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. The performance of real world applications often critically depends on a few computationally intensive routines that are either invoked numerous times by the application and/or include a significant number of loop iterations. It is desirable to separately study the performance of these critical routines, particularly in the context of automatic performance tuning, where a routine of interest is repetitively transformed using varying optimization strategies, and the performance of different implementations is gathered to guide further optimizations. This paper presents a framework for automatically generating timing drivers that independently measure the performance of critical routines separately from their original applications. We show that the timing drivers can accurately replicate the performance of routines when invoked directly within whole applications, thereby allowing critical routines to be correctly optimized in the context of empirical performance tuning.

1 Introduction

The ever-growing number of new computer architectures have made it increasingly difficult to precisely determine how to apply a large number of performance optimizations (e.g., cache and register blocking) to maximize the benefits presented by the platform in question. Manually determining the optimization parameters requires intimate knowledge of both the underlying architecture and the performance tradeoffs of various optimizations. Because such knowledge is often not readily available to regular developers, empirical performance tuning [9, 2, 4, 18, 19, 8, 11] has been used as a primary approach to addressing this problem. Specifically, by experimenting with different optimization configurations, developers (or feedback-directed compilers) can measure the performance of differently optimized code and use this feedback to guide further optimizations.

One issue that arises in the empirical tuning process is the collection of performance feedback for the optimized code. Both analytical performance models [22] and runtime measurements on real machines [9, 2, 4, 18, 8, 11] have been used to

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

serve this purpose. The analytical modeling approach builds abstract performance models based on detailed knowledge about both the underlying architecture and the input computation to be tuned. Because accurate performance models are difficult to build and are extremely sensitive to platform changes, the analytical modeling approach is often used only within compilers and other highly specialized systems. Runtime timing of the optimized code has remained as the dominant approach used by most tuning systems.

The performance of real world applications often critically depends on a few computationally intensive routines that are either invoked numerous times by the application and/or include a significant number of loop iterations. These routines are often chosen as the target of empirical performance tuning, where their runtime performance is repetitively measured with different optimization configurations until satisfactory performance is found. While the application can be instrumented so that performance feedback is collected whenever an interesting routine is invoked within the application, it is much more economical to measure the performance of the differently optimized routines independent of the application, so that the entire application does not need to be run whenever a single routine is optimized differently. Since the time required to compile and execute a single routine can be drastically less than the time needed to compile and execute an entire application, and empirical tuning may require differently optimized code to be recompiled and re-executed hundreds or even thousands of times, separately studying the performance of individual routines can significantly reduce the time needed for tuning an application.

Separately studying the performance of individual routines requires a timing driver that 1) invokes the routine with appropriate input values and 2) accurately reports the performance of each invocation. Manually creating these *drivers* can be tedious and error prone. 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 application, the feedback could mislead the auto-tuning system into producing sub-optimal code.

```
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. 1: gemm.spec: Sample specification for the *GEMM* driver.

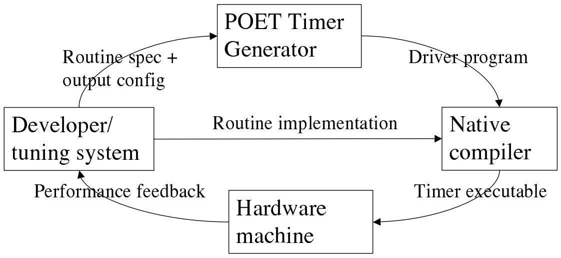


Fig. 2: The Timer Generation Framework

This paper presents a framework that automatically generates timing drivers from simple routine interface specifications. An example of the specification for a matrix multiplication kernel is shown in figure 1. By comparing our timers with that of the widely-used ATLAS empirical tuning framework [16] and by studying the use patterns of two critical routines selected from the SPEC2006 benchmark suite, we demonstrate that when used to independently measure the performance of individual routines, our generated timers can not only significantly reduce tuning time but also sufficiently replicate the expected performance of routines invoked within whole applications.

While some components within our framework (e.g., profiling of applications to collect use case information) are yet to be automated, our goal is to explore effective approaches of measuring the performance of computational routines in the context of automatic performance tuning. By studying the efficiency and effectiveness of different ways to produce the timing drivers, this paper provides insight for selecting a tuning method that is both efficient (in terms of the required tuning time) and effective (in terms of reporting performance that accurately reflects the expected performance of a routine when invoked directly within the application).

2 Automatically Generating Timers

Figure 2 illustrates our general-purpose framework for automatically generating timers to independently tune the performance of individual routines. Specifically, a developer or an auto-tuning system first produces a routine specification, an example of which is shown in Figure 1. The developer/tuning system then invokes the POET timer generator with the routine specification as input together with some command-line options to configure the timer generation. The POET timer generator produces a timing driver (currently in the C language), which is then compiled together with the routine implementation by a native complier to produce an executable (i.e., the *timer*) on the targeting machine. Finally, the *timer* is run on the targeting machine, and performance feedback is reported to the developer/tuning system.

The *POET timer generator* in Figure 2 is essentially a translator written in POET [20], an interpreted transformation language that can be used to build ad-hoc translators between arbitrary languages (e.g. C/C++, Java). A skeleton of the POET timer generator is shown in Figure 3.

```
<parameter inputFile message="input file name"/>
<parameter outputLang message="file name for output language syntax"/>
<parameter outputFile message="output file name"/>
<parameter CacheKB type=INT default=1000 message="The cache size in KB"/>
<parameter UseWallTime type=0..1 default=1 message="Whether to use wall clock time"/>
<parameter ISA type=_ default="generic" message="Instruction Set Architecture"/>
<parameter MHZ type=0.._ message="Clock rate of the processor"/>
......
<xform GenTimerDriver pars=(inputSpec)>
......
</xform>
......
<input from=(inputFile) syntax=(outputLang "specification.code") to=inputCode/>
<eval resultCode= XFORM.GenTimerDriver(inputCode);  />
<output to=(outputFile) syntax=(outputLang "timerCodeTemplates.code")
        from=resultCode/>
<output cond=(ISA=="x86" || ISA=="x86_64") to="GetCycleCount.S"
       syntax="timerCodeTemplates.code" from=(GetCycleCountCode#(ISA))/>
```

Fig. 3: The POET timer generator

The POET translator in Figure 3 starts with a number of external parameter declarations (e.g., *inputFile*, *outputLang* and *outputFile*), whose values can be redefined by command-line options, to control the configuration of timer generation. The parameterization allows different timer implementations to be manufactured on demand based upon different feature requirements, e.g., to target architectures with different ISAs, clock rates, and cache sizes & states.

The *GenTimerDriver* routine in Figure 3 is essentially a function that takes a single input parameter, the internal representation of the routine specification, and returns the generated driver program as result. After defining relevant transformation routines, the POET translator uses a single *input* command to parse the input file (i.e. the routine specification file), build an abstract syntax tree (AST) representation of the input code, and then store the AST into a user-defined variable (the *inputCode* variable). Note that AST is a built-in concept in POET (which makes it much easier to build translators in POET than using a general-purpose language such as C/C++/Java). The POET timer generator then invokes the *GenTimerDriver* routine to traverse the *inputCode* and apply various program analysis and translations. Finally, the resulting timing driver (stored in variable *resultCode*) is unparsed to an external file using the *output* command. If the x86 architecture is targeted, relevant assembly code to collect cycle counts is also produced as part of the output.

A template of the auto-generated timing driver is shown in Figure 4 , 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 4 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
    $time \leftarrow time_e - time_s$
    $times_r \leftarrow time$
  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. 4: Template of auto-generated timing Driver

As shown in Figure 4, the generated timer first allocates memory for the routine parameters and initializes input parameters for the routine. A loop is then used to control the number of times to repetitively invoke the routine. For each timing, the driver optionally flushes the cache, measures the elapsed time spent while invoking the routine of interest, and then records the time of each invocation. Finally, the minimum, average, and maximum of the collected 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 (*millions of floating point operations per second*) are also computed and reported.

Note that we adopt sophisticated cache flushing mechanisms as discussed in [17]. 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 interesting routine. Further, if the execution time of a single routine invocation is under clock resolution, multiple invocations could be collectively measured to increase the timing accuracy.

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 of the routine (i.e., exceptions, segmentation faults, and core dumps). Further, the supplied values should reflect the common usage pattern of the routine; that is, they should be representative of the majority of invocations from within real-world applications.

We have implemented an instrumentation library to collect information on the common use 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 not data sensitive; that is, the amount of computation within the routine is determined by a few integer parameters controlling problem size, but is not noticiably affected by the particular values stored in the input data structures. An example of such a data-insensitive operation is matrix multiplication. For these routines, it is possible to recreate their performance in applications by simply reproducing the same array sizes (initialized using a random number generator) and carefully controlling the memory heirarchy state to match that used in the actual application. The second category includes routines that are much more data sensitive and thus cannot be handled as easily. An example of such routines is a sorting algorithm whose performance is largely determined by the specific data values being operated on (e.g., the algorithm may exit immediately after finding out the data are already sorted). Another example is complex pointer-chasing algorithms whose input can only be roughly approximated outside a given application. To support the independent tuning of these routines, we have used a *checkpointing* approach, discussed in Section 4. Note that timer generation with random data may still allow reasonable tuning of some data-sensitive kernels, as illustrated by our experimental results in Section 5.

Our library instruments the invocations of routines *within* their respective applications to record their integer parameter values and the wall clock times spent in evaluating each invocation. The collected information is used as reference in determining what values to supply when independently tuning data-insensitive routines. Furthermore we use the collected performance information as reference in section 5 when comparing the performance of our automatically generated timers against timings collected within the applications, for both data-sensitive and insensitive routines.

4 Checkpointing

When invoked from within applications, a routine sometimes has complex pointer-based data structures (e.g., linked lists, trees and graphs) as input parameters. These input parameters are almost impossible to replicate in a separate timing driver. To handle these situations, 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 a 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. Figure 5 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 translator (see Section 2) to automatically generate a timing 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.

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

Fig. 5: Creating a contextualized snapshot of a routine

It is possible to call *enter_checkpoint* immediately prior to the region of interest. However, this is not the case in Figure 5. 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 function calls ahead the region of interest, so that execution of these instructions can 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; for the timings in this paper, we simply kept increasing the distance of the checkpoint until this stopped producing noticeable variance in performance. 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."

5 Experimental Evaluation

The goal of our experimental evaluation is to validate that using our automatically generated timers, independently tuning individual routines can accurately reproduce the expected performance of the timed routine when invoked within whole applications. ATLAS is an auto-tuning system [18] that is widely used in the scientific computing community. By comparing our timing results with those obtained from using the ATLAS matrix-multiplication timer (which was know to accurately reflect the common use patterns of the kernel), we show that our automatically generated timing drivers can similarly meet the need of accurately reporting performance of computation intensive routines in scientific computing. Furthermore, we present performance results for two routines selected from different benchmarks in the SPEC2006 suite. Here we first instrument the benchmark programs to gather both profile and performance information of the routines of interest. We then compare the profiled performance with our results of independently invoking and measuring the performance of individual routines using our auto-generated timers. We demonstrate that the performance results reported by our auto-generated timers closely reproduce those collected directly from within the applications.

5.1 Methodology

All the timings are obtained in serial mode using a single core of a 3.0Ghz AMD Opteron 2222 (2$^{\text{nd}}$ generation Opteron), using `gcc` 4.2.4 with various optimization flags (discussed below).

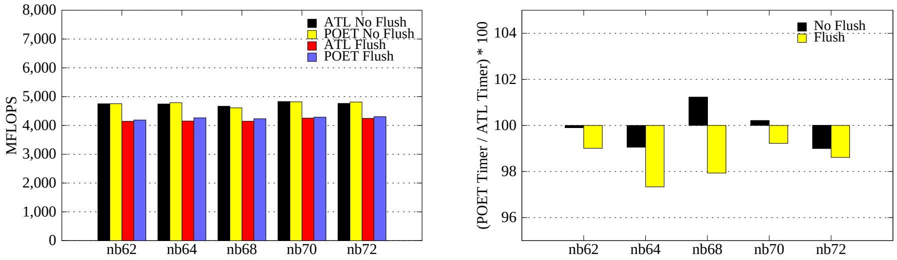
To compare our auto-generated timing drivers with the ATLAS timer, we selected five different implementations of a Matrix Multiply kernel where each implementation differs only in the cache blocking factor. The routine specification of this kernel is shown in Figure 1. The implementation of the kernel has been heavily optimized using techniques described by Yi and Whaley in [21], and achieves roughly 80% of theoretical performance when timed in cache (therefore, while this is not the fastest known kernel, it is quite good). We measure the performance of the optimized kernel using both ATLAS's hand-crafted timer (referred to as the **ATL** timer) and our auto-generated timer (referred to as the **POET** timer), and report results in MFLOPS(*millions of floating point operations per second*). 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 they can fit into by an unguarded operand initialization immediately prior to the timing call (labeled as **No Flush** in figures). All kernel implementations are 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 2$^{\text{nd}}$ generation Opterons, both units have the same double precision theoretical peak, but the x87 unit has markedly decreased code size and increased precision).

We selected the following routines from the SPEC2006 benchmark suite.

- Routine *mult_su3_mat_vec* from the 433.milc benchmark. This routine works on array based data structures and performs a matrix-vector multiplication. The routine is data-insensitive in that the computation required is not at all dependent on the content of the input matrix/vector (ignoring exceptional conditions such as NaNs or over/underflow). We compare the performance of independently timing the routine using randomly generated matrix/vectors (through our auto-generated POET timers) with timings obtained by profiling the whole application.
- Routine *mainGtU* from benchmark 401.bzip2. This routine uses array based data structures and is a variant of the *quicksort* algorithm. The routine is extremely data-sensitive in that the computation required depends on the content of the array being sorted. We have obtained independent timings for this routine using a randomly generated array as well as using our checkpointing approach. We then compare the independent timing results with the timings obtained by profiling the whole application.

We compiled each benchmark routine using a variety of optimization flags `-O0, -O1, -O2, -O3, -Os`, the standard optimization levels for `gcc`, and report performance results for all the optimization flags. The POET timing drivers themselves are always compiled with the flag `-O2`.

5.2 Comparing with The ATLAS Timer

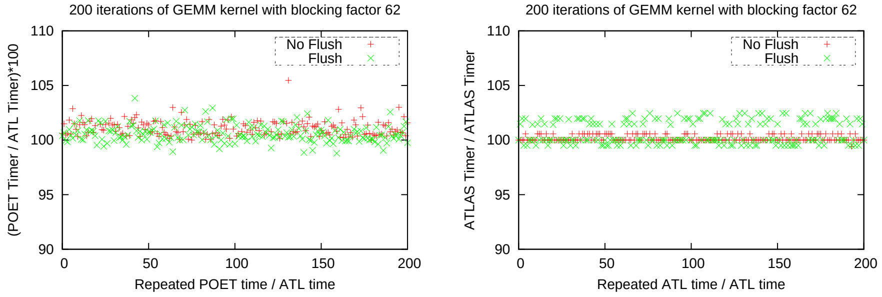


(a) ATLAS timer versus POET timer.

(b) POET-produced results as a percentage of the ATLAS timer results

Fig. 6: Comparison of ATLAS and POET timers on identical GEMM kernels.

Figure 6 compares the timings reported by the ATLAS and POET timers with and without cache flushing for the matrix multiplication kernel using five different cache blocking factors, where `nb62` denotes a cache 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 (as expected) 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%.



(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.

Timings taken on actual hardware running commodity OS are never precisely repeatable in detail. Since our generated timers are implemented independently from those in 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.

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 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.

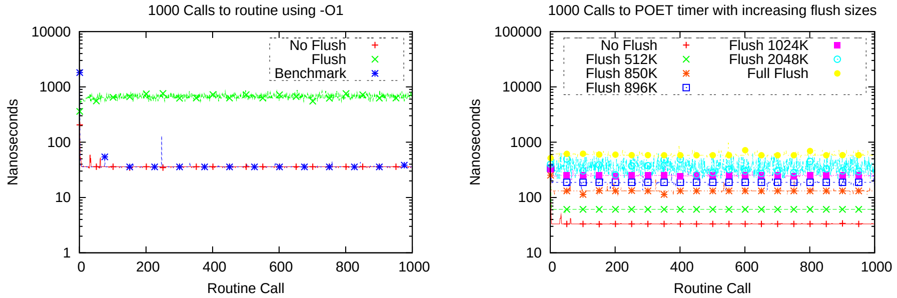
Figure 7(b) provides a strong indication that the most of the observed variance is indeed due to the nature of empirical timings on actual machines. Here (using the same initial ATLAS timing as we compared against in Figure 7(a)), we do 200 more timings using the ATLAS timer itself. When we compare Figures 7 (a) and (b), we see that the variance between POET and ATLAS is only slightly greater than the variance between ATLAS and itself (more specifically, while the ATLAS timings tend to cluster more tightly than POET, they appear to have almost the exact same worst-case variance over the 200 samples). 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.

5.3 Timing Results For SPEC2006 Routine *mult_su3_mat_vec*

This matrix-vector multiplication kernel was timed 1000 times using the POET timers (which initialize the matrix/vector with random values) both with and without cache flushing. Figure 9 shows performance results of the routine when measured using the POET-generated timers (with and without cache flushing) and when measured from within the benchmark. As shown in Figure 8(a), the POET 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 in (a)), and a spike just past the 200th call of routine from within the application. The spike was 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 mostly using the kernel with warm caches (except the first few calls) and the POET-generated timer could obviously be used to tune this kernel with this usage.

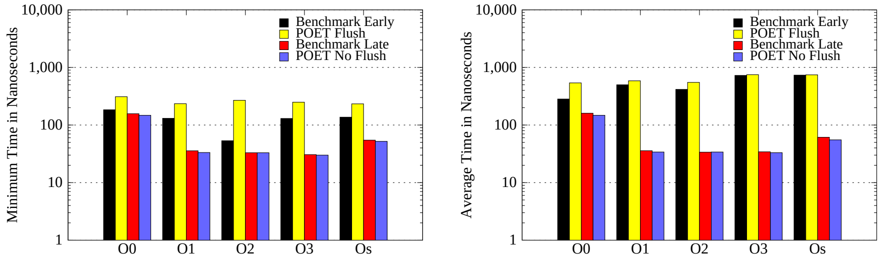
Since it uses static initialization, the data is not in the cache on the first call, which closely matches our **no flush** times. However, after calling the routine several times with the same workspace, the operands become cache contained, and so the overwhelming majority of these cases closely match 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 machine we are using, however, has a psuedo-random replacement L2 cache, 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 that for both minimum and average times that the **no flush** times are almost an exact match for the **late** group (verifying the general trend of Figure 8(a)). The **flush** group is not an exact match for our **early** group, since it averages the first 4 calls, where only the first is fully out-of-cache. However, even this rough grouping would certainly be adequate to tune this kernel for both contexts, assuming they were both important enough (in this case, tuning for the in-cache case would probably be all that was required).



(a) Benchmark compiled with *-O1* (b) POET Timers w/ increasing flush size

Fig. 8: Timings of 1000 consecutive calls to *mult_su3_mat_vec*

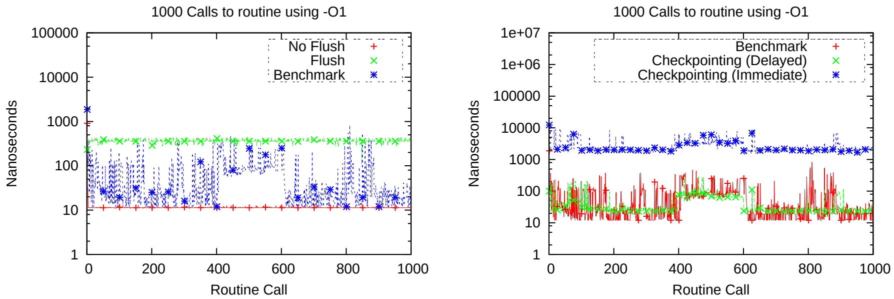


(a) Minimum of timings. (b) Average of timings.

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

Our 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 POET timer. Figure 8(b) demonstrates the ability of the POET timer to capture the variation that arises as a result of the cache state. The generated timers can recreate 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. This chart shows that we can use the POET-generated 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 between these two lines, and the flush size can be adjusted so that the timer faithfully reproduces the required cache state.

5.4 Timing Results For SPEC2006 Routine *mainGtU*



(a) POET timings.

(b) Checkpointing timings

Fig. 10: Timings of 1000 consecutive calls to *mainGtU*

The routine *mainGtU* from the 401.bzip2 benchmark is a sorting algorithm whose performance is sensitive to the content of the input array. We have used both random initialization of the array (the POET timer) and the checkpointing approach (the checkpointing timer) to independently measure performance of this routine. Figure 10(a) compares timings generated using the POET timer with those taken from profiling the entire benchmark. The POET timer uses randomly generated data, and so it will not capture cases where the data is in a particular order (eg., near sorted) or is particularly distributed (eg., lost of small numbers and a few very large ones in particular places in the input list). Given these factors it is not expected that the POET-generated 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 bahavior of individual calls needs to be replicated, then checkpointing can yield accurate timings by exactly duplicating the input data. Figure 10(b)compares the results for both *immediate* and *delayed* checkpointing to those obtained from profiling the benchmark. Immediate checkpointing creates a checkpoint image of exactly one call which is timed. Delayed checkpointing creates a checkpoint image of several calls (three calls for this timing) to the routine of which only the last is timed. The methodology to determine how much to "delay" is discussed in section 4. As discussed there, immediate checkpointing tends to result in a cold memory heirarchy, and we see that these numbers are therefore even slower than our **flush** results from Figure 10(a). Delayed checkpointing allows for bringing the working set into the cache, thereby allowing for an extremely accurate re-creation of the timings taken from within the benchmark.

5.5 Cost comparison of timing mechanisms

	Benchmark	**Delayed Checkpoint**	**Immediate Checkpoint**	**POET Timer**
Average Runtime	45,765 ms	2,019 ms	1,975 ms	4ms
% of Benchmark	100%	4.41%	4.32%	0.0087%

Table 1: Comparison of timing mechanism for *mainGtU*

For data-sensitive routines such as *mainGtU*, checkpointing or running the full benchmark yields the most accurate results. However, these more intrusive methods are significantly more costly in runtime, which can decrease a tuning package's ability to take multiple samples for greater statistial accuracy and to find good solutions in tolerable time. As shown in Table 1, the POET-generated timer takes significantly less time to run than the other approaches and therefore should be used where possible. Both checkpointing approaches take significantly less time than running the entire benchmark, so delayed checkpointing should be used when the input values of the data structures are critical.

6 Related Work

Our work follows [17], which discussed in detail various techniques for achieving accurate and context-sensitive timing for code optimization. We have automated the generation of such 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], there has been surprisingly little literature dedicated to accurately measuring improvements in performance, aside from papers on benchmarking systems [6, 13, 23].

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 [23], to provide performance feedback of the optimized routines.

The matrix multiplication kernel used in this paper (see Section 5) is automatically generated using techniques presented in [21]. The timers employed in ATLAS [16] are used in our research as a baseline for comparison.

Many application profiling systems, such as HPCToolkit [7], allow for procedure level timing. These tools can be used to obtain timing data for a routine. However the entire application must be executed even if the performance results for a single routine are required. Our framework aims to support the timing of single routines independent of their original applications.

Our implementation of the checkpointing approach follows that documented in the ROSE compiler [12]. We are not aware of any existing research that compares the performance obtained via checkpointing with those obtained via profiling applications.

7 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 use 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 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, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. A comparison of empirical and model-driven optimization. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".
23. 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.