Reducing Overheads for Acquiring Dynamic Memory Traces

Xiaofeng Gao, Michael Laurenzano, Beth Simon, Allan Snavely
San Diego Supercomputer Center
{xgao, michaell,bsimon,allans}@sdsc.edu

Abstract

Tools for acquiring dynamic memory address information for large scale applications are important for performance modeling, optimization, and for trace-driven simulation. However, straightforward use of binary instrumentation tools for such a fine-grained task as address tracing can cause astonishing slowdown in application run time. For example, in a large scale FY05 collaboration with the Department of Defense High Performance Computing Modernization Office (HPCMO), over 1 million processor hours were expended in order to gather address information on 7 parallel applications. In this work, we discuss in detail the issues surrounding the performance of memory address acquisition using low-level binary instrumentation tracing. We present three techniques and optimizations to improve performance: 1) SimPoint-guided sampling, 2) instrumentation tool routine optimization, and 3) reduction of instrumentation points through static application analysis. The use of these three techniques together reduces instrumented application slowdown by an order of magnitude. The techniques are generally applicable and have been deployed in the MetaSim tracer thereby enabling memory address acquisition for real-sized applications. We expect the optimizations reported here will reduce the HPCMO effort by approximately 80% in FY06.

1 Introduction

Tools for acquiring dynamic memory address information are important for enabling performance research. One needs to know the memory addresses accessed by a program in order to understand its performance on today's deep memory hierarchy machines. Indeed memory operation execution times vary more than do other kinds of operations (such as floating point operations); the variation can span several orders of magnitude from a single CPU cycle when operands fall in L1 cache to several thousand cycles when operands fall in shared main memory. Therefore, the potentially high cost of some memory operations has come to dominate program execution time to the point where, if one does not know the memory reference patterns of a program, it may be impossible to predict how it will perform. On the other hand, with memory reference information (and very little else) one may predict and understand cross-platform performance differences of many applications within about 80% accuracy [10]. Historically though, tools for this purpose are crude, not available for parallel applications, not portable, and/or are much too slow and cumbersome for use in studying the memory behavior of full-scale parallel high performance computing (HPC) applications.

In previous work we created MetaSim [27, 9], a high-level tool for collecting memory address information via the low-level instrumentation APIs ATOM [28], PIN [19], and Dyninst [6] and thus available on most HPC platforms. MetaSim has sophisticated capability to capture and analyze memory address information; it is available for parallel applications, it is portable (to any machine supporting one of the standard APIs), but prior to this work it was so slow that memory performance studies of large applications were cumbersome. Nevertheless, because the information it can acquire is so valuable for reasoning about performance and memory subsystem design, MetaSim is in widespread use at DOE, DOD, and NSF labs.

The HPCMO FY05 effort burned over 1 million processor hours on supercomputers at ARL [2],ARSC [4], ERDC [14], NAVO [20], the Pittsburgh Supercomputer Center [21], and the San Diego Supercomputer Center [23]. The result was deep insight gained into several strategic applications and related supercomputer performance [10, 11]; however a wider community interested in memory performance may not even be able to gain access to this many supercomputer hours which roughly translates to a dedicated 500 processor system devoted only to tracing 24 hours a day 7 days a week for a year. The level of effort, nevermind the huge amount of HPC resources consumed, is beyond the regime that most users could endure no matter how useful the resulting data might be.

This paper is focused on understanding, measuring, and ameliorating the causes of slowdown for methods to acquire dynamic address information at runtime, specifically investigating issues with instrumentation-based approaches. We identify several fundamental causes of slowdown that are generally true of any technique that inserts instrumentation into code for the purposes of collecting dynamic memory addresses. We measure the overheads of these for three common APIs available for the purpose (ATOM, PIN, and Dyninst) that together span most modern HPC systems. We develop several techniques of general applicability for reducing these overheads, and then deploy them on top of the APIs. The specific result reported on is a vastly performance-improved MetaSim with which it is now feasible for ordinary users of HPC system to memory-trace their applications. Additionally, power users such as HPCMO will save approximately 80% of their previous tracing costs.

2 Fundamental Challenges in Collecting Memory Addresses

Considering that processors can issue millions of memory references in seconds, it is not trivial to collect, store and process memory traces for any large parallel application. There are in fact two fundamental problems to consider when developing tools for studying application address traces:

1. Space to store the addresses: A trace file of all the addresses touched by even a short running program can be huge. A full trace of serial CG.A [5], a mere benchmark that completes in 8 seconds on an outdated Alpha, requires more than 43 Gigabytes of disk storage (storing effective addresses and minimal other control data). This implies that writing out an address trace file from all the processors of a long-running parallel application is impractical. Various compression techniques have been proposed to reduce the size of traces [13]. However irregular memory access patterns cause significant problems for these compression-based schemes. In addition, later processing of the traces from a file, compressed or not, is also very slow due to the slow speed of I/O operations.

2. Time to acquire the addresses: Many techniques for gathering address information involve instrumentation of the running code. Instrumented codes run significantly slower than the original non-instrumented application. As a simple example, using the ATOM toolkit [28] to naively instrument and write out each address of a program can slow down application execution more that 400 fold (requiring more than 3000 seconds to collect and dump a trace of serial CG.A). If one only collects the trace (but does not write it to a file) slowdown is still more than 30 fold. This means that for real applications that might run for several hours on hundreds of processors, it would be exceedingly expensive to collect a full memory address trace – even if one had infinite disk storage.

We have abandoned attempts to store memory address traces. In the next section we describe the basic implementation of Metasim, which collects memory trace information and computes statistics on the address stream on-the-fly without ever storing the raw traces. In sections 4, 5.4, and 6 we outline a range of techniques to address the second fundamental challenge above i.e. acquiring memory traces. In total, we move the costs of memory trace acquisition into the realm of the feasible for large applications.

3 Introduction to MetaSim

We developed MetaSim [26] on top of three different low-level instrumentation APIs, ATOM, Dyninst, and PIN, and have deployed it on several different HPC platforms including ones based on the Alpha, Power3, Power4, and Itanium 2 processors. It is available for download at `www.sdsc.edu/pmac`. Despite some differences in technical details of the instrumentation APIs, the underlying idea and the analysis routines are the same across the APIs and platforms.

MetaSim uses two forms of pre-defined probes which are automatically inserted into the instruction stream. One probe is inserted after each memory instruction to capture effective addresses and perform analysis detecting memory access patterns and simulating several cache configurations. Additionally, a probe is inserted at the end of each basic block to capture information at the basic block level and to control sampling state. When the instrumented application completes, the analysis routines generate concise memory access pattern reports for each processor. Such reports can be then used in our performance prediction framework [27] or as a guide to tuning [8]. Alternatively users can modify the instrumentation points and analysis routines as they wish, customizing them for their own purposes. For example, one could specify that only loads and stores in certain functions be instrumented, or user defined memory trace-driven simulators could be inserted instead of, or in addition to, our predefined ones.

In the following work, it is important to realize that the analysis code snippet which gathers effective addresses will be visited much more frequently than the snippet at the end of each basic block. On Alphas for example, the basic blocks for major loops normally are unrolled 8 to 16 times. It is not uncommon to see more than 100 memory instructions in one basic block. Any reduction in the overhead caused by effective address monitoring will have a significant performance impact.

3.1 Comparing Binary Instrumentation APIs

MetaSim is implemented on top of three well known instrumentation APIs for portability. Each API instruments the binary codes with a different process and has different costs. In Table 1 we show a comparison of the overhead exhibited by each tool on our dynamic memory tracer from the HPCMO effort in FY05. One of the APIs, Dyninst, has such high overheads that even tracing benchmarks rapidly becomes prohibitive so only representative results are run and reported.

ATOM [28] is a binary rewriting tool available on Alpha processors. It creates a new binary from the original binary guided by user defined instrumentation and analysis routines. ATOM has the most flexibility of these APIs in placing analysis routines since it is creating a new binary on disk. It also has the ability to inline instrumentation snippets into the original code and performs certain other performance optimizations. Overall ATOM has the lowest overhead of all three instrumentation APIs. However instrumentation is done pre-execution so the user has to specify a priori all the places in the program where interesting events might happen. In addition, the analysis snippets are integrated into the new binary; there is no way to remove them during a run. These snippets will cause overhead even if the analysis portion of the snippets are occasionally bypassed (e.g. for sampled data collection).

PIN is a JIT (just in time) instrumentation API available for Intel processors (IA-32 and IA-64). PIN utilizes a code cache to instrument and keep the instrumented code sequence in memory. Within the code cache, PIN can perform limited optimization to reduce overhead. Similarly to ATOM, once instrumentation is inserted it cannot be removed.

DyninstAPI is a cross-platform API available on most common platforms including those supported by PIN and ATOM. It is based on the idea of dynamic patching. It is the most flexible in handling analysis snippets. The user can program instrumentation in such a way that snippets can be inserted and removed dynamically during the application's runtime. Ideally with such ability, sampling of data collection could be implemented very efficiently by running an instrumented binary when data collection is turned off and only paying the overhead of instrumentation when sampling is turned on. However this flexibility comes with other overheads. DyninstAPI patches the program after it has been loaded into memory. It does not have much freedom in placing the analysis snippets in the original code space of the host application. In most cases, all the snippets reside in a separate code space. The snippets are installed by FAR JUMP instructions in the program that then point to a base trap. Then from the base trap, there is a JUMP to the entry of the corresponding snippet. The cost of these two jumps is considerably heavier than the cost of calling the snippets with the other two APIs. As shown in Table 1 this overhead renders instruction level instrumentation with Dyninst impractical for all but small benchmarks.

Application	ATOM	PIN	DyninstAPI
CG.A.4	98.82X	222.67X	896.86X
FT.A.4	44.22X	127.64X	1054.70X
MG.A.4	107.69X	168.61X	989.53X
LU.A.4	80.72X	153.46X	>>301.4X
SP.A.4	67.56X	93.04X	>>203.66X
CG.B.8	26.96X	163.05X	
FT.B.8	27.48X	88.18X	
MG.B.8	97.18X	161.01X	
CG.B.16	25.49X	131.22X	
FT.B.16	14.93X	99.50X	
MG.B.16	101.18X	107.75X	
CG.C.8	21.68X	225.06X	
FT.C.8	50.42X	96.87X	
MG.C.8	55.81X	112.52X	
CG.C.16	15.02X	107.77X	
FT.C.16	11.59X	84.87X	
MG.C.16	34.67X	126.61X	

Table 1: Sample Slowdowns of Different Tools:
Results from TI05 Baseline MetaSimSlowdown reported over uninstrumented runtime

4 Sampling

Sampling is standard technique useful for gathering representative results from very large sets of data. We previously reported on the performance effects and accuracy tradeoffs of two major types of sampling [9]: 1) sampling in time (only gathering data at certain points in the application's execution) and 2) sampling in space (gathering data for certain static parts of an application at a time). In the realm of time sampling we looked at regular periodic interval-based sampling. Here we summarize our work on SimPoint guided sampling [18] – which uses a preliminary analysis step to select short, representative, application sections to sample.

4.1 Sampling with SimPoint

Regular periodic sampling can produce representative results when an appropriate sampling period is chosen [9]. SimPoint [24] is a tool which uses low-cost dynamic analysis and an off-line phase classification algorithm to detect phases in program behavior that, together with associated weights, are representative of the entire program behavior. This process allows one to selectively sample as little as one percent or less of dynamic instructions, yet have confidence that various stages of runtime behavior are being captured to enable accurate interpolation.

We implemented SimPoint in ATOM, and modified our memory address tracer to collect and simulate addresses at intervals determined by the SimPoint algorithm. We compared the cache hit rates simulated from SimPoint traces to full application traces and traces generated by regular periodic 1% and 10% sampling methods. We found that, in on average, SimPoint traces were as accurate as 10% regular periodic traces, but in general, to achieve results as accurate as 10% random sampling, SimPoint would only sample between 0.18% and and 1.5% of all dynamic memory instructions (the same order of magnitude as 1% regular periodic sampling). Thus, via sampling with SimPoint, we were able to process only about 1% of all dynamic addresses while still very accurately representing the behavior of all dynamic addresses via interpolation.

5 Cost of Binary Instrumentation

Even after employing sampling techniques which process only around 1% or 2% of all dynamic memory instructions without unduly sacrificing the accuracy of interpolated results, execution time of instrumented code to gather memory address data from large parallel applications is prohibitive.

There are several key factors that cause an instrumented binary application to run longer than the original application.

1. Instrumented code executes more instructions than does the original binary (i.e. the instructions in analysis snippets in addition to the original instructions)

2. Jumping to an analysis snippet is a control flow interruption.

3. Program state has to be saved when jumping to analysis snippets.

4. Analysis snippets pollute cache from the standpoint of the original program.

5. Different amounts of processing in analysis snippets on parallel codes becomes a source of load-imbalance.

Factor 1 is an inherent artifact of gathering data using software-based techniques. The slowdown from factor 2 is related to the frequency and availability of data that one wants to gather and can be sometimes optimized as discussed in section 6. Factors 3 and 4 are amenable to optimization and this is investigated in section 5.4. Factor 5 has not been observed as a problem in current efforts, but is a topic for future work.

Next we investigate the basic costs of binary instrumentation required to gather memory trace information. For the results in the following subsections Table 2 shows the experimental setup of each system. The slowdowns presented in this section are compared to the execution times of uninstrumented applications.

5.1 Measuring Overhead: Control Flow Interruption

Modern processors rely heavily on pipelines to achieve high instruction throughput. Once these pipelines bubble or break, performance deteriorates very quickly. When analysis snippets are inserted after every memory instruction in an otherwise well-scheduled binary, the practical effect is to interrupt the pipeline every three or four instructions – a varying but usually large source of overhead depending on the architecture of the machine.

In order to examine the cost of interruption of control flow, we exhibit Experiment 0 – which does nothing but insert a dummy analysis function after each memory reference. This dummy function is passed no parameters and merely returns. Thus we insert the fewest additional instructions possible for address acquisition into the original application, and induce no state-saving overhead (at least theoretically).

The Experiment 0 column of Table 3, Table 4, and Table 5 show the impact of interrupting control flow for every memory reference in a set of benchmarks. Slowdowns of around 10-fold are observed for both PIN and ATOM. The slowdown for Dyninst is significantly higher – due to a more onerous process for switching to analysis snippets as described above.

This experiment shows that the best one could hope to achieve in capturing all memory accesses dynamically is a 10-fold slowdown using these APIs, if every memory instruction is to be instrumented. Achieving the lower bound requires the analysis routine to be passed no parameters and for to it to actually do nothing. In the next subsections we investigate the (worsening) effects of removing these restrictions.

5.2 Measuring Overhead: Maintaining Machine State

Any binary instrumentation tool must maintain correctness of the host application during an instrumented run. Thus, instrumentation tools normally wrap analysis snippets with safeguards when they are inserted into the host binary. These safeguards are responsible for saving and restoring the machine state that could be changed by inserted code snippets. Such overhead is frequent for fined-grained instrumentation (such as for each memory instruction) and can be quite expensive per instance, especially if the processor has many registers. Theoretically it is sufficient to only save the registers that could be changed by the inserted analysis code, but APIs differ in their ability to optimize register saves.

These APIs, almost uniformly and except in trivial cases, simply save all the registers that could be touched in the snippets whether the snippets actually change them or not. Because of this, for the users of instrumentation tools, it is important to make the most visited snippets as simple as possible so fewer registers are required to be saved and restored. As an example, a call to a complicated cache simulation analysis snippet in the ATOM API requires saving all 64 registers before the instrumentation point and another 64 loads after. A similar effect is seen with Dyninst on Power 4 architectures, but an increased overhead is seen with PIN on Itanium architectures, due to larger register files.

We explore the minimal impact that state saving can have on a memory address tracing tool with Experiment 1. In this experiment, a one line analysis snippet is called after each memory access which records the effective address in a global array. A second routine is called at the end of each basic block to clear the global array. Effectively all addresses are recorded and then discarded shortly thereafter. This is a minimal overhead representative of recording memory addresses. The costs are shown in Tables 3, 4, and 5. It can be seen that using ATOM or PIN, the overhead caused by register saves and restores increases slowdown dramatically over Experiment 0 – ranging from 22 to 70 fold. Interestingly the overhead when using Dyninst did not increase much; Dyninst saves and restores all registers for all analysis snippets, no matter how simple.

5.3 Measuring Overhead: Cache Interference

Because the execution of analysis snippets and original application is interleaved, there could be contention between them in both the instruction and data cache. Normally this is minimized if the analysis snippets are very simple or if cache-friendly applications are being profiled. However, HPC applications of interest already stress the memory hierarchy, while the code to perform cache simulations in analysis snippets accesses large data structures and has significantly complicated control flow. Consequently some of the instructions and data belonging to the host application may be evicted. These victims will have to be loaded into caches when the host application resumes execution. If this happens frequently enough, the performance of the instrumented code will drop dramatically.

We present Experiment 2, which shows the impact on runtime using full analysis code to collect effective addresses and run each through a set of 26 cache simulators. We were only able to complete 2 runs due to the incredible slowdown of this process. Note that sampling and a maximum visit to any given memory address was used in this experiment. That is, only when in a "sampling on" period would the memory access analysis actually run the cache simulators. Additionally, after a given memory access had been simulated 15,000 times, it was deemed no longer eligible for simulation.

Finally, it is important to note that the slowdown reported in Experiment 2 is not entirely due to cache interference. This experiment increases the number of instructions executed in the analysis snippets significantly (by running the cache simulators). However, it is this execution of extra instructions that is inherently tied to the increase in cache interference (it is hard to decouple the two effects).

Machine	Processor	Tool version	Compiler and Flags
Lemieux	Alpha EV67, 1GHz	ATOM V3.25	mpif77 -g3 -O4
Cheetah	Power4, 1.33GHz	Dyninst 4.0	mpxlf -g -O5 -qstrict -qarch=ppwr4 -qtune =pwr4
Teragrid	Itanium2, 1.3GHz	PIN 1.71	mpif90-g -O3

Table 2: System Setup

App.	w/o Instru.	Experiment 0	Experiment 1	Experiment 2
CG.A.4	1.7	24.2 (14.2)	91.0 (53.5)	4857 (2857)
FT.A.4	9.0	89.0 (9.9)	316.1 (35.1)	12314 (1368)
MG.A.4	5.2	56.8 (10.9)	211.2 (40.6)	
LU.A.4	47.0	731 (15.5)	2494 (53.0)	
SP.A.4	84.0	769 (9.2)	2637 (31.4)	

Table 3: Instrumentation Overhead using ATOM: execution time in seconds, slowdown in parenthesis

Application	w/o Instrument	Experiment 0	Experiment 1
CG.A.4	1.3	22.2 (17.1)	112.6 (86.6)
FT.A.4	5.4	61.2 (11.3)	279.7 (51.8)
MG.A.4	2.4	28.0 (11.7)	138.0 (57.5)
LU.A.4	27.8	406.8 (14.6)	1848.0 (66.5)
SP.A.4	55.9	602.8 (10.8)	2664.0 (47.7)

Table 4: Instrumentation Overhead using PIN: execution time in seconds, slowdown in parenthesis

Application	w/o Instrument	Experiment 0	Experiment 1
CG.A.4	2.2	1920.0 (872.7)	1933.9 (878.6)
FT.A.4	7.2	7302.3 (1000.3)	7327.0 (1003.7)
MG.A.4	5.1	4716.9 (924.7)	4756.5 (932.5)
LU.A.4	65.4	>19720 (301.4)	>19720 (301.4)
SP.A.4	96.8	>19720 (203.7)	>19720 (203.7)

Table 5: Instrumentation Overhead using DyninstAPI: execution time in seconds, slowdown in parenthesis

5.4 Tool-based Optimization: Buffering

Factors 3 and 4 of overhead can be improved in tandem with a single simple technique – buffering of memory addresses before processing them. This effectively reduces the impact of state saving overhead by having the simplest possible analysis snippet for each memory access.

We implemented a memory access probe which, in a single line of C code, stores the effective address of a memory access in a large global buffer. We then delay the more complex cache analysis until such time as the buffer is full. In the less frequently invoked basic block level analysis snippet (which controls sampling among other things) we monitor the effective address buffer and run the cache simulators when the buffer is full.

With the much simplified per memory access snippet, the overhead of state saving for the most frequent analysis snippet has been greatly reduced. Since the cache simulator code is run through the less frequent basic block analysis snippet, we see a decrease in state saving overhead proportional to the number of static memory accesses per basic block.

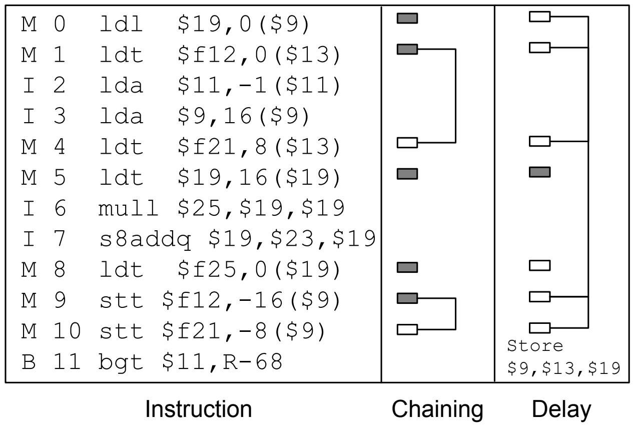
Additionally, this approach ameliorates cache interference. The primary offender in this arena is the cache simulation code, which simulates 26 cache structures for each memory access. By buffering a large number of effective addresses and then running the cache simulators for all of those addresses at once we reduce the frequency of interference with the program's cached data. We implemented a buffer of 10,000 addresses, reducing the number of times the original application's cache behavior is impacted significantly. Overall, using ATOM, we find buffering to speedup unbuffered tracing by as much as 30 fold.

6 Static Application Analysis to Reduce Instrumentation Points

Though buffering improves factors 3 and 4 of binary instrumentation, we have not yet addressed the issue of control flow interruption. As illustrated in Experiment 0, simply having a control flow interruption for every memory access causes a slowdown of around 10 fold – and that is without actually storing or analyzing any data! In order to break this "limit" we must reduce the number of times we actually store effective address information, while maintaining accurate address trace information. In this section, we describe two methods to significantly reduce the number of instrumentation points required to generate a full memory address trace. This is possible by performing some static analysis prior to tracing and using that information to selectively choose instrumentation points from which it is possible to derive complete memory access behavior.

6.1 Chaining Memory Instructions

With simple static analysis of the binary code in each basic block, we chain memory instructions together if they will always keep the same static offset during run-time. Only one of the memory instructions in a chain (the leader) must be instrumented. The others' effective addresses can be calculated from the leader's address.



```
M 0  ldl  $19,0($9)
M 1  ldt  $f12,0($13)
I 2  lda  $11,-1($11)
I 3  lda  $9,16($9)
M 4  ldt  $f21,8($13)
M 5  ldt  $19,16($19)
I 6  mull $25,$19,$19
I 7  s8addq $19,$23,$19
M 8  ldt  $f25,0($19)
M 9  stt $f12,-16($9)
M 10 stt $f21,-8($9)
B 11 bgt $11,R-68
```

Figure 1: Reducing Instrumentation Points with Static Analysis
Instructions labeled M are memory, I integer, B branch
Dark box indicates the memory instruction is instrumented

Consider a basic block code segment shown in Figure 1 where there are 7 memory instructions: 0,1,4,5,8,9, and 10. Note that lda on Alpha is not a memory instruction (rather it is like an addi). Until execution, we do not know the value of the registers so there is no way to tell the exact effective address of these memory instructions. But we can tell from static analysis that the effective address of instruction 1 will always be 8 less than the effective address of instruction 4. The effective address of instruction 9 will always be 8 less than that of instruction 10. Using this knowledge, we do not need to instrument after instruction 4 and instruction 10 if instructions 0 and 9 have already been instrumented. However, we are not as successful in chaining instruction 0 to 9 and 10 due to the modification of the shared register $9 in instruction 3.

6.2 Delayed Instrumentation with Symbolic Execution

Chaining memory instructions can reduce the number of instrumentation points, but there is still one instrumentation point per chain. These instrumentation points leave a number of control flow interruptions (at memory accesses) that impair performance. Additionally, register values which are updated within the basic block break chains and create additional instrumentation points. The goal of delayed instrumentation is to find a way to merge all memory address collection into a single instrumentation call at the end of the basic block.

The key is to record register values rather than the effective addresses themselves. These register values can then be used in conjunction with static knowledge of the basic block binary, such as immediate offsets, to enable us to re-create the effective addresses of each memory instruction in the basic block. Straightforward delayed instrumentation would, in the case of the code in Figure 1, allow us to remove instrumentation points after instructions 1, 8 and 9 and instead use a single instrumentation call at the end of the basic block to save the values of register $13, $19 and $9. The effective addresses of these memory instructions can be properly calculated from the recorded corresponding register values. In reality, a per basic block analysis routine already exists, we simply increase the number of parameters passed.

Removing instrumentation after instruction 0 requires a bit more effort. The recorded value of register $9 isn't the exact value when instruction 0 executes. Register $9 is modified by a lda instruction in the middle of the basic block (which originally broke $9's references into two chains). We notice that lda is making a simple modification to register $9 and the effect can be reversed to deduce the correct value for instruction 0.

We define a set of tractable modifications to registers which we can easily reverse the effect of and calculate earlier register values based on a final end-of-basic block register file snapshot. Table 6 lists the four integer instructions we support as well as the symbolic execution that updates register values, Any other instructions that assign to an integer register are defined as intractable. If and only if a register used for addressing is modified by an intractable instruction in the middle of the basic block, we insert a snippet to store its value before modification. We cannot remove the instrumentation point after instruction 5 because register $19 has been changed intractably. For the block in Figure 1, we only need to explicitly instrument one memory instruction, the rest are captured at the end of the basic block.

Instruction	Example	Symbolic Execution
lda	lda $target$, imm($base$)	$target$ = $base$ + imm
ldah	ldah $target$, imm($base$)	$target$ = $base$ + imm * 65536
addli, addqi	addi $target$, $base$, imm	$target$ = $base$ + imm
subli, subqi	subq $target$, $base$, imm	$target$ = $base$ - imm

Table 6: Instructions Supported for Symbolic Execution

```
I  1 ldah $25,18($7)
I  3 ldah $27,9($7)
I  4 ldah $8,4($7)
I  8 ldah $24,18($7)
M  13 ldt $f16,25352($8)
M  16 ldt $f15,-29536($24)
M  17 ldl $8, 9($8)
M  21 ldt $f11,25480($27)
M  22 ldt $f14,10648($25)
```

Figure 2: Register Aliasing Supported in Delayed Analysis

Figure 2 shows an excerpt from a very common assembly sequence on Alpha architectures from an important basic block in

the NAS Parallel Benchmark SP. Note that it seems there are four registers which are used to address into memory for load instructions. In reality these registers are serving in the role of temporary variables used to construct a base plus displacement style address where the displacement is larger than 16 bits in size. These are actually 4 accesses to different displacements from the address in register \$7. These temporary registers are frequently assigned in the block. Such assignments make delayed instrumentation less effective. However, based on the fact that the values of these temporary registers are calculated from register \$7, the effective addresses of memory instructions using these temporary registers can also be calculated using register \$7.
For example, register \$8 is changed in instruction 17, which makes the register value used for addressing by instruction 13 and 17 intractable. One the other hand, the value of register \$8 at instruction 13 equals to the value of register \$7 at instruction 4 plus $65536 * 4$. If \$7 is tractable and can be delayed, the effective addresses of instruction 13 and 17 can be calculated from the end-of-block value of register \$7, plus proper adjustment. Identifying and optimizing this form of register aliasing provides another chance to delay instrumentation. It has been shown to greatly increase the ability to remove instrumentation points.

An additional performance benefit is that less data must be buffered when storing register values than when storing effective addresses. This results in reduced numbers of interruptions from the analysis routine which processes the buffer.

6.3 Related Issues

Buffering has been proven to be very effective in improving trace performance. Generally the bigger the buffer, the better the performance. However for parallel applications, instrumentation can cause extra overhead due to synchronization. When the buffer is too big, each processor takes much longer in the analysis routines – sometimes it causes busy waiting of other processors. Currently our buffer is set to 10,000 entries and such busy waiting is negligible.

We anticipate that each of these techniques must re-analyzed when considering porting them to other binary instrumentation frameworks. In particular, the chaining approach provides no improvement for handling register-register addressing, which is commonly used for indirect memory references on Power processors because the distances of effective addresses of such instructions cannot be determined prior to execution.

7 Experimental Results

We implemented the static analysis algorithms, integrated chaining and delayed-style instrumentation into ATOM-based MetaSim, and measured speed on Lemieux, an Alpha based system, at PSC.

Table 7 shows the number of actual memory address instrumentation points required after using the static analysis and delayed instrumentation techniques. We see less than 10% of total memory instructions in most applications are required to be instrumented. Although fewer instrumentation points generally leads to faster tracing, the actual speed depends on whether the most frequently executed sections of code have significantly fewer instrumentation points. For example, E3D only shows a reduction of 80% in number of instrumentation points after static analysis, but the 12 basic blocks which account for more than 90% of total memory references have had all memory instrumentation points optimized away. All 925 memory instructions in those blocks can be analyzed with delayed instrumentation at the end of their basic blocks. From Table 9, we can see the slowdown of E3D has been reduced to only 4-fold despite the many remaining instrumentation points.

Application	Num. of Mem Inst	Delayed Instrumentation	
		Instrument. Points	Reduction
CG.A.4	3595	141	96.1%
FT.A.4	4302	224	94.8%
MG.A.4	9662	439	95.5%
LU.A.4	20579	233	98.9%
SP.A.4	38383	306	99.2%
HYCOM	87520	3029	96.5%
AVUS	124341	10013	92.0%
E3D	45698	9373	79.5%

Table 7: Reduction in Number of Instrumentation Points with Delayed Instrumentation

Next we assess how well the technique of static analysis and delayed instrumentation impacts our identified costs of control flow interruption (Experiment 0) and costs of saving machine states (Experiment 1). Again, Experiment 0 uses dummy instrumentation functions that do not actually record any data and Experiment 1 actually collects memory addresses, but immediately discards them without running any further processing.

Table 8 shows that the overhead of simply instrumenting to gather necessary effective addresses can be reduced from about a 10 fold slowdown to a range of 1.2 to a 3.5 fold slowdown. Adding in a bit more realism, Experiment 1 (which actually collects effective addresses) shows that for simplest processing of the collected addresses (by simply discarding them), one could achieve slowdowns of less than 6 fold and often around 2-3 fold.

Table 9 compares the execution time of our memory trace code (MetaSim) running 26 cache simulators using ATOM, showing the impact of the two steps of our static analysis optimizations. The buffering column reports the baseline execution time using sampling and a 10,000 element buffer. The chaining column baseline shows that speedup from chaining is very application dependent with CG.A.4 showing only a 1.07 times speedup over buffering while SP.A.4 shows over a 15 times speedup. The effectiveness of chaining is dependent on how many memory references can be chained in the important basic blocks. The results in the delay column show the significant impact of additional static analysis to enable delaying as many memory instrumentation points as possible. Slowdowns over the original uninstrumented code are shown in parenthesis and we can see they are much less

Application	w/o Instru.	Experiment 0				Experiment 1			
				Delayed				Delayed	
CG.A.4	1.7	24.2	(14.2)	5.8	(3.41)	91.0	(53.5)	10.8	(6.35)
FT.A.4	9	89.0	(9.9)	13.8	(1.53)	316.1	(35.1)	19.0	(2.00)
MG.A.4	5.2	56.8	(10.9)	7.5	(1.44)	211.2	(40.6)	14.0	(2.50)
LU.A.4	47	730.7	(15.5)	56.5	(1.20)	2494.0	(53.0)	74.4	(1.58)
SP.A.4	84	769.4	(9.2)	120.7	(1.44)	2637.1	(31.4)	209.1	(2.49)

Table 8: Comparison of Instrumentation Overhead

than simply buffering. The larger slowdowns appear in shorter applications are due to the fact they are too short for our sampling to be effective. Of specific note is that results on full applications such as HYCOM, AVUS, and E3D are particularly good. An additional benefit of this improvement is that these traces can now be collected in a single run and still fit within standard queue limits (typically 12 hours). Previously, the much higher slowdown of tracing required a multi-stage process where many applications were traced in 10 different runs – collecting data on subsets of basic blocks each run.

8 Related Work

Much work on memory system design [29, 25, 33] relies heavily on memory trace-driven simulation. Uhlig and Mudge provide an extensive survey [30] on various aspects of trace-driven simulations. Trace-driven simulations are prone to incomplete, unrepresentative data [17, 16]. Full memory reference trace storage on disk is no longer an option even for small benchmarks. Many sampling techniques have been proposed to reduce the amount of data stored while still keeping the data representative [31, 32]. Work focusing on trace compression based on its internal structure and statistical properties has been done by [13] and [15]. Agarwal et.al [1] introduced an efficient method of capturing address traces using microcode supported by VAX1 8200 processor.

Recent progress in binary instrumentation [28, 19, 6] makes it possible to simulate memory traces without storing them first on disk. Much work on cache-conscious data layout [7, 22], software prefetching [12], and data locality optimization, rely on the ability to instrument the binary to acquire effective address traces. However the slowdown caused by binary instrumentation often means this work is restricted to kernels and very short programs.

Various sampling techniques again are proposed to reduce the quantity of data to process. Among them, SimPoint [24] determines sampling by observing application behaviors and phases. This generally leads to more representative sampling. We adopted SimPoint-guided sampling in our work. However instrumentation causes overhead no matter if sampling is turned on or off. Arnold and Ryder [3] remove the overhead when sampling is off by duplicating the binary, creating one instrumented and one uninstrumented. Sampling is turned on and off by switching between two binaries. This technique is extremely effective and have been applied extensively [13, 12]. In contrast, we reduce the overhead by reducing the number of instrumentation points which has the benefit of also reducing overhead when sampling is on.

9 Conclusion

In this paper, we dissect, analyze, and then optimize the severe causes of slowdown in binary instrumentation-based memory tracers. We show a triad of optimizations including SimPoint-based sampling, instrumentation tool-based optimizations, and optimizations enabled by static binary analysis of the the application to be traced.

Sampling has long been used to enable (hopefully) representative data gathering over large data sets. Here we show the application of SimPoint phase analysis for parallel code and modify the MetaSim tracer to collect data based on SimPoint identified and weighted intervals. We show that the cache simulations gathered from SimPoint sampling are as accurate as 10% random sampling but can be gathered from many fewer dynamic instructions (1%).

Despite greatly reduced sampling requirements, the dominant portion of binary instrumentation overhead is caused by the process itself which, in a naive implementation, places instrumentation around every memory access. We show control flow interruption caused by binary instrumentation will cause at least a 10 fold slowdown – on any of the three tested APIs: ATOM, PIN, or Dyninst. Maintaining machine state around instrumentation points (register saving and parameter passing) causes even more slowdown. These overheads make it painful to trace full applications even when sampling techniques are employed.

We argue that, given the state of current binary instrumentation tools, the most effective way to reduce overhead is to reduce the number of instrumentation points. We introduce static analysis techniques which accomplish this with no change in the acquired memory address trace. We first introduce a technique which can chain memory instructions in a given basic block which share a common base register into a set which requires only one instrumentation point. We then describe a method to generate effective addresses from register values and the defer recording of register values and calculation of addresses until the end of the basic block. Though not all memory addresses can be optimized in this way, very significant reductions are seen in the number of instrumentation points required, and even greater reductions are seen in the overall slowdown required by this delayed information gathering.

App	w/o Instru.	w/o Buffering	w/ Buffering	Chaining	Delayed
CG.A.4	1.7	4858 (2857.4)	168 (98.8)	156 (91.8)	65 (38.2)
FT.A.4	9.0	12315 (1368.3)	398 (44.2)	298 (33.1)	166 (18.4)
MG.A.4	5.2		560 (107.7)	376 (72.3)	156 (30.0)
LU.A.4	47.1		3799 (80.7)	1605 (34.1)	813 (17.3)
SP.A.4	84.0		5676 (67.6)	376 (45.3)	1568 (18.7)
CG.B.8	67.2		1812 (27.0)	840 (12.5)	396 (5.9)
FT.B.8	70.3		1932 (27.5)	509 (7.2)	253 (3.6)
MG.B.8	8.9		864 (97.2)	543 (61.1)	221 (24.9)
CG.C.8	183		3967 (21.7)	1426 (7.8)	743 (4.1)
FT.C.8	189.6		17630 (50.4)	11652 (33.3)	7978 (22.8)
MG.C.8	63.5		3544 (55.8)	1129 (19.4)	480 (7.6)
HYCOM	3506		78855 (22.5)		19633 (5.6)
AVUS	2359		34816 (14.5)		11851 (5.0)
E3D	1569		34058 (21.7)		6388 (4.0)

Table 9: Tracing Time Comparison in seconds (slowdown in parenthesis)

In a final analysis, we show that memory tracing with large-scale cache simulation can be performed using binary instrumentation tools with about a 5 fold slowdown on real applications (shorter running benchmarks show slightly larger slowdowns). Prior tools required very significant time and machine investments with slowdowns ranging from 30 to over 100 for real applications. While our static analysis techniques will vary in impact based on the specific application to be traced, our results on three real applications lead us to be confident that these techniques are very useful on real applications. This significant improvement in tracing slowdowns should reduce the effort required to trace real applications such that the benefits of application tracing should be available to the common user.

10 Acknowledgements

This work was sponsored in part by the Department of Defense High Performance Computing Modernization Program Office through a contract award entitled HPC Benchmarking and the Department of Energy Office of Science through a SciDAC contract award entitled High-End Computer System Performance: Science and Engineering.

References

[1] A. Agarwal, R. L. Sites, and M. Horowitz. Atum: a new technique for capturing address traces using microcode. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 119–127. IEEE Computer Society Press, 1986.

[2] ARL. URL at `http://www.arl.army.mil`.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01*, pages 168–179. ACM Press, 2001.

[4] ARSC. URL at `www.arsc.edu`.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, and et. al. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[6] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[7] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 139–149. ACM Press, 1998.

[8] L. Carrington, X. Gao, N. Wolter, A. Snavely, and R. Campbell, Jr. Performance sensitivity studies for strategic applications. In *UGC '05*, 2005.

[9] L. Carrington, A. Snavely, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. In *ICCS '03: Proceedings of International Conference on Computational Science*, pages 926–935, 2003.

[10] L. Carrington, A. Snavely, M. Laurenzano, R. Campbell, Jr., and L. Davis. How well can simple metrics represent the performance of hpc applications? In *Supercomputing'05 to appear*, 2005.

[11] L. Carrington, N. Wolter, A. Snavely, and C.Lee. Applying an automated framework to produce accurate blind performance prediction of full-scale hpc applications. In *UGC'04*, 2004.

[12] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02*, pages 199–209. ACM Press, 2002.

[13] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis. In *Supercomputing '02*, 2002.

[14] ERDC. URL at `www.erdc.usace.army.mil`.

[15] X. Gao and A. Snavely. Exploiting stability to reduce time-space cost for memory tracing. In *ICCS '03: Proceedings of International Conference on Computational Science*, pages 966–975, 2003.

[16] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *SIGMETRICS '93*, pages 146–157. ACM Press, 1993.

[17] S. F. Kaplan. Collecting whole-system reference traces of multiprogrammed and multithreaded workloads. In *WOSP '04: Proceedings of the fourth international workshop on Software and performance*, pages 228–237. ACM Press, 2004.

[18] M. Laurenzano, B. Simon, X. Gao, and A. Snavely. Low-cost, portable trace-driven memory simulation using simpoint. In *WBIA05 to appear*, 2005.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200, 2005.

[20] NAVO. URL at `www.navo.navy.mil`.

[21] PSC. URL at `www.psc.edu`.

[22] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02*, pages 140–153. ACM Press, 2002.

[23] SDSC. URL at `www.sdsc.edu`.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57. ACM Press, 2002.

[25] N. T. Slingerland and A. J. Smith. Cache performance for multimedia applications. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 204–217. ACM Press, 2001.

[26] A. Snavely, L. Carrington, and N. Wolter. Modeling application performance by convolving machine signatures with application profiles, 2001.

[27] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing '02*, pages 1–17. IEEE Computer Society Press, 2002.

[28] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94*, pages 196–205. ACM Press, 1994.

[29] C. B. Stunkel, B. Janssens, and W. K. Fuchs. Address tracing for parallel machines. *Computer*, 24(1):31–38, 1991.

[30] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[31] X. Vera, N. Bermudo, J. Llosa, and A. Gonzalez. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Trans. Program. Lang. Syst.*, 26(2):263–300, 2004.

[32] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.

[33] H. Zhenchun and L. Sanli. Ipuloc - exploring dynamic program locality with the instruction processing unit for filling memory gap. *J. Comput. Sci. Technol.*, 17(2):172–180, 2002.