# Using Dynamic Tracing Sampling to Measure Long Running Programs

Jeffrey Odom
Dept. of Computer Science
University of Maryland
College Park, MD 20742
jodom@cs.umd.edu

Luiz DeRose
Cray Inc
Mendota Heights, MN
ldr@cray.com

Kattamuri Ekanadham
IBM
T. J. Watson Research Center
Yorktown Heights, NY 10598
eknath@us.ibm.com

Jeffrey K. Hollingsworth
Dept. of Computer Science
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

Simone Sbaraglia
IBM
T. J. Watson Research Center
Yorktown Heights, NY 10598
sbaragli@us.ibm.com

## ABSTRACT

Detailed cache simulation can be useful to both system developers and application writers to understand an application's performance. However, measuring long running programs can be extremely slow. In this paper we present a technique to use dynamic sampling of trace snippets throughout an application's execution. We demonstrate that our approach improves accuracy compared to sampling a few timesteps at the beginning of execution by judiciously choosing the frequency, as well as the points in the control flow, at which samples are collected. Our approach is validated using the SIGMA tracing and simulation framework for the IBM Power family of processors.

## Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques; I.6.3 [**Simulation and Modeling**]: Applications

## General Terms

Measurement, Performance

## 1. INTRODUCTION

As the depth and complexity of memory hierarchies continue to grow, it is becoming increasing important for application developers and tuners to be able to understand the interaction of their program with the memory system. Hardware counters are widely used to provide insight into these issues. However, hardware counters are limited in the level of detail that they can provide. For example, only limited support is available to associate memory system events with data structures in the application. Alternatively, execution driven simulation has been effectively used for years to study performance as part of architectural studies. However, due to the significant slowdowns required for high fidelity simulations (1000-10,000x are common), this approach has not been widely used for tuning full applications.

Previously, we developed the SIGMA[4] framework to allow efficient collection of trace data from scientific applications. The SIGMA system also permits detailed simulation of architecture features. While this approach has proved useful, gathering trace data for long running programs (hours to days of running time) still requires too much time to be practical. Moreover, for many programs their performance varies during execution, and gathering data for only a few early timesteps does not provide an accurate picture of the application's performance. For example, adaptive mesh refinement (AMR) applications can have very different performance as the mesh evolves during the program's execution.

In this paper, we present a new technique that permits gathering detailed trace data based on sampling of timesteps throughout a program's execution. Our approach uses the Dyninst runtime code re-writing system to allow us to instrument only those timesteps we wish to sample. Using this approach, un-instrumented timesteps run at native speed, yet we are able to gather full address traces for selected timesteps.

The rest of this paper reviews the SIGMA tracing system, and then introduces dynSIGMA, our dynamic adaptive tracing system. We show that for one common benchmark, sampling only a few timesteps at the beginning of its execution results in an inaccurate characterization of the overall performance. We then present a validation study that shows that sampling only a few percent of the timesteps throughout the execution of an application provides representative information about the entire application. Finally, we present a case study that demonstrates the process of using dynSIGMA to identify and correct a TLB problem in a real program.

This research is supported by DOE Grants DE-FG02-01ER25510 and DE-CFC02-01ER25489, and NSF Award CNS-0406336.

SC|05 November 12-18, 2005, Seattle, Washington, USA

## 2. SIGMA

SIGMA [4, 5, 12] is a data-collection framework that gathers memory references using software-based instrumentation and provides a family of tools that simulate caches and provide detailed information on variables and functions. This data can be employed by programmers to improve cache performance. Typical tuning operations that programmers can perform include padding of data structures to improve cache alignment, blocking of their code to improve cache reuse, and loop fusion to increase cache and register reuse. The SIGMA infrastructure guides programmers in the tuning process by highlighting the precise memory references that are causing poor utilization of the memory system.

The SIGMA environment consists of a pre-execution binary instrumentation tool that locates and instruments all the instructions that refer to memory locations, a runtime data collection and compression engine that performs a highly efficient lossless compression of the memory stream, and a simulator of the memory subsystem. Users can choose to only monitor selected data structures and functions. Furthermore, SIGMA presents performance metrics expressed in terms of functions as well as data structures defined in the source code. For this purpose, SIGMA implements a runtime symbolic mapping engine, which associates each instruction address to the source line in the program that made the reference, and each data address to the symbolic name of the data structure that corresponds to the reference. The symbolic mapping engine supports both local and global variables as well as dynamically allocated data structures.

The SIGMA trace of execution captures the control flow as well as the memory addresses generated. The machine code of the program is divided into basic blocks, numbered sequentially so that each instruction can be uniquely identified with a pair *(blockNumber, blockOffset)*. The trace file consists of a sequence of *block-instances* of the form: $\langle blockNumber,\ entryOffset,\ memoryAddressesAccessed \rangle$. Each block instance completely specifies the sequence of instructions executed from the block and the addresses of locations accessed during that instance. The application can also inject *markers* into the SIGMA trace file. Markers are used to identify certain locations in the source code (like the beginning or end of a code segment or a function), to record the occurrence of certain events or to issue signals to the memory simulator. Signals can be employed for activating/deactivating certain controls, such as dumping/resetting caches and other counters.

Regular loops represented in the trace are in a very compact form. Pattern matching is performed to determine if control flow has caused a backward edge to be taken - that is, if a loop has been traversed. This is performed without any actual control flow analysis, relying solely on matching basic block numbers, thereby obviating the need for intra-procedural analysis. A linear regression is performed on each set of memory accesses within a matching sequence of basic blocks, and if a sequence of memory accesses can be represented as a linear relation, only the coefficients of the relation are output. This leads to a simple yet efficient compression of regular memory accesses. This approach has been used with up to 4 levels of nested loops.

The SIGMA cache simulator takes two inputs: a SIGMA trace and a specification of the memory system. It simulates the memory subsystem for the sequence of references made in the trace and provides summary statistics on the cache performance. The simulation results are expressed in terms of source level functions and data structures and a condensed repository of statistics is prepared. The repository is organized in such a manner that statistics can be displayed based on functions or data structures. An interactive interface allows the user to query the repository, compute derived metrics and create tables and bar charts.

## 3. SAMPLING TECHNIQUE

To illustrate the results of the above apparatus, we choose the following crude, but effective, metric to gauge the memory performance of the application. The Sigma simulator provides the counts of hits/misses at various levels of the caches and TLB. The machine architecture manual provides estimates of average latencies at each of these components. Ignoring the concurrent behavior of the components of the memory system, we can estimate the total time spent in the memory subsystem by multiplying the miss counts with the respective latencies, thus yielding the following metric:

$$memtime = \sum_{i=1}^{n} (h_i \cdot l_i) + Tmiss \cdot Tlatency \qquad (1)$$

where $n$ denotes the number of cache levels, $h_i$ the number of hits at level $i$, $l_i$ the latency of the cache level $i$, $Tmiss$ the number of TLB misses and $Tlatency$ the penalty associated with a TLB miss. This metric represents an approximation of the time spent in memory operations. While cache concurrency is ignored in this metric, we observed that memtimes compared reasonably well with wall clock times on several NERSC codes.

There are two sets of statistics that we utilize. First, we observe the results of the simulation for each global memory object. This includes C static and extern variables that are contained within the executable proper (static variables in and extern variables imported from shared libraries are not included). For Fortran, we include the variables that are defined within a common block, even if that common block is never shared between functions.

In an orthogonal view, we can gather the statistics for loads and stores on a per-function basis. Similar to our criteria for data objects, we only consider functions that reside within the executable itself. The choice of view, data-centric or function-centric, is one that is be determined by the analyst.

For each memory object, we gather the statistics used in equation (1) and calculate the memtime. This represents the total amount of time spent, within the simulation, for all accesses to that object. Each object is then given a canonical name (we chose *variable* for global variables and *variable@function* for local variables), alphabetically sorted, and an array of memory times is output for the program. We consider this the *characteristic data object pattern*. A *characteristic function pattern* is generated in a similar fashion. Since neither view is more or less important, for the remainder of this section we use the characteristic data object pattern in our examples.

This process, however, is not without its drawbacks. Most significantly, instrumenting every load and store results in a dramatic increase in runtime of the application. Slowdowns of 6000x, just in trace generation alone, are not uncommon. Of secondary interest is the size of the trace file, which can grow to significant size for short-running (as measured by uninstrumented execution time) applications. For instance, seis_s, which runs in 8 seconds uninstrumented, generates a trace file of 1.9G.

We therefore propose that a *sampled* trace may approximate a full trace to a good degree. Instead of recording all memory accesses, we only record those inside the work loop. Additionally, we record work loops only periodically. That is, although we record all memory accesses from the beginning of the outermost work loop to the end of that same loop, we only record every $M$ (mod 100) loops, giving an $M$% sampled trace. As we demonstrate in section 5, these sampled traces are good representations of the actual memory behavior of an application.

# 4. IMPLEMENTATION

Dyninst[2] is an Application Program Interface to a library that permits the insertion of code into a running program. The library provides a machine-independent interface to permit the creation of tools and applications that use runtime code patching. The application being modified is able to continue execution without recompilation, relinking or restarting. This is accomplished by direct manipulation of the application's user space. Although we only consider POWER-based executables here, Dyninst supports several architectures and operating systems. Because the interface to the developer is machine-independent, very little modification is needed to extend runtime modifications to other platforms.

We use Dyninst to enumerate every load and store instruction, as well as every call to heap allocation functions, and to replace them with wrapper routines that insert the memory access or memory allocation/deallocation into the Sigma trace. All compression and file I/O is done entirely within the process space of the instrumented application to eliminate additional execution time resulting from context switching.

We modify the source of the application to be measured with calls to functions *sample_begin* and *sample_end*, which delineate the timestep boundaries. Currently, these modifications must be done by hand by analyzing the control flow of the application, identifying the outermost section of the main work loop, adding these two additional function calls, and recompiling the binary. Note that these functions are initially empty - that is, while we have modified the application at the source level, all we have done is insert two function calls that immediately return. We also place markers around calls to memory allocation/deallocation routines, if they are performed outside the work loop. This is done so as to track ownership of dynamic memory allocations. All major modification is performed at runtime.

Under Dyninst, to insert instrumentation, a separate controlling application called a *mutator* is invoked that executes the application to be instrumented (*mutatee*) and adds or removes instrumentation by manipulating the mutatee's process space. When the application to be instrumented is first loaded, a shared library, *sigmatra.so*, is also loaded into the mutatee's process space. Then, calls to these dummy functions, which have no code in the executable itself, are replaced inline with calls to corresponding functions within the shared library. This allows different sampling algorithms to be run without recompilation of the instrumented executable.

As stated previously, instrumentation of the application is controlled by the mutator. Dyninst also allows, in addition to insertion and deletion of instrumentation, deactivation and reactivation of that instrumentation. Those operations are also performed by the mutator. To perform trace sampling, we deactivate and reactive instrumentation on a periodic basis, whenever the application reaches *sample_begin* or *sample_end*. To alert the mutator that one of these points have been reached, the instrumented application sends a signal (SIGSTOP) to itself, which alerts the mutator and as a side effect also pauses execution of the application. When the mutator sees that the application has paused, it toggles the tracing activation state of the instrumentation, inserts a marker containing the current sample iteration into the trace, and then continues the application's execution. The periodicity at which these signals are emitted is controlled via the environment variable *DYNSIGMA_PERIOD*.

# 5. EVALUATION OF SAMPLING

We chose five benchmarks on which to perform our analysis, three from the NAS Parallel Benchmark suite v3.0 (bt.W, lu.W, sp.W), one from the SPECFP2000 suite (swim), and one from the SPECHPC2000 suite (seis_s). We limited the benchmarks we analyzed to those with at least 300 iterations, so that a 1.0% sampling rate would provide at least 3 timesteps. Each benchmark was compiled using the IBM native compiler (xlf or xlc), with an optimization level of -O3. All experiments were performed on an IBM p670 with 8 POWER4 processors, running AIX 5.2 with 8.0G of memory, and used SIGMA 2.1.4 and a modified version of Dyninst 4.1.

First, each benchmark was instrumented with the static SIGMA tool. The instrumentation itself was performed offline via binary rewriting; that is, after the instrumentation phase of program *foo*, a separate program *foo.inst* was produced that contained the instrumented code. All load and store instructions were instrumented, with the exception of loads into R2 (the Table of Contents pointer), which is intentionally ignored by SIGMA. Then the rewritten, instrumented program is run which produces a trace file for each thread of execution. While the architecture is present to capture memory references on a per-thread basis in both the static and Dyninst based SIGMA, for simplicity all benchmarks analyzed were single-threaded.

Next, the trace file was input into the SIGMA cache simulator, which recorded statistics such as accesses, hits, and misses differentiated by access type (load or store), cache level (L1, L2, L3, or TLB), function, and data structure. While the SIGMA simulator allows for modeling of different cache architectures, we chose as our model the same architecture we ran the simulator on, that is the POWER4. The

results were then output into a human-readable file and an XML-based file for further processing. For statistics generated by binaries instrumented with the static SIGMA tool, we refer to these as the "Full" versions, as the static instrumentation performed no sampling.

To achieve sampling, we instrumented the benchmarks with our Dyninst-based version of SIGMA. Unlike the static version, our instrumentation is all performed online, with no intermediate binary produced. The instrumented in-memory image produces a trace file in the same format as the rewritten executable, so the results of the sampled instrumentation were fed directly into the simulator as in the Full case.

To compare a sampled trace with the output from a full program, the characteristic data object patterns of both sets of traces are examined. We perform a simple linear correlation using Pearson's $r$[11] between the two arrays. This computes a weighted variation difference between vectors of values. The goodness of the sample is then given by the linear correlation coefficient obtained (Pearson's $r$). For values $r \approx 1$, we can say that the sampled trace is *nearly* representative of the program's execution. This means that the access pattern for the data objects approximates the true execution *relative* to each object. In particular, the information provided to an analyst about the weighted ranked order of data (either functions or data structures) is identical if the correlation coefficient is 1.

## 5.1 Sampling of One Timestep is Not Enough

Of course, it is logical to ask at this point if sampling of 1%, 5%, 10%, etc. is even necessary. After all, boundaries of the work loop do not change throughout the lifetime of the execution. It is tempting to choose a single timestep as representative of every loop iteration. And for applications with regular memory accesses and regular control flow behavior, this may be an acceptable strategy. We shall see, however, that an application with irregular control flow within a timestep requires more than a single sample.

Let us consider the seis_s benchmark. Its work loop consists of a variable pipelined sequence of processing steps. The sequence is variable because, during the lifetime of the simulation, as processing steps at the head of the pipeline complete they are removed from the work queue. This results in an irregular control flow structure for the application. Because of this irregularity, sampling just a single timestep or even a few early ones) is not sufficient to represent the characteristics of the application. To show this, we consider three single timestep samples taken at the beginning, middle, and end of the seis_s application. It happens that the memory behavior of the first and last timesteps are identical.

In Table 1, we see the top five memtime data structures for the full trace, a 2.5% sampled trace, and the three single timesteps (with the first and last timesteps presented together). Some of the data points are labeled as "unknownDt@*function*" These correspond to memory accesses within a function that, because of optimization (or additionally, in the case of sampling, stack-based variables created outside the scope of the sample), cannot be attributed to a particular data object. Those variables without an '@' in them are global variables.

| | Full | 2.5% | Begin./ End | Middle |
|---|---|---|---|---|
| sa | 65 | 64 | 0 | 0 |
| otr | 19 | 24 | 0 | 5 |
| unknownDt@r8syn | 8 | 7 | 0 | 9 |
| unknownDt@ord2 | 2 | 1 | 0 | 3 |
| ra | 1 | 1 | 0 | 72 |
| unknownDt@r8tr | $< 1$ | $< 1$ | $< 1$ | 8 |
| unknownDt@seisproc | $< 1$ | $< 1$ | 55 | $< 1$ |
| name | $< 1$ | $< 1$ | 16 | $< 1$ |
| unknownDt@cproc | $< 1$ | $< 1$ | 16 | $< 1$ |
| unknownDt@sysproc | $< 1$ | $< 1$ | 9 | $< 1$ |

**Table 1: Percentage of memtime for data structures for full trace, sampled, and single timesteps in seis_s**

Note that the set of data structures referenced in the first and last timesteps are almost completely different from ones taken throughout the entire execution. A timestep taken in the middle of the execution has more similarities with the full trace, but even then its data reference pattern is skewed more towards the structure *ra* than *sa*. This is reflected in Table 2, where the correlation is extremely poor for the beginning/end timestep and middle timestep compared to a 2.5% sample.

| | 2.5% | Begin./ End | Middle |
|---|---|---|---|
| Linear correlation | 0.997124 | -0.043267 | 0.009820 |

**Table 2: Linear correlation of memtime for data structures in seis_s for 2.5% sampling compared to single timesteps**

## 5.2 Accuracy of Sampling

We now expand our analysis to include the remaining applications, and examine the effect of sampling rates on our accuracy. For every application, we varied the sampling rate between 1%, 2.5%, 5%, and 10%. Although we used the data from all the data structures for our calculations, in our analysis we were particularly focused on the top consumers of memtime, as would typically be the case for any application tuner.

Figures 1, 2, 3, 4 and 5 show the data structures that constitute the majority of the memtime in each of the benchmarks (with respect to the full trace), and their associated memtime percentages. Each series represents a sampling rate (or Full for the static instrumentation traces). As can be seen in each of the figures, the sampled traces are quite representative of the memory access patterns.

To quantitatively show the correctness of our sampling technique, first we consider different sampling rates. We take the memtimes for all variables in each sampling run and calculate the linear correlation against the respective memtimes of the full simulation. Variables with no data (i.e. variables that are only accessed outside the boundaries of a timestep) are given a zero memtime. The results in Table 3 show these linear correlation coefficients for four different sampling rates, as compared to the total memtime (all cache levels) for every data structure.

| | BT | LU | SP | seis | swim |
|---|---|---|---|---|---|
| 1.0% | 0.999994 | 0.999969 | 0.999812 | 0.996139 | 0.999382 |
| 2.5% | 0.999998 | 0.999957 | 0.999812 | 0.997124 | 0.999888 |
| 5.0% | 0.999999 | 0.999965 | 0.999811 | 0.997307 | 0.999964 |
| 10.0% | 0.999999 | 0.999980 | 0.999811 | 0.997131 | 0.999985 |

**Table 3: Linear correlation of sampling at various rates**

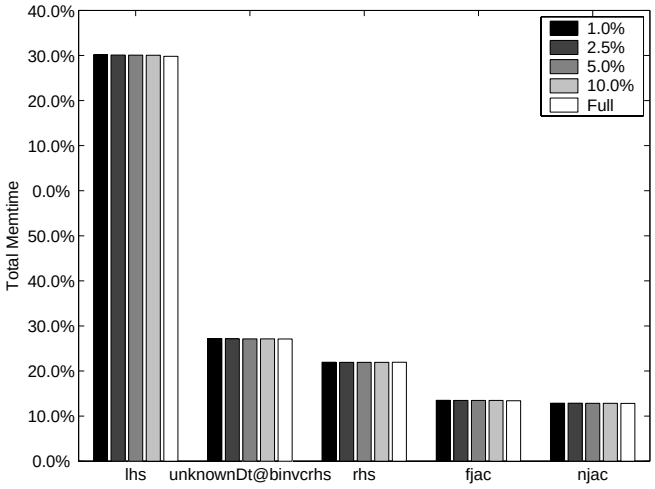


**Figure 1: Top 5 memtime data structures, bt.W**

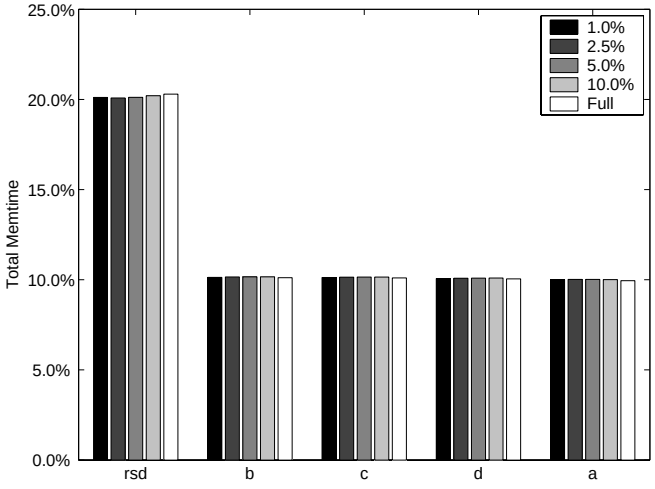


**Figure 2: Top 5 memtime data structures, lu.W**

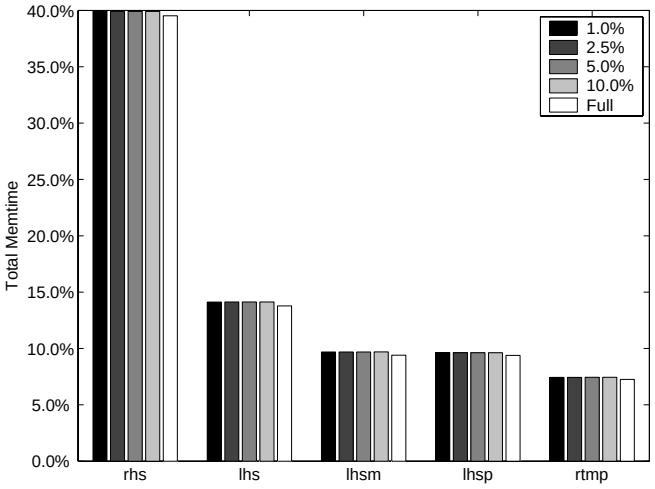


**Figure 3: Top 5 memtime data structures, sp.W**

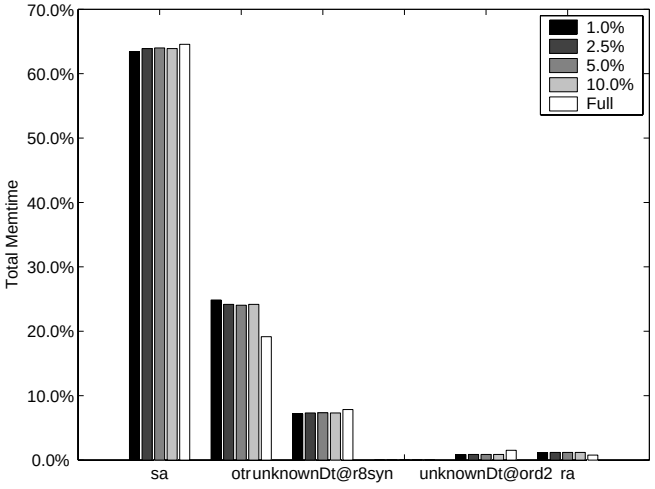


**Figure 4: Top 5 memtime data structures, seis_s**

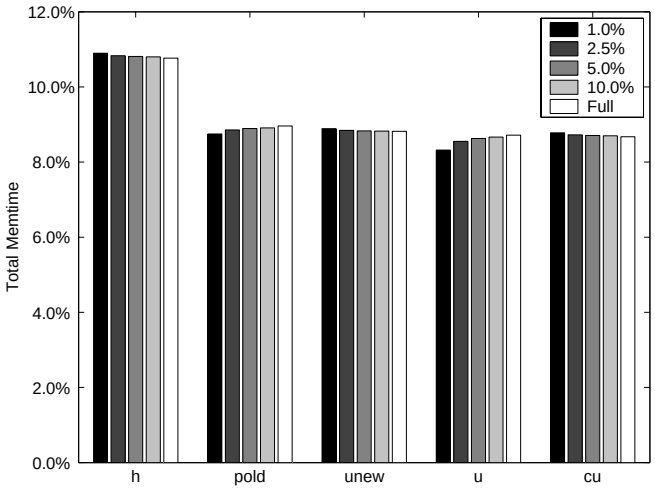


**Figure 5: Top 5 memtime data structures, swim**

We believe that 2.5% represents a good tradeoff between trace size and accuracy. To demonstrate that we examined the correlation of memtime for various cache levels, including L1, L1 plus contributions from L2, L1 plus L2 and L3, and all cache levels including TLB and main memory. These results are summarized in Table 4.

| | L1 | L1+L2 | L1+L2+L3 | All |
|---|---|---|---|---|
| BT | 0.999998 | 0.999999 | 0.999997 | 0.999998 |
| LU | 0.999944 | 0.999986 | 0.999940 | 0.999957 |
| SP | 0.999994 | 0.999994 | 0.999790 | 0.999812 |
| seis | 0.988851 | 0.997133 | 0.997130 | 0.997124 |
| swim | 0.997257 | 0.999766 | 0.997296 | 0.999888 |

**Table 4: Linear correlation of 2.5% sampling for various levels of memory heirarchy**

## 5.3 Space and Time Gains from Sampling

Finally, we consider the gains in trace size and running time (of both the instrumentation phase and subsequent simulation) from sampling timesteps. First, we examine the reduction in storage space for memory reference traces, as given in Table 5. A couple of points of explanation need to be made here. Note that for LU and SP, all the sampled traces are roughly the same size. This is because the regularity of the computations performed during the timesteps allow the memory references to compress extremely well in the SIGMA trace format. In fact, the only reason the file sizes are not the same for each trace is the additional markers that are inserted to track the boundary of each sample. Additionally, for these benchmarks, the sampled traces are only roughly 1% of the Full trace size. Again this comes from the fact that we only examine memory accesses within timesteps - thereby excluding initialization, finalization, and reporting or I/O. As can be seen from earlier tables, these phases do not constitute a significant portion of memtime for the overall application, but they can contribute significantly to the size of the generated trace.

While trace sizes can grow to be rather large, especially for programs with irregular memory access patterns, often the size of the trace is not the limiting factor when considering trace-based simulations. Rather, it is the amount of additional time spent in instrumentation, trace generation, and simulation that is the biggest burden on an application analyst. Consider the slowdown to the uninstrumented executable when static and dynamic SIGMA tracing takes place, as shown in Table 6. In this table, the value in each column represents the total running time of trace generation and simulation for each sampling level, as a factor of the uninstrumented time, with the exception of the last column which is the running time of the uninstrumented executable in seconds. As we have stated previously, slowdowns in the order of magnitude of 1000 are the norm, not the exception. While the toggling of trace generation does incur some overhead (so for instance a 10% sampling takes slightly more than 10% of the Full time), it is small compared to the savings in overall execution time.

| | 1.0% | 2.5% | 5.0% | 10.0% | Full |
|---|---|---|---|---|---|
| BT | 885 | 2,213 | 4,426 | 8,852 | 74,221 |
| LU | 166 | 168 | 172 | 180 | 16,390 |
| SP | 192 | 192 | 192 | 193 | 21,832 |
| seis | 13,832 | 33,929 | 67,922 | 136,370 | 1,934,667 |
| swim | 19 | 19 | 19 | 20 | 29 |

**Table 5: Trace sizes in KB at various sampling rates**

| | 1.0% | 2.5% | 5.0% | 10.0% | Full | Uninst. Runtime |
|---|---|---|---|---|---|---|
| BT | 90 | 233 | 443 | 713 | 6000 | 8s |
| LU | 63 | 206 | 286 | 463 | 3567 | 21s |
| SP | 47 | 96 | 231 | 469 | 4400 | 27s |
| seis | 68 | 300 | 546 | 1020 | 4463 | 8s |
| swim | 9 | 25 | 66 | 79 | 777 | 396s |

**Table 6: Slowdowns of tracing and simulation compared to uninstrumented runtime at various sampling rates**

# 6. CASE STUDY

The following example, albeit hand-picked for illustration, highlights the dismal performance of a TLB when the size of the data structure is in the neighborhood of a boundary condition. The code is the standard SPEC2000 benchmark, *swim*, whose structure is shown in Figure 6. The size of the data structure is set by the parameter, $N$, used in the *common-block* in Figure 6. The TLB behavior is adversely affected when the size changes from $N = 1023$ to $N = 1024$.

Using the estimates for the memory subsystem of the IBM Power4 machine, we perform the simulation. The memtimes are computed separately for each major data structure (*u,v,p, uold, vold, pold, etc.*) accessed from each of the major subroutines (*CALC1, CALC2, CALC3, CALC3Z*). In Figure 8, each point on the x-axis has a label of the form *arrayName@functionName* indicating the selected array name and function name. For each selection, two bars are shown: one for N=1023 and the other for N=1024. The length of the bar is the memtime calculated for the accesses made to that array in that function. This shows remarkable increase in memtime for N=1024 (about 40 times higher).

```
      PROGRAM swim
      IMPLICIT INTEGER  (I-N)
      IMPLICIT REAL*8   (A-H, O-Z)

      COMMON  U(N,N), V(N,N), P(N,N),
     *        UNEW(N,N), VNEW(N,N),
     *        PNEW(N,N), UOLD(N,N),
     *        VOLD(N,N), POLD(N,N),
     *        CU(N,N), CV(N,N),
     *        Z(N,N), H(N,N), PSI(N,N)

      CALL INITAL

      NCYCLE = 0

   90 NCYCLE = NCYCLE + 1
      CALL CALC1
      CALL CALC2

      IF(NCYCLE .GE. ITMAX) THEN
         STOP
      ENDIF

      IF(NCYCLE .LE. 1) THEN
         CALL CALC3Z
      ELSE
         CALL CALC3
      ENDIF

      GO TO 90
      END
```

**Figure 6: SWIM Main Program**

To probe further into this disparity, we break down the memtimes into components spent in each of the caches and TLB. Figure 9 shows the breakdowns for each value of $N$ separately. Each bar is a stack of time slices estimated to be spent in L1,L2,L3, Memory and TLB. While on the left (N=1023) the major portion is spent in memory access, the right side (N=1024) shows that the major portion of the time is spent in TLB miss latency. When the length of a matrix column used in the inner loop exceeds the page size, it needs a translation entry for an additional page and the finite size of the TLB causes constant eviction of the entries resulting in thrashing.

Figure 10 illustrates the same analysis using sampled traces. Each point on the x-axis corresponds to a selected data structure and function and shows 6 bars. Each bar corresponds to different percentages of sampling, the black being the full trace. In most of the cases, shorter traces project performance close to the full trace. A few selection points show poor prediction when the sampling rate is very small.

## 7. RELATED WORK

The technique of binary-modification for execution-driven simulators[6, 15, 10, 7] forms the basis for SIGMA. All these simulators were concerned with timing characteristics of program segments. They trapped memory access instructions and transferred control to a backend that can simulate desired memory architecture. Non-memory access instructions were run on the native processors and efficient techniques of estimating their timing were incorporated. Unlike these simulators, our objective is not timing characteristics, but rather tying memory accesses to symbolic data structures and subroutines in the source program.

Some tools provide a sampling interface via hardware counters. On Intel platforms, Vtune[8] is available. PAPI[1] provides a multi-platform interface. However, the drawback to using these (and any other hardware counter-based interface) is that the granularity is limited to sampling across code regions. Our work in CacheScope[3] provides data centric information related to each level in the cache. Our approach, using SIGMA, captures the actual memory address being accessed, allowing for statistics on a per-data access basis.

Martonosi et al.[9] present results of cache simulations on sampled traces, where the samples are collected blindly - that is, N samples of length L (distributed uniformly over the whole trace) are selected and N and L are varied. They report cache simulation results with less than 0.5% error compared to the results from the whole trace. In contrast our technique allows us to select more meaningful samples from the source program that correspond to symbolic steps. To show this, we modified a version of Sigma to emulate the behavior of the Martonosi technique. In Figure 7, we compare the results of running our sampling at 10.0% ('Timestep-aware') and the Martonosi sampling with N= 20 and L= $1/10^{th}$ the size of a single interval ('Timestep-oblivious') against a full Sigma trace ('Full'). The size of 10% was chosen because Martonosi asserts that this is their ideal sampling rate. The results show that our timestep-aware sampling technique produced increased accuracy (the colored bars are much closer in length than the white bar). This is also borne out quantitatively in terms of correlations, where timestep-aware sampling results in $r = 0.999985$ compared to $r = 0.756358$ for timestep-oblivious sampling. For this experiment, the running time to generate the traces was 50 minutes for both sampling techniques. As stated previously, a lower sampling rate of 2.5% results in lower overhead and no significant loss in accuracy.

A framework similar to dynSIGMA is described by Snavely et al.[13, 14]. In their work, they too have based their instrumentation on Dyninst. Their sampling technique defines sample boundaries in terms of CPU cycles, similar to the procedures employed by Martonosi et al. Additionally, calculations of memory subsytem statistics (e.g. L1/L2 hit ratios) are performed at the basic block level by the tracer proper, whereas dynSIGMA's simulator describes the performance of data objects at the procedure level.

## 8. CONCLUSIONS

In this paper we presented a technique to use dynamic sampling of trace snippets throughout an application's execution. The technique can be used to dramatically reduce the execution and storage overheads associated with a traditional full memory trace. Additionally, we provided a framework for identifying boundaries of the outermost work loop, and a method of altering sampling rates and algorithms without recompilation of the target application.

We demonstrated that our approach improves accuracy compared to sampling a few timesteps at the beginning of execution by judiciously choosing both the frequency as well as the points in the control flow at which samples are collected. We showed that, while taking a single sample is usually not sufficient, in general only a fraction of the total execution

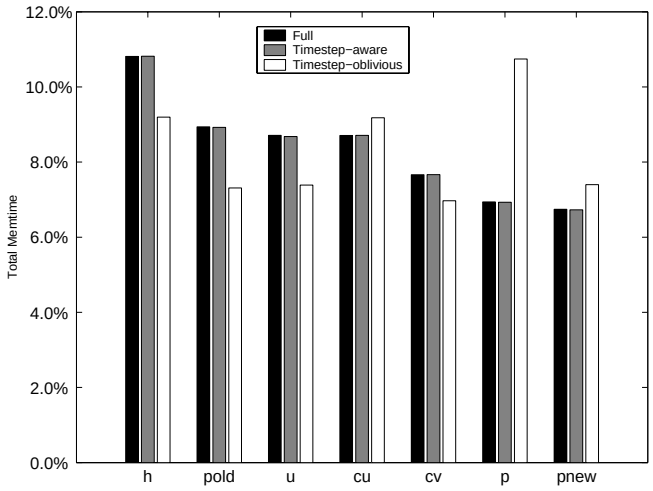


**Figure 7: Comparison of random sampling techniques, seis, N=1024**

needs to be monitored to provide a close representation of the application's performance characteristics.

Our validation results using applications from three benchmark suites demonstrate that periodic sampling gives results that are comparable to full traces, with many above 0.99 linear correlation accuracy. We showed that an application's control flow need not be the same for every iteration of the work loop, as in the case of the seis_s benchmark. Lastly, we presented a case study highlighting the capabilities of the SIGMA simulator and the need for data- and function-centric analysis.

# 9. REFERENCES

[1] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, November 2002.

[2] B. R. Buck and J. K. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 1994.

[3] B. R. Buck and J. K. Hollingsworth. Data centric cache measurement on the intel itanium 2 processor. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, November 2004.

[4] L. Derose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, November 2002.

[5] L. DeRose, K. Ekanadham, and S. Sbaraglia. An approach for symbolic mapping of memory references. In *Proceedings of EuroPar '04*, September 2004.

[6] S. Herrod. Tango lite: A multiprocessor simulation environment. Technical report, Stanford University, Computer Systems Laboratory, 1993. http://citeseer.ist.psu.edu/herrod93tango.html.

[7] Augmint6k: The augmint multiprocessor simulation toolkit for ibm powerpc architecture. IBM Internal Report, 1998.

[8] Intel corporation. http://developer.intel.com/software/products/vtune/index.htm.

[9] M. Martonosi, A. Gupta, and T. E. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 248–259, May 1993.

[10] A. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. In *Proceedings of the International Conference on Computer Design*, 1996.

[11] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge (UK) and New York, 2nd edition, 1992.

[12] S. Sbaraglia, K. Ekanadham, S. Crea, and S. Seelam. psigma: An infrastructure for parallel application performance analysis using symbolic specifications. In *Proceedings of EWOMP '04*, October 2004.

[13] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, November 2002.

[14] A. Snavely, X. Gao, C. Lee, N. Wolter, J. Labarta, J. Gimenez, and P. Jones. Performance modeling of hpc applications. In *Proceedings of Parallel Computing '03*, 2003.

[15] J. Veenstra and R. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 207–207, January-February 1994.

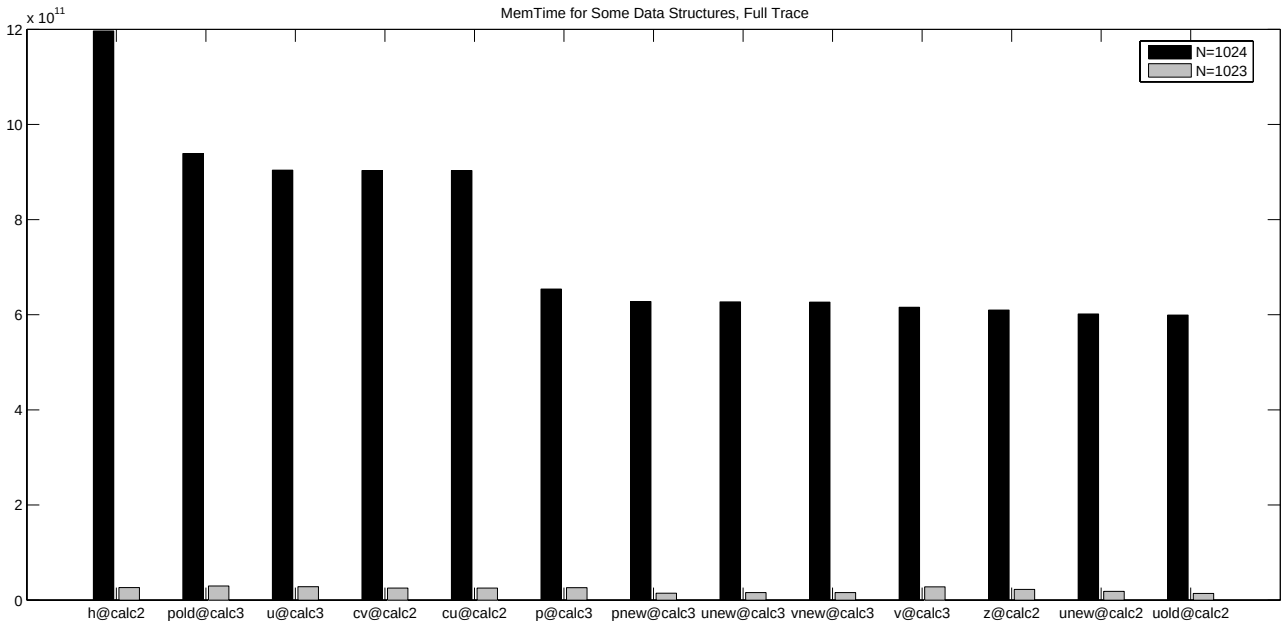


**Figure 8: Memory time in SWIM for $N = 1024$ and $N = 1023$**

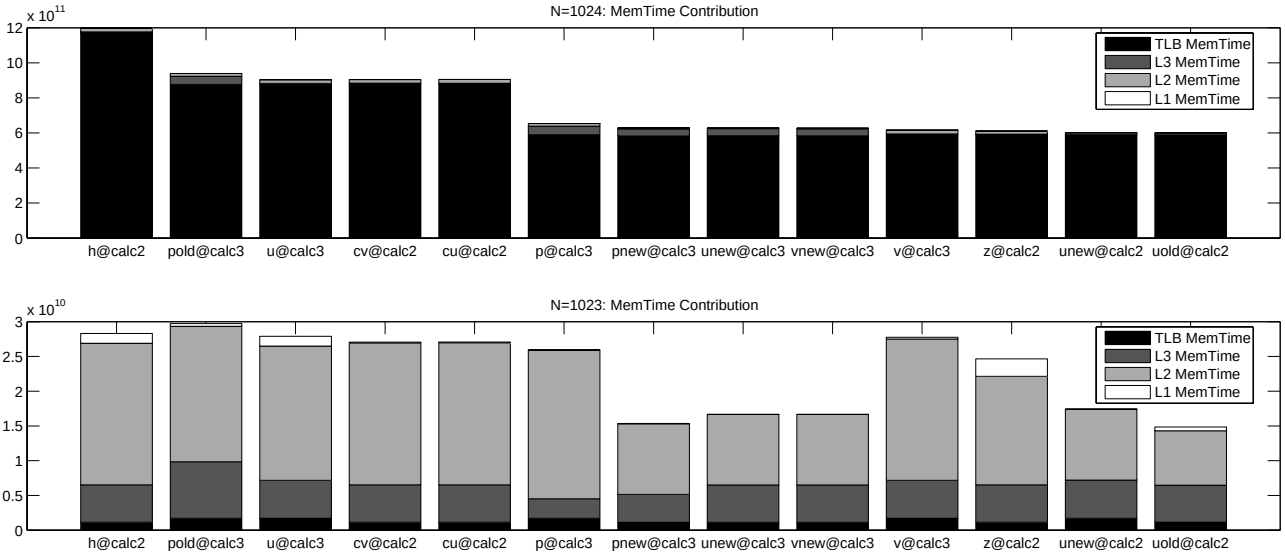


**Figure 9: Contributions to the memtime in SWIM for $N = 1024$ (above) and $N = 1023$ (below)**

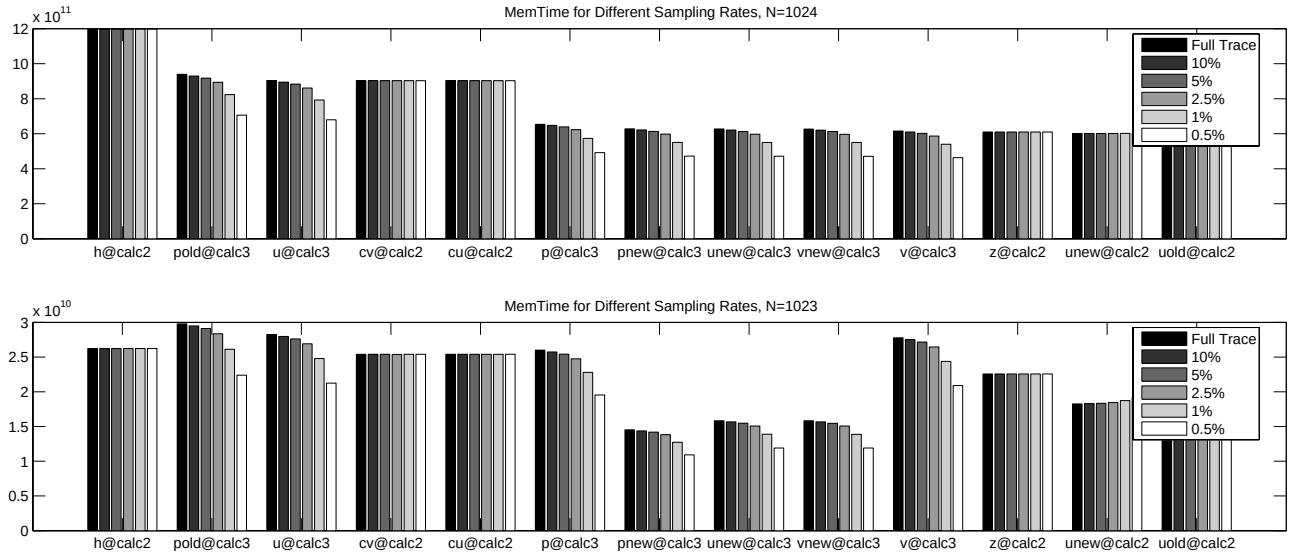


**Figure 10: Comparison between different sampling rates in SWIM for $N = 1024$ (above) and $N = 1023$ (below)**