they would be used in the repetition loop), or additionally use one of the methods shown in Figure 5 to force the initial flush.

5 Timer Refinements

In this section we briefly mention a few refinements that the programmer should be aware of when writing a high quality timer. This includes guarding against overflow and underflow in floating point data, and avoiding unpredictable system-level events such as lazy page zeroing and instruction loading (all of these terms are described below). Further, §5.1 discusses how timers can guarantee a particular memory alignment for operands, and why this can be important. Finally, §5.2 discusses some of the pitfalls and techniques for performing timings on shared memory parallel machines.

Guarding against over/underflow: Overflow (underflow) occurs when a number grows too large (small) to be stored using the restricted number of exponent bits available in floating point storage. Many machines handle overflow and underflow arithmetic (even denormalized or partial underflow) in software, rather than in hardware. This fact means that arithmetic experiencing overflow or underflow will possibly run hundreds or thousands of times slower than normal computation. Therefore, if the timer is calling a kernel multiple times in order to get the timing interval above a certain resolution, it is important to guard against creating conditions that can lead to overflow or underflow. In any timer that repeatedly adds into the same output data, it is possible that a buildup of magnitude could cause overflow, particularly if the results are all the same sign. Similarly, any time the same result is reused across many multiplications, there is a risk of overflow (for numbers > 0) or underflow (for numbers < 0). It is possible to take several actions to guard against this, including varying the input in known ways (eg. `input1` produces the negation of `input0`), changing the accumulating operation slightly (eg. on one call, add results in, and on the second subtract results), or always using separate output locations for each call (this method is often ruled out by storage costs when the output is a large vector or matrix). Figure 6(b) shows an example of changing the accumulation to guard against overflow. We are performing a series of dot products using a fixed number of invariant input vectors, and if the number of dot products is large enough, this could eventually cause overflow on the output scalar `dot`. In order to guard against this, we say `dot += alpha*dotprod(...)`, rather than `dot += dotprod(...)` (Note that `dot = dotprod()` is not safe, in that if the compiler figures out that `dotprod` is a pure function, the loop can be replaced with a single call). Alpha is initially set to 1, but every time we traverse all the working sets, we reverse the sign of alpha, so that in the second traversal of the working set we subtract off the same numbers that we added in during the previous traversal. Therefore, as long as we can make one traversal of the working set without overflow, this timer will not overflow (actually, due to error caused by aligning the mantissas of the numbers, it is possible to construct a case where overflow could still happen, but in practice it should not). We ensure that overflow doesn't happen for particularly long vectors by using input vectors that produce mixed sign data (so that the dot product is not always increasing in magnitude). Of course, we could

rule out any problem with overflow by initializing all vectors to zero, but since it is possible some hardware might handle this case more optimally than normal arithmetic (particularly architectures that do floating point in software) we prefer to use the `alpha` method instead.

Lazy page zeroing: If the kernel being timed accesses operands which are not initialized by the timer before invoking the kernel (eg., workspace or output operands), it becomes critical to access each page of data before making the call. This is because many OSes do "lazy zeroing" of pages. For security reasons, memory allocated from the system (which can include pages freed from another user's processes) must be zeroed before the allocating process is allowed to read it. In lazy zeroing, the newly-allocated page is not zeroed, but is protected so that any access of the page raises an exception. Then, when the allocating process attempts to access the unzeroed page, an exception handler which zeroes the entire page is called. The advantage of this scheme is in avoiding the overhead of zeroing pages that are never accessed, and postponing some overhead which allows user programs to begin operation faster. However, when this is allowed to happen during kernel timing, the timer will report a cost (which for some kernels is quite substantial) that only occasionally occurs: if the needed space is already available in user-space, no extra cost is seen, but if the allocation requires using a system page, then the cost is added. Whether this occurs depends not only on the particular OS and environment settings, but also on the initial state of the process's memory. The easiest fix for this is to be sure to access all memory that the kernel uses before calling the kernel, even when no initialization is required.

Effects of instruction load: Instruction load time can cause strong variance in kernel timings, particularly on the first call to the kernel. This is because most OSes load only a few pages of an executable at the beginning of the program and load the remaining pages only if and when they are needed. Thus, if the kernel isn't allocated to the same page as the kernel timer, the first call to the kernel may have disk access time added to its cost. Whether the kernel must be loaded from disk depends on a host of factors that defy a priori prediction, and therefore in the interest of getting repeatable timings it is advisable to ensure that the kernel routine is in memory before beginning timings, which can be done by making a dummy kernel call before commencing the timings.

In our own work, we make the dummy call to force the load of the kernel's instruction page before doing any cache flushing, so that the data-cache flushing that is performed will also flush any shared caches. Most machines have instruction and data share all caches except the L1 cache, which is almost always separate. This means that small kernels will probably be retained in the L1 cache. Unfortunately, we have not discovered a portable way to reliably flush the instruction cache, particularly when calling the kernel multiple times. Preloading the I-cache is a more realistic assumption in general than preloading the data cache, since it is usually a smaller cost and most kernels are called multiple times in loops

where it is highly likely that the kernel will be retained in some level of the cache. However, in those cases where the kernel in question is called only once and has low complexity (so that the computational costs do not dominate the instruction load), this may cause timings to be too optimistic. This could theoretically lead to poor optimization. For example, it might incorrectly show that an optimization that increases code size (eg. loop unrolling) is a performance win, when in fact for the way this kernel is called it is a loss due to the increased instruction loads. In our own work, most of the kernels have a large enough computational and data complexity compared to their code size that this cost is not very important, and so in the interest of getting repeatable timings we ensure that the kernel is in memory rather than on disk before beginning timings.

5.1   Enforcing memory alignment on operands

For some kernels on some architectures, the memory alignment of the operands can have a drastic effect on performance. Probably the most important example of this is SSE-enabled kernels on the x86. SSE (Intel's SIMD vectorization ISA extension) operates on vectors that are 16-byte in length. Most C libraries, however, have memory allocation routines that return memory only 8-byte aligned (this matches the size of C's `double`, which is the longest unmodified native C type). Like any data type, the SIMD vectors must be aligned to their native length to avoid the possibility of cache line splits, which can roughly double the cost of a load. Since malloc will, roughly speaking, return a 16-byte aligned address on only half of the calls, this can cause timing of codes which benefit from 16-byte alignment to vary widely in an unpredictable way. More specifically, a memory-bound SSE operation such as vector copy might actually run twice as fast when the timer gets "lucky" and allocates memory that is 16-byte aligned, and then in a second call gets "unlucky" and run at the cache line split speed.

The key to avoiding these problems is to program alignment adaptability into the timer. Again, context sensitivity demands the ability to vary the timer's behavior, since some users may typically call the kernel with aligned data, some unaligned, and some a mixture. Therefore, the timer needs to be able to generate both aligned and purposely misaligned operands (i.e. a kernel might have different performance for an 8-byte aligned address that is not allowed to be 16-byte aligned, than one that is 16-byte aligned). The kernel can then be tuned in multiple phases, where first the aligned kernel is optimized, and then this kernel is adapted to handle misaligned operands as efficiently as possible.

Forcing a given alignment on memory is simple in theory, but fairly complicated to get right in practice. Therefore Figure 7 shows wrappers that can be used to force alignment restrictions on memory allocations. In these routines, `align` is a nonzero variable that indicates what byte boundary the memory should be aligned to, while `misalign` indicates a greater alignment that the allocation should <u>not</u> be allowed be aligned to (eg., passing

```
void afree(void *P)                        // We allocated space. Move n past our
{                                          // 'extra' header space.
  size_t n = ((size_t) P) - sizeof(void*); n = ((size_t) P)+extra;
  n = n - (n % sizeof(void*));             n = n-(n%align); // Back up to align it.
  void **A = ((void**) n);
  free(*A);                                // n is aligned. Check if TOO aligned.
} // END *** afree ***                     if (misalign > align)
                                           {
/****************************************/    if ((n%misalign) == 0)
void *amalloc(size_t size, int align,           n += align;  // Force misalignment.
             int misalign)                 }
{
  void *P;                                 //-----------------------------------
  size_t extra, n, s;                      // n is the finished user pointer. Now
  extra=2*sizeof(void*)+align;             // back up from 'n' to a valid address
  if (misalign > align)                    // to store the original malloc void*,
    P = malloc(size+extra+align);          // and save it for afree() to use.
  else P = malloc(size+extra);             //-----------------------------------
                                           s = n - sizeof(void*); // Get space.
  // If malloc fails, we fail.             s = s - (s%sizeof(void*)); // Align it.
  if (P == NULL) return(NULL);             *((void**) s) = P;  // Store original.
                                           return( (void*) n); // Exit with new.
  // (continued in next column)          } // END *** amalloc ***
```

Figure 7: Wrapper code forcing a memory allocation to be aligned on an `align`-byte boundary, but not allowed to be aligned to a `misalign`-byte boundary. If `misalign=0`, then no maximal alignment is set. If it is nonzero, then `align` must be < `misalign`.

`align=4` and `misalign=16` asks for memory aligned to a 4-byte boundary that is not allowed to be aligned to a 16-byte boundary). If `misalign=0`, then no maximal alignment is to be forced.

The aligning wrappers `amalloc` and `afree` shown in Figure 7 allocate extra space in front of the user's requested area, which serves two purposes: First, the over-allocation provides room to position a pointer that is aligned as the user has requested. We call this the "user pointer". However, when we call the `free` routine to release the memory, we must provide the pointer received from the call to `malloc`, which we designate the "original pointer". So the extra space also stores the original pointer. In practice we must align it on a pointer boundary; but for any given user pointer the location of the original pointer is deterministic, so `afree` needs only the user pointer. The memory overhead we incur is (2*sizeof(void*)+align) on allocations that do not require forced misalignment, and (2*sizeof(void*)+2*align) for allocations that do require forced misalignment. The (2*sizeof(void*)) portion ensures enough space to store the original pointer on a pointer-

aligned boundary. If misalignment is needed we first set the user pointer to the earliest possible aligned position, then if that position also happens to be aligned on the prohibited boundary, we add the alignment increment to the user pointer, which will preserve the alignment but force the requested misalignment (hence the necessity of (2*align)).

5.2   Cache flushing for shared memory parallel timings

So far this paper has been concentrated on getting reliable serial timings. To extend these timing techniques to shared memory parallel operations, it is necessary to understand the cache state of all processors in the machine. The techniques we have been discussing should be adequate to flush any shared cache, at least those shared caches where one processor can access the entire shared cache. However, caches local to the processors that are used for computation, but which do not run the timing thread (which does flushing), must be handled explicitly. How this is handled will differ depending on the timer method being used. In the following discussion, assume that N_p is the number of processors possessed by the shared memory parallel computer, and p is the number of processors being used $(1 < p \le N_p)$.

For the single-invocation timers discussed in §3, there is sometimes no adaptation required for parallel operation. Since only the master (timer) process has initialized the operands, the operands should only be in cache on that process's processor, and the normal flush mechanism will ensure that no processors' cache is preloaded with the data. More sophisticated timers, however, are likely to use the operands repeatedly, even when each timing interval contains only a single kernel invocation. The most common reason is the need to perform multiple timings (each one of which consists of only one call to the kernel) of the kernel in order to get reliable timings, as described in §2.1. In these cases, the timer will often perform the kernel timing several times in a row on the same data. Since the prior kernel call will have brought some operands into the local caches of other processors, it will no longer suffice to flush only the master thread's cache. At first glance, it may appear sufficient to simply re-initialize the data, thus invalidating the other processors caches, but this assumes a certain shared memory cache protocol (i.e. this wouldn't work on some systems with snoopy caches).

Therefore the most straightforward adaptation is to simply spawn the appropriate cache flush loop of Figure 5 to all processors which might be used during the timed computation. If the timer cannot control which processors are used in the computation, it will be necessary to flush all processors in the system, even if the kernel uses only a subset of the available processors. To see why, imagine that the kernel selects to use only three processors for the given problem, and on the first invocation the threads were spawned to processors {0,3,5} of an 8 processor shared memory computer. We now wish to flush the cache before invoking the kernel a second time, and we spawn the flush loop using three threads, which happen to go to processors {0,2,4}, thus leaving processors 3 and 5 with preloaded data, which

will cause timing variations if these processors are used in a subsequent kernel invocation. Therefore, if the processors used when threading cannot be precisely controlled, it will by necessary to spawn the cache flush loop to all N_p processors, even if $p = 3$.

For the multiple-invocation timer given in Figure 6, adapting to parallel timing requires us to increase the cache flush area. In an operation like dot product, each processor of a p-thread parallel implementation (assuming one thread per processor) will need to access only a $\frac{\texttt{setsz}}{p}$ of data, where `setsz` is the size of the working set of the operation. Since the idea of this timing method is to not reuse data until it has been discarded from cache, we must therefore increase the size of our flush space by p (and, in the rare case of heterogeneous processors, choose our flush size based on the largest cache). There are some operations that do not exactly divide their working sets by p (for instance, perhaps they divide an output operand, but not the input), in which case a smaller increase will provide adequate cache flushing. Increasing by p should always provide a safe flush, though it may be too large an area (i.e., either it flushes a larger cache that we hoped to preload, or it is so large that the allocation fails). Again, if we cannot be sure the same p processors are used in each invocation, we will have to set p of the above discussion to N_p (and insist that only one thread be spawned to each processor), rather than allowing p to be based on the number of threads spawned by the kernel being timed.

6   Summary

This paper first introduced and demonstrated the importance of timing methodology in optimization, including highlighting its critical importance for automatic tuning frameworks. The following sections provided detailed discussions of the techniques we have found necessary to obtain high enough quality timings that automated optimizations decisions can be based on them in the real world. We have not found much mention of these techniques in the literature, though many are, or course, direct applications of basic architecture information. It seems likely that many hand tuners have probably used and reinvented similar techniques historically, though in our discussions we have seen no mention of the technique discussed in §4 that we developed in our original ATLAS work. At any rate, both discussions at conferences and the publication record firmly establish that many researchers are either unaware of the importance of taking careful timings, or do not know precisely how to perform them, and so we believe publishing these techniques explicitly will be a strong contribution to the field of optimization in general, and automated empirical optimization in particular.

References

[1] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the ACM SIGARC International Conference on SuperComputing*, Vienna, Austria, July 1997.

[2] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph Ueberhuber. Efficient utilization of simd extensions. Accepted for putblication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

[3] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.

[4] M. Frigo and S. G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, 1997.

[5] John Hennessy and David Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1990.

[6] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *CPC2000*, pages 35–44, 2000.

[7] J. Moura, J. Johnson, R. Johnson, D. Padua, M. Puschel, and M. Veloso. Spiral: Automatic implementation of signal processing algorithms. In *Proceedings of the Conference on High-Performance Embedded Computing*, MIT Lincoln Laboratories, Boston, MA, 2000.

[8] M. O'Boyle, N. Motogelwa, and P. Knijnenburg. Feedback assisted iterative compilation. In *LCR*, 2000.

[9] Markus Pushel, Jose Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Frenchetti, Aca Cacic, Yevgen Voronenko, Kang Chen, Robert Johnson, and Nick Rizzolo. Spiral: Code generation for dsp transforms. Accepted for putblication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

[10] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline – unrolling tradoffs. In *SCOPES99*, 1999.

[11] Paul van der Mark. Iterative compilation. Master's thesis, Leiden Institute of Advanced Computer Science, 1999.

[12] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing. (*to appear*).

[13] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. `http://www.netlib.org/lapack/lawns/lawn131.ps`.

[14] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.** `http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[15] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[16] R. Clint Whaley and Antoine Petitet. Atlas homepage. `http://math-atlas.sourceforge.net/`.

[17] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[18] R. Clint Whaley and David B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005 International Conference on Parallel Processing*, June 2005.