Automated Timer Generation for Empirical Tuning [⋆]

Josh Magee Qing Yi R. Clint Whaley

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX
{jmagee,qingyi,whaley}@cs.utsa.edu

Abstract. This paper presents a framework that significantly reduces the time required for automatically applying empirical tuning to improve the performance of large scientific applications, where the overall performance is often critically determined by a small number of individual routines that are either repetitively invoked or include a large number of loop iterations. Our framework allows these critical routines to be evaluated separately from their original applications by automatically generating timing drivers that accurately replicate their execution environment from within the whole applications. We have explored several alternatives to precisely simulate the input parameters, cache states of machines, and working environment of critical routines from both ATLAS and SPEC2006. Our experiments show that our timing drivers can accurately replicate the performance of these routines when invoked directly within whole applications, while reducing the time required to tune these routines by multiple orders of magnitude.

1 Introduction

In recent years, empirical tuning [9, 2, 4, 18, 19, 8, 11] has become a de facto approach that both developers and optimizing compilers adopt to extract high performance for scientific applications on a wide variety of modern computing platforms. However, since auto-tuning typically requires differently optimized code to be recompiled and re-executed hundreds or even thousands of times, the cost of experimentally evaluating a large optimization space could be prohibitive, especially for large scientific applications that take minutes or even hours to complete each run. In particular, for one of the SPEC2006 applications, we have found the time required to run the entire application is more than 175, 000 times longer than evaluating the routine of interest independently. It is therefore critically important to reduce the cost of each empirical evaluation of the optimized code, so that a sufficiently large optimization space can be explored to identify the desirable optimization configurations.

[⋆] This research is supported by the National Science Foundation under grant No. CCF-0833203, CCF-0747357, and CNS-0551504.

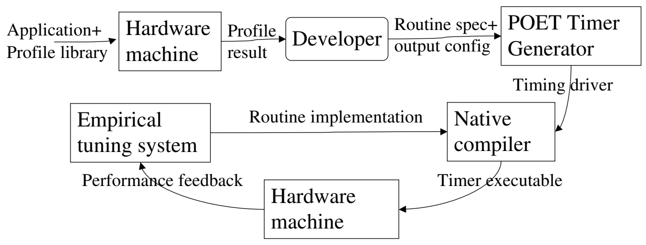


Fig. 1: The Timer Generation Framework.

We present a framework that significantly reduces the time required to empirically evaluate the performance of differently optimized code. Our framework is based on the observation that large scientific applications often critically depend on a few computationally intensive routines that are either invoked numerous times by the application and/or include a significant number of loop iterations. Since these routines are often chosen as the target of empirical performance tuning, the tuning time can be greatly reduced by separately studying the performance of an individual routine independent of the original application. Our experimental results show multiple orders of magnitude speedup in execution time when collecting performance feedback of a routine using an independent timing driver instead of instrumenting the whole application.

Separately studying the performance of individual routines requires a timing driver that 1)invokes the routine with an appropriate execution environment and 2)accurately reports the performance of each invocation. Whaley and Castaldo [17] showed that the measured performance could be seriously skewed when the runtime state of the system, especially caches, is not properly controlled by the timer before calling the routine. Unless the measured performance of the routine accurately reflects its expected performance when invoked directly within the whole application, the feedback could mislead the auto-tuning system into producing sub-optimal code.

Our framework automatically generates independent timing drivers that employ several approaches to precisely simulate the input parameters, cache states of machines, and working environment of individual routines so that their performance accurately matches those observed when instrumenting the original application. In particular, after identifying an individual routine to be tuned from within a large application, the work flow of our framework is shown in Figure 1. The framework starts with an instrumentation library which is invoked in a profiling run of the whole application to collect details of the routine's execution environment. The profiling result is then used to produce a *routine specification*, which is then used as input to a POET (Parameterized Optimizations for Empirical Tuning) [20] code generator to automatically produce a timing driver. The timing driver can then be compiled and repetitively used in the iterative performance tuning process, where every time a differently optimized code is generated for the routine, the routine implementation is compiled and linked with the timing driver to produce an executable (i.e., the *timer*), which measures the performance of the routine implementation on the targeted machine and reports performance back to the tuning system.

Our framework currently requires manual user intervention to profile the application and to write the routine specification after profiling. However, it provides a complete instrumentation library to collect the routine's execution environment (i.e., all the information required to write a *routine specification*) and provides a general-purpose code generator to automatically produce a reconfigurable implementation of the timing driver. Additionally, our framework provides a library that uses automatic *application checkpointing*[5] to control the execution environment of routines that are data-sensitive (e.g., an array sorting algorithm or a pointing chasing algorithm). We show that by checkpointing several instructions or iterations before invoking the routine of interest, the accuracy of timing results can be greatly improved.

We have applied our framework to tune several routines from ATLAS and SPEC2006. Our experiments show that our framework can accurately replicate the performance of these routines when invoked directly within whole applications, while significantly reducing the time required to tune these routines.

2 Related Work

Whaley and Castaldo [17] explored methods for achieving accurate and context-sensitive timing for code optimization. We have automated the generation of context-sensitive timers and have investigated both the efficiency and effectiveness of such timers in replicating the expected performance of routines invoked within applications. Apart from [17], the literature dedicated to accurately measuring performance has been limited to benchmarking systems [6, 13, 22].

The automatically generated timers presented in this paper can be integrated within a large body of existing auto-tuning frameworks, including many general-purpose iterative compilation frameworks [9, 1, 3, 11, 14, 19], and a large number of domain-specific tuning systems, e.g., ATLAS [18, 17], SPIRAL [10, 8], FFTW [4], PHiPAC [2], OSKI [15], and X-Ray [22], to provide performance feedback for the routines being optimized. The timers employed in ATLAS [16] are used in our research as a baseline for comparison.

When used to measure performance, most profiling systems, such as HPC-Toolkit [7], need to evaluate the entire application even when the performance of only a single routine is required. In contrast, our framework supports the timing of routines independent of their original applications. We use profiling only to collect the execution environment of routines.

Our application checkpointing approach (see Section 3.2) follows that documented in the ROSE compiler [12]. We have additionally introduced a delayed checkpointing approach that increases the accuracy of obtained timings.

3 Profiling Performance of Applications

The challenge faced when timing routines independently of their applications is ensuring that appropriate input values and operand workspaces are provided when invoking the routine. In particular, the supplied values should not result in abnormal execution (e.g., exceptions, segmentation faults) and should reflect the common usage pattern of the routine.

We have developed an instrumentation library to collect information on the common usage patterns (as well as the performance) of routines when invoked within an application. We separate these routines into two rough categories. The first category of routines is *data insensitive*; that is, the amount of computation within the routine is determined by a few integer parameters controlling problem size, but is not noticeably affected by the particular values stored in the input data structures. An example of such routines is dense matrix multiplication. The second category includes routines that are much more *data sensitive*, e.g., a sorting algorithm whose performance is largely determined by the specific data values being operated upon, as the algorithm may exit immediately after finding out the data are already sorted. Other examples include complex pointer-chasing algorithms whose input can only be roughly approximated.

Based on whether the routine of interest is data sensitive, we use different approaches to simulate its execution environment. In particular, for data-insensitive routines (e.g., matrix multiply), our *default timer* recreates their performance by reproducing the same array sizes, initializing arrays using a random number generator, and carefully controlling the memory hierarchy state to match those used in the whole applications. For data sensitive routines (e.g., a pointer-chasing algorithm), we use a *checkpointing* approach, discussed in Section 3.2. Note that the default timing approach may still allow reasonable tuning of some data-sensitive routines, as illustrated by our experimental results in Section 5.

3.1 The Default Timing approach

```
1:  $n \leftarrow$ number of routines
2:  $r \leftarrow$ list of routines to instrument
3:  $p \leftarrow$ maximum number of parameters $\times$ sizeof(largest parameter type)
4:  $i \leftarrow$ estimated calls to routine $+$ sizeof(double precision floating point)
5:  on application start, Setup instrumentation data buffer $D[n \times p \times i]$
6:  for each routine $i$ in $r$ do
7:      $v \leftarrow$ parameter values of $r_i$
8:      time $r_i$
9:      $t \leftarrow$ wall clock time of $r_i$
10:     $D \leftarrow$ key-value pair $r_i(v) : t$, avoid polluting cache
11: end for
12: on application termination, Write contents of $D$ to stdout or file
```

Fig. 2: Instrumentation Algorithm.

For data insensitive routines, we instrument their invocations within whole applications to record the integer parameter values and the wall clock times spent in evaluating each invocation. The collected information is then used to determine what values to supply when independently tuning these routines. The performance of routines collected within the applications is also used as reference in section 5 to determine the accuracy of the performance reported by our auto-generated timers.

Figure 2 presents the default approach to instrumenting the applications, where a data buffer D is created at the start of the application. The data buffer is of a configurable size, which is based upon the number of routines profiled, the maximum number of *sampled* parameters, and the estimated number of calls to each routine. Note that the maximum number of parameters include only parameters that are of interest (in most cases integer type parameters). Once setup is complete, every time a routine in r (the list of routines to instrument) is invoked, the routine parameter values (v) and execution time of the routine (t) are recorded. These values are written as a key-value pair to the data buffer (D) using instructions that avoid cache pollution. Upon application termination, the contents of data buffer D are written to a file or *stdout*.

3.2 The Checkpointing Approach

When invoked within whole applications, a data-sensitive routine sometimes has complex data structures (e.g., linked lists, trees and graphs) which are almost impossible to replicate without the original application. For these routines, we have adopted the *checkpointing* approach, a technique most commonly associated with providing fault tolerance in systems, to precisely capture the memory snapshot before the application invokes the routine. A similar checkpointing approach was first successfully employed in an end-to-end empirical optimization system built using the ROSE compiler [12].

Checkpointing works by saving a "snapshot" image of a running application. This image can be used to restart the application from a saved context. Our framework utilizes the Berkeley Lab Checkpoint/Restart (BLCR) library [5]. It facilitates checkpointing by providing a tiny library that can be used to generate a context image of an application.

```
enter_checkpoint(CHECKPOINTING_IMAGE_NAME);
.....
starttime=GetWallTime();
retval = mainGtU(i1, i2, block, quadrant, nblock, budget);
endtime=GetWallTime();
.....
stop_checkpoint();
```

Fig. 3: Creating a contextualized snapshot of a routine.

Figure 3 demonstrates how a context image for a call to the routine *mainGtU()* can be created using two calls: *enter_checkpoint* and *stop_checkpoint*. Specifically, the created image includes all the data in memory before calling *enter_checkpoint* and all the instructions between *enter_checkpoint* and *stop_checkpoint*. The image can be used in a specification file to our POET timer generator, discussed in Section 4, to automatically generate a checkpoint driver. The driver then contains code that loads and restarts the checkpoint image and reports the time spent in evaluating the routine of interest. Note that while the intended usage involves checkpointing a call to a routine, this approach can be used to measure the performance of any critical region of code.

It is possible to call *enter_checkpoint* immediately prior to the region of interest. However, this is not the case in Figure 3. Since restoring a checkpoint memory image does not restore any of the data into cache (and indeed may force normally-cached data onto disk), it essentially destroys the cache state of the original program. To restore the cache state before calling the routine of interest, it is better to call *enter_checkpoint* several loop iterations or instructions ahead of the region of interest, so that execution of these instructions can help restore the original cache state before the timed region is reached. How far in advance to place the *enter_checkpoint* call is a trade-off between reproduction accuracy and sample time.

To calculate the amount of "delay" between the *enter_checkpoint* call and the region of interest, the distance between the checkpoint and the region is incrementally increased until there is no noticeable variance in performance. In detail, the *enter_checkpoint* call is placed immediately prior to the region of interest and a timing is obtained. Next the *enter_checkpoint* call is placed in the previous iteration (when the region is in a loop) or the previous function invocation (when the region is not in a loop) and a new timing is obtained. The process is applied iteratively until the variation between timings falls below a chosen threshold. This process is semi-automated: once the boilerplate calls (*enter_checkpoint* and *stop_checkpoint*) are in place the best "delay" can be selected. We refer to placing the checkpoint immediately prior to the region of interest as "immediate checkpointing" and to placing the checkpoint several instructions prior to the region as "delayed checkpointing."

4 Automatically Generating Timers

After profiling an application to collect information on the execution environment of individual routines, we express such information in the format of a *routine specification*, which is then used by our POET timer generator (see Figure 1) to automatically produce a reconfigurable implementation (currently in the C language) of the timing driver, so that performance of varying routine implementations can be tuned independently of their original applications.

```
routine=void ATL_USERMM(const int M,const int N, const int K,
             const double alpha, const double* A,const int lda,
             const double* B,const int ldb, const double beta,
             double* C, const int ldc);
init={
  M=Macro(MS,72); N=Macro(NS,72); K=Macro(KS,72);
  lda=MS; ldb=KS; ldc=MS; alpha=1; beta=1;
  A=Matrix(double, M, K, RANDOM, flush|align(16));
  B=Matrix(double, K, N, RANDOM, flush|align(16));
  C=Matrix(double, M, N, RANDOM, flush|align(16));
} ;
flop="2*M*N*K+M*N";
```

Fig. 4: gemm.spec: Sample specification for the *GEMM* driver.

An example routine specification for a matrix multiplication kernel is shown in Figure 4, which includes a routine declaration, a section to allocate, initialize, and control the cache states of routine parameters, and a formula for computing the MFLOPS (*millions of floating point operations per second*). In particular, three integer parameters, M, N, and K, are initialized with environmental macros whose values can be dynamically adapted by simply re-compiling the timing driver; three matrices A, B, and C are allocated with appropriate sizes, initialized with pseudo-randomly generated data, aligned to a 16 byte boundary, and are flushed between timings.

Our timer generator in Figure 1 is essentially a translator written in POET [20], an interpreted transformation language designed for building ad-hoc translators between arbitrary languages (e.g. C/C++, Java) as well as applying sophisticated transformations to programs in these languages. The POET translator is extensively parametrized with variables whose values can be redefined via command-line options. Therefore, a wide variety of timing driver implementations can be manufactured from a single routine specification by redefining values of the command-line parameters, including the input/output file names, the output language, cache size, Instruction Set Architecture, processor clock rate, and timing methods (e.g., whether to use wall or CPU time).

A template of the auto-generated timing driver is shown in Figure 5 , which can be instantiated with the concrete syntax of different programming languages (our current work uses the C language) as well as different implementations (e.g., elapsed wall time, cycle-accurate wall time, or CPU cycles) to measure the performance of the invoked routine. Note that some of the control-flow statements (e.g., `for-endfor` and `if-endif`) in Figure 5 are not part of the generated code. They are used by the POET timer generator to selectively (in the case of `if-endif`) or repetitively (in the case of `for-endfor`) produce necessary statements in the resulting driver.

```
Require: Routine definition $R$
  for each routine parameter $s$ in $R$ do
    if$s$ is a pointer or array variable then allocate memory $m_s$ for $s$ endif
    if$s$ needs to be initialized then initialize $m_s$ endif
  end for
  for each repetition of timing do
    if Cache flushing = true then Flush Cache endif
    $time_s \leftarrow$ current time
    call $R$
    $time_e \leftarrow$ current time
    $times_r \leftarrow time_e - time_s$
  end for
  Calculate $min$, $max$, and $average$ times from $times_r$
  if flops is defined then
    Calculate Max MFLOPS as $flops \times \frac{1,000,000}{min}$
    Calculate Average MFLOPS as $flops \times \frac{1,000,000}{average}$
  end if
  Print All timings
```

Fig. 5: Template of auto-generated timing Driver.

The generated driver repetitively invokes the routine, optionally flushes the cache for each invocation, and measures the elapsed time spent while invoking the routine. Once all timing measurements are collected, the minimum, average, and maximum timings are calculated and reported. If a formula to calculate the number of floating point operations is included in the routine specification, the maximum and average MFLOPS are also computed and reported. Further, if the execution time of a single routine invocation is under clock resolution, multiple invocations can be collectively measured to increase timing accuracy.

We adopt sophisticated cache flushing mechanisms as discussed in [17] in our timing drivers. Specifically, the cache flushing strategy is re-configurable by command-line and every strategy makes sure that the flushing does not stand in the way of accurately measuring the performance of the routine being timed.

5 Experimental Evaluation

The goal of our evaluation is to validate that our POET-generated timers can not only significantly reduce the tuning time for large applications, they can accurately reproduce the performance of the timed routine when invoked within whole applications. To achieve this goal, Section 5.2 shows the reduction in tuning time achieved using our framework. Section 5.3 compares the timing results using our auto-generated timer for a matrix multiplication kernel with those obtained using the ATLAS timer, which is known to accurately reflect the common usage patterns of the kernel. Sections 5.4, 5.5, and 5.6 present performance results for three routines selected from different SPEC2006 benchmarks.

Our results show that our timers can perform similarly to the ATLAS timer in accurately reporting performance of data-insensitive routines in scientific computing, and that the performance results reported by our timers closely reproduce those collected directly from within the SPEC2006 benchmarks.

5.1 Methodology

We performed our evaluation on two multicore platforms: a 3.0Ghz Dual-Core AMD Opteron 2222 and a 3.0Ghz Quad-Core Intel Xeon Mac Pro. The timings are obtained in serial mode using a single core of each machine.

To compare our auto-generated *default timer* (see Section 3.1) with the ATLAS timer, we selected five different implementations of the ATLAS Matrix Multiply (referred to as *MMK*) kernel automatically generated using techniques presented in [21], where each implementation differs only in the cache blocking factor. The routine specification of this kernel is shown in Figure 4. The implementation of the kernel has been heavily optimized using techniques described by Yi and Whaley in [21], and achieves roughly 80% of theoretical peak performance when timed in cache (therefore, while this is not the fastest known kernel, it is quite good).

Each timer is tested for the totally cold-cache state (all operands cache flushed, labeled as **Flush** in figures) and with operands that have been allowed to preload the levels of cache by an unguarded operand initialization immediately prior to the timing call (labeled as **No Flush** in figures).

On the AMD platform, each *MMK* implementation is compiled using `gcc` 4.2.4 with optimization flags `-fomit-frame-pointer -mfpmath=387 -O2 -falign-loops=4 -m64` (note we use the x87 rather than the vector unit because on 2nd generation Opterons, both units have the same double precision theoretical peak, but the x87 unit has markedly decreased code size and increased precision). On the Intel platform, each *MMK* implementations is compiled using `gcc` 4.0.1 with optimization flags `-fomit-frame-pointer -mfpmath=sse -msse3 -O2 -m64`.

We selected the following routines from the SPEC2006 benchmark suite.

– Routine *mult_su3_mat_vec*, from 433.milc, performs an array-based matrix-vector multiplication and is data-insensitive (i.e., the performance does not depend on the content of the input arrays). We compare the performance of the POET-generated *default timer* using randomly generated arrays with timings obtained by profiling the whole application.

– Routine *mainGtU*, from 401.bzip2, is a variant of the *quicksort* algorithm using arrays. The computation depends on the content of the array being sorted and thus is data-sensitive. We compare the performance results obtained by the POET-generated *default timer* using randomly generated arrays, by the POET-generated *checkpoint timer*, and by profiling the application.
– *scan_for_patterns*, from 445.gobmk, is an extremely data-sensitive routine that operates on pointer-based linked-list data structures. We compare our POET-generated *checkpoint timer* with timings obtained by profiling the application.

Each SPEC2006 benchmark is compiled on both the AMD and the Intel platforms using a variety of optimization flags, including `-O0, -O1, -O2, -O3,` and `-Os`. All POET-generated timers themselves are compiled with the flag `-O2`. When using POET-generated *checkpoint timers*, we present results using both the immediate checkpointing and delayed checkpointing approaches (for details of these approaches, see Section 3.2).

5.2 Cost comparison of timing mechanisms

	Benchmark	**Delayed Checkpoint**	**Immediate Checkpoint**	**Default Timer**
mult_su3_mat_vec	877,430ms	3,502ms	3,510ms	5ms
mainGtU	45,765ms	2,019ms	1,975ms	4ms
scan_for_patterns	90,460ms	6,218ms	5,930ms	n/a

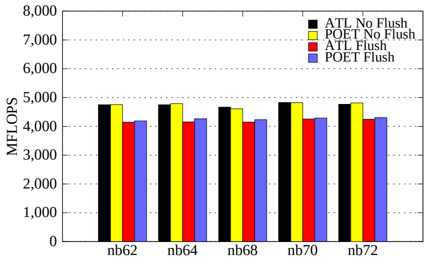
Table 1: Average runtimes for each SPEC2006 routine on AMD.

As shown in Table 1, the time required to collect performance feedback of a routine can be drastically reduced by using an independent timing driver instead of instrumenting the whole application. We see that our *default timer* is as much as 175,486 times faster than running the full benchmark, while even our *delayed checkpoint timer* is over 250 times faster. The timing results on the Intel machine are similar. These reductions in execution time are critically important for empirical tuning systems which may evaluate a routine hundreds or thousands of times.

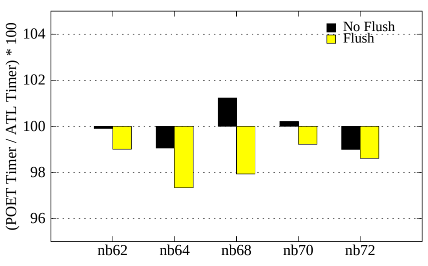
For data-sensitive routines such as *mainGtU* and *scan_for_patterns*, checkpointing or running the full benchmark yields the most accurate results. However, these methods are also more expensive than the default timer approach. The *default timer* takes significantly less time to run than the other approaches and therefore should be used where possible. The *checkpoint timers* take significantly less time than running the entire benchmark and should be used when the values of the input arrays are critical or when pointer-chasing routines are involved.

5.3 Comparing with The ATLAS Timer

Figure 6 compares the timings reported by the ATLAS timer and POET *default timer* with and without cache flushing for a *MMK* kernel using five different



(a) ATLAS timer versus POET timer.

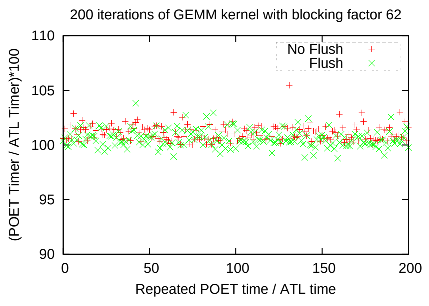


(b) POET as a percentage of ATLAS timer

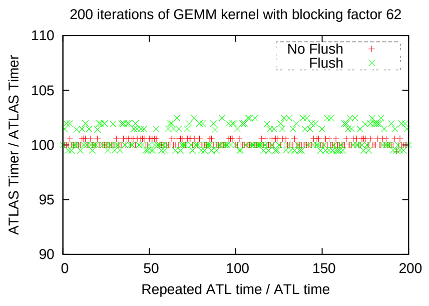
Fig. 6: ATLAS vs. POET timers with identical GEMM kernels on AMD.

cache blocking factors, where `nb62` denotes a blocking factor of 62*62 for the matrices. Figure 6(a) provides an overview of the timing results by comparing the MFLOPS measured by the ATLAS and the POET timers respectively. For each kernel implementation, the performance with cache flushing is slower than without flushing and the performance reported by ATLAS and POET are extremely close. This is easier to see in Figure 6(b), which reports the POET timer results as a percentage of the ATLAS timer results. For these kernel implementations, we see that the variation between the timers is less than 3%.

Timings taken on actual hardware running commodity OS are never precisely repeatable in detail. Since our POET timer is implemented independently of ATLAS, we expect some minor variance due to implementation details. The question is whether these relatively minor variances in timing represent errors, or if they are instead caused by the nature of empirical timings in the real world.



(a) ATLAS timer versus POET timer.



(b) ATLAS timer versus ATLAS timer.

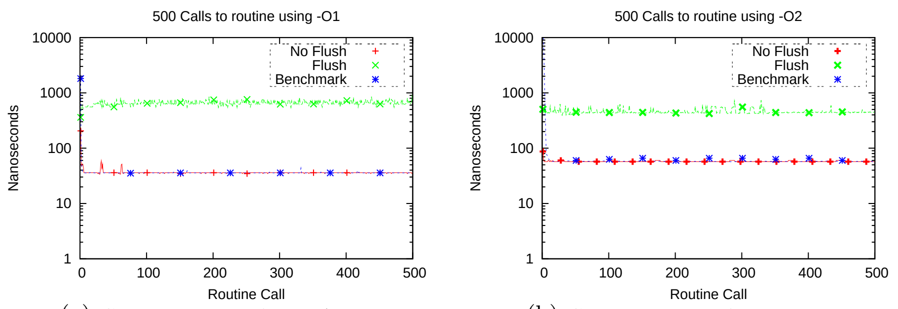
Fig. 7: Variance on a given kernel implementation between an initial ATLAS timing step with 200 subsequent results reported by POET or ATLAS timers on AMD.

Figure 7 sheds some light on this question: this figure shows the variation between timing runs for a single kernel implementation (with a predetermined blocking factor of 62), which we have timed 200 times. In Figure 7(a), we first run the ATLAS timer once for each cache state (`+`: no cache flushed, `x`: with cache flushing). We then time the same kernel 200 times using the POET *default timer*. For each cache state, we plot the POET timer's reported performance as a percentage of the original ATLAS measurement. We see some obvious jitter in the results, with variances that are mostly contained within a similar 3% range as we saw in Figure 6.

Figure 7(b) provides a strong indication that most of the observed variance is indeed due to the nature of empirical timings on actual machines. Here (using the same initial ATLAS timing as we compared against in Figure 7(a)), we do 200 more timings using the ATLAS timer itself. When we compare Figures 7 (a) and (b), we see that the variance between POET and ATLAS is only slightly greater than the variance between ATLAS and itself. Therefore, we conclude that our generated timer is able to adequately reproduce the behavior of ATLAS's hand-crafted timer for these hot and cold cache states.

5.4 Timing Results For SPEC2006 Routine *mult_su3_mat_vec*

This is a matrix-vector multiplication routine and is timed 1000 times using the POET-generated *default timer* (with arrays initialized with random values) both with and without cache flushing. The goal of this experiment is to verify that the *default timers* accurately replicates the measured performance from within the benchmark. Specifically, we expect that the *default timer* without cache flushing will accurately reproduce the benchmark's performance, since this routine is called with its operands in cache for all invocations except the initial call.

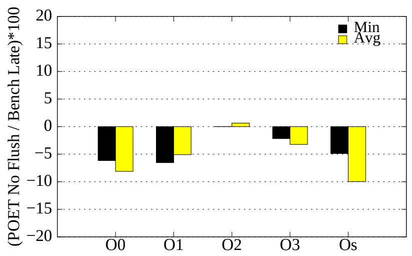


(a) Compiled with *-O1* on AMD. (b) Compiled with *-O2* on Intel.

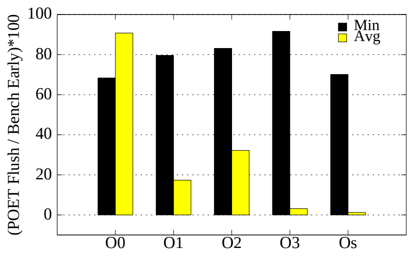
Fig. 8: Timings of 500 consecutive calls to *mult_su3_mat_vec*.

Figure 8 confirms our expectation by showing performance results of the routine when measured using the POET *default timers* (with and without cache flushing) and when measured from within the benchmark. As shown in Figure 8(a) and Figure 8(b), the default timings without flushing match extremely closely to the calls timed from within the benchmark, except when the routine is called the first couple of times (demonstrated by a lone aberrant $*$ along the 0th call), and a spike just past the 200th call of routine from within the application in (a). The spike is apparently due to unrelated activity affecting our wall times (this spike either disappears or shows up in other places if the timing is repeated). Therefore, we can classify this benchmark as using the kernel with warm caches (except the first few calls).

Since the routine uses statically initialized arrays, the input data are not in the cache on the first call, whose performance matches our **flush** timing results.



(a) No Flush / Late Group.



(b) Flush / Early Group.

Fig. 9: Minimum and Average timings of *mult_su3_vec* on AMD.

However, after calling the routine several times with the same workspace, the operands become cache contained, and so the overwhelming majority of calls closely match our **no flush** timings. On a cache with LRU cache line replacement, the second call would already have brought the data into any cache large enough to contain it. The machines we are using, however, have non-LRU replacement L2 caches, and this means that several passes over a piece of data are required before it is almost completely retained in the cache. Therefore, what we see is that the first call essentially runs at **flush** speed, and then the time for the next few calls decreases monotonically until the **no flush** time is reached. In such cases, it can be helpful to sort the usage contexts into several important groups, and tune them separately.

In Figure 9 we have sorted the first four routine calls into the **early** group, and all remaining calls into the **late** group. We see in Figure 9(a) that for both minimum and average times[1] that the **no flush** times are an extremely close match for the **late** group (verifying the general trend of Figure 8(a)). In Figure 9(b), we see that the **flush** group is not an exact match for our **early** group, which averages the first 4 calls, where only the first is fully out-of-cache. However, even this rough grouping would be adequate to tune this kernel for both contexts, assuming they were both important enough.

Our POET-generated timers can obviously capture the performance of routines when invoked either in-cache or fully out-of-cache, but for some applications the data may be only partially cache-contained. We can simulate these cases by partially flushing the cache in the *default timer*. Figure 10 demonstrates the ability of the *default timers* to capture the variation that arises as a result of the cache state. The *default timers* can reproduce a range of timings between the non-flushed and completely flushed state by using different flush sizes. This figure shows a progression of flushing sizes (from 512K to 2048K) in addition to no-flushing and complete flushing. Tracing through each flushing size (from none to full), a monotonic increase in the wall clock time can be observed such that an increase in the size of the flush results in an increase of time.

[1] maximum wall clock time can be extremely unstable and is therefore not used.

We can use the POET-generated *default timers* to reproduce timings that lie anywhere in between the **flush** and **no flush** lines. If the cache state of the application during a typical routine call is unknown (the usual case), profiling can be used to capture where the results lie, and the flush size can be adjusted so that the timer roughly reproduces the required cache state.

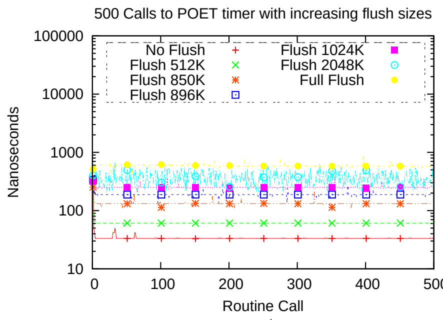
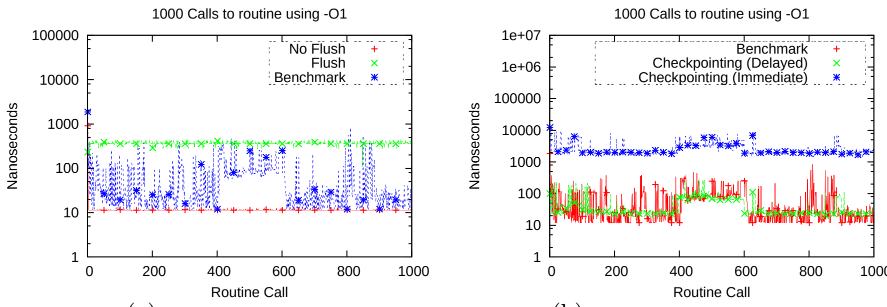


Fig. 10: POET Timers w/ increasing flush.

5.5 Timing Results For SPEC2006 Routine *mainGtU*

The routine *mainGtU* is a sorting algorithm whose performance is sensitive to the content of the input array. We have used both the POET-generated *default timer* and the POET-generated *checkpoint timer* to independently measure performance of this routine. The goal of this experiment is to determine how closely both the *default timer* and the *checkpoint timer* can reproduce the timings obtained from instrumenting the benchmark.



(a) POET timings. (b) Checkpointing timings.

Fig. 11: 1000 consecutive calls to *mainGtU* on AMD.

Figure 11(a) compares timings generated using the *default timer* with those taken from profiling the entire benchmark. Since the *default timer* uses randomly generated data, it will not capture cases where the data is in a particular order (eg., near sorted). Given these factors it is not expected that the POET-generated *default timer* should identically replicate each individual call; indeed unless the data distribution and order can be characterized, random data is probably as accurate as anything short of using the application's actual input.

If the precise behaviour of individual calls needs to be replicated, then checkpointing can yield accurate timings by exactly duplicating the input data. Figure 11(b)compares the results for both *immediate* (denoted by `*`) and *delayed* (`x`) checkpointing to those obtained from profiling the benchmark (`+`). Immediate checkpointing creates a checkpoint image of exactly one routine call. Delayed checkpointing creates a checkpoint image of several calls (three calls for this timing) to the routine, of which only the last call is timed. The methodology to determine how much to "delay" is discussed in section 3.2. Immediate checkpointing results in a cold memory hierarchy, and we see that these numbers are therefore even slower than our **flush** results from Figure 11(a). Delayed checkpointing allows for bringing the working set into the cache, thereby allowing for an extremely accurate reproduction of the timings taken from within the benchmark.

5.6 Timing Results For SPEC2006 Routine *scan_for_patterns*

The routine *scan_for_patterns* is a data-sensitive pattern matching algorithm that processes pointer-based linked-lists. Faithfully reproducing the necessary contexts for such a pointer-chasing algorithm is extremely difficult and often not feasible. To support the tuning of routines such as *scan_for_patterns* we use the POET-generated *checkpoint timer*. Here, the goal of our experiment is to demonstrate that the performance of complex pointer-chasing routines can be accurately replicated using the *checkpoint timer*.

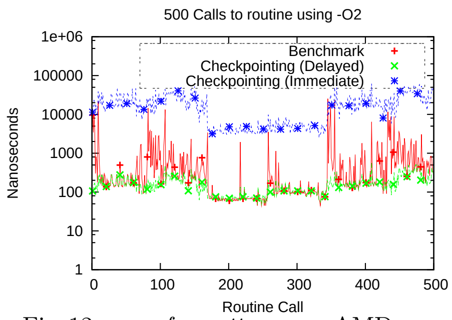


Fig. 12: *scan_for_patterns* on AMD.

Figure 12 compares the result of delayed and immediate checkpointing with timings obtained by profiling the benchmark. We can see that delayed checkpointing (designated by `x`) closely replicates the benchmark profiled timings (`+`) while immediate checkpointing follows the general trend of the benchmark (i.e., calls 200-300 are faster than 400-500) but with increased times due to the cold-cache state as discussed in Section 3.2.

6 Conclusion

This paper presented a general-purpose framework for automatically generating timing drivers that can accurately report the performance of computational routines in the context of automatic performance tuning. We have explored a variety of ways to accurately reproduce the common usage patterns of a routine so that when independently timing the routine, the reported performance results accurately reflect the expected performance of the routine when invoked directly within applications. We have demonstrated the importance of checkpointing several instructions or iterations before the routine being measured (delayed checkpointing) in obtaining accurate timings. Finally, we have shown that using the auto-generated timers can significantly reduce tuning time without compromising tuning accuracy.

References

1. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization, 2006. (CGO 2006).*, New York, NY, 2006.
2. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.
3. C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO*, San Jose, CA, USA, March 2005.
4. 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.
5. Future Technologies Group. Berkeley lab checkpoint/restart (blcr), 2009. https://ftg.lbl.gov/CheckpointRestart.
6. L. McVoy and C. Staelin. lmbench: portable tools for performance. In *Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, pages 35–44, Berkeley, California, 1996. USENIX Association.
7. J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 2002. In press. *Special Issue with selected papers from the Los Alamos Computer Science Institute Symposium.*
8. 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.
9. Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proc. Parallel Architectures and Compilation Techniques*, 2006.
10. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.
11. A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2004.
12. ROSE Team. Rose-based end-to-end empirical tuning: Draft user tutorial (associate with rose version 0.9.4a), 2009. www.rosecompiler.org.
13. A. Saavedra and R. Smith. Measuring cache and tlb performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, Oct 1995.
14. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, March 2005.
15. R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC*, San Francisco, California, 2005. Institute of Physics Publishing.
16. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Orlando, FL, Nov. 1998.
17. R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software—Practice and Experience*, 38(15):1621–1642, 2008.
18. R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.
19. R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *34th International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, 2005. IEEE Computer Society.
20. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Long Beach, California, Mar 2007.
21. Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.
22. K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *In Proceedings of the 2nd International Conference on Quantitative Evaluation of SysTems*, 2005.