

# ALITER: An Asynchronous Lightweight Instrumentation Tool for Event Recording

Xiaofeng Gao, Beth Simon, Allan Snively  
San Diego Supercomputer Center  
{xgao,bsimon,allans}@sdsc.edu

## Abstract

Binary instrumentation tools are very useful for collecting traces of program events. Common uses for such traces include trace-driven simulation and performance modeling. However commonly available general-purpose instrumentation tools are inefficient for capturing fine-grained events as for example a sequence of dynamic memory addresses. We introduce ALITER, an asynchronous lightweight instrumentation tool for event recording which is extremely light in terms of tracing overhead as compared to commonly available binary instrumentation tools. The tool creates a buffer in the instrumented code space and inlines buffer maintenance functions into the instrumented code. User supplied analysis routines are only invoked when the buffer is fairly full. This approach, i.e. having a user code space buffer managed under ALITER's control, ensures that most control transfers between user code and instrumentation code are eliminated. In addition, storing events to the buffer and checking buffer status are implemented very cheaply. Thus traditional sources of tracing overheads are greatly reduced. Overall we report less than a 2-fold slowdown to collect memory traces of the selected benchmarks; this contrasts with tens and even hundreds of fold slowdown using generally available instrumentation tools.

## 1 Introduction

Tools for acquiring dynamic execution events [10] such as memory address information for large scale applications are important for performance modeling [11], optimization [4, 9], and for trace-driven simulation [7, 5]. Much research in architecture design and code optimization is conducted via trace-driven simulation. Thus the ability to collect traces of dynamic execution events, especially for real applications, is important. Many of today's tracers and profilers are built on top of binary instrumentation tool kits such as PIN [8], ATOM [12] and Dyninst [3].

A common practice is to use binary instrumentation tools to insert probe functions at various locations in the to-be-traced code. These probes may simply dump interesting events to disk. Subsequently the collected event trace may be processed, as for example through a trace-driven simulator. However saving fine-grained traces of large-scale applications is very chal-

lenging. For example, the serial version of the NAS Parallel Benchmark [2] CG.A, a mere benchmark that completes in 8 seconds on a 21264 Alpha processor, requires more than 40 Gigabytes of disk storage to save a complete memory trace along with some minimal control data. For large applications that may run for hours on many processors, saving a complete memory trace onto the disk is simply impractical. More practical is to process and compute statistics against event streams on-the-fly during the execution of instrumented code.

However adding analysis routines into the instrumented code causes overhead. And because commonly available binary instrumentation tools are not specifically optimized for very fine-grained tasks such as address gathering, straightforward implementation of such tasks can cause astonishing overheads. For example, collecting memory traces by instrumenting each memory instruction with a very simple user level callback function via ATOM, PIN, or Dyninst causes instrumented code runs that are tens, even hundreds, of times slower than the uninstrumented version. If one is interested in tracing and simulating short-running benchmarks this may be bearable, but studies of full scale applications become impractical with such overheads. Besides the obvious extra time needed to run the user's analysis code, these overheads are mainly from two factors. First, instrumentation tools need to save system state (registers and etc.) every time an instrumentation code snippet is encountered, because the snippet could change that state and thus affect the correctness of the original program. On IBM Power 4 systems for example, more than 40 instructions are commonly needed solely for state saving and restoring. If every memory instruction is to be instrumented, an instrumented code could need to execute at least 40x more memory instructions than the original. For scientific applications which are typically memory-intensive to begin with, the resulting slowdown can be tremendous. Second, the instrumentation code snippets introduce control hazards relative to the original code. Every time control transfers to the inserted code pipelines have to be purged. For fine-grained tracing tasks, such interruption becomes unbearable performance-wise. For example, most scientific applications have one memory instruction every three or four instructions. If each memory instruction is instrumented with a function call, the original program control flow will be interrupted every three or four instructions. Such frequent interruption makes various per-

formance enhancing hardware features that exploit instruction level parallelism useless.

We have demonstrated various methods to reduce overhead by removing non-necessary instrumentation points; for example we group memory operations whose addresses can be deduced by static analysis if just one of the group's dynamic addresses are known, and only instrument one from each group [?]. This approach works fairly well to reduce the number of instrumentation points, the overhead per-instrumentation point remains high.

We observe that for many tracing and processing tasks, there is no need to process the event immediately after it happens. Such tracing is logically independent from processing. We term such a tracing with post-processing scenario asynchronous. Intuitively, if it is logically possible to dump the event trace and then process the trace offline (regardless of whether it is performance-feasible), the tracing and processing are asynchronous. Asynchronous tracing and processing scenarios are very common.

In this paper we introduce ALITER: an asynchronous lightweight instrumentation tool for event recording designed specifically for asynchronous tracing tasks such as trace-driven simulations. The underlying instrumentation scheme specializes in handling tracing tasks which can be programmed asynchronously. Unlike the many generic, synchronized instrumentation tools, this new instrumentation scheme doesn't invoke the user's routine every time an interesting event happens. Instead, it stores the event in a buffer created in the application's space. Execution control only transfers to the user's analysis routines when there are enough events in the buffer. By creating a buffer and inlining all the buffer maintenance codes (besides the benefit that the user does not have to implement the buffering scheme), most of the overhead caused by control transferring can be eliminated. In addition, for most cases only one instruction is needed to save the event to the buffer and there is no need to transfer control to user's routine, thus removing a possible control hazard.

Currently ALITER is implemented as an assembly code rewriter for IBM Power4 systems. We have measured less than 2 fold slowdown on the selected benchmarks to collect memory traces. In the common case, only one instruction is needed for each instrumentation point, plus 20 extra instructions per each basic block, which are responsible for maintaining the status of the buffer. Previously we had tens of fold slowdown using ATOM and PIN and nearly 1000 fold slowdown using Dyninst for accomplishing the same tasks. Future work includes implementing ALITER as a true binary rewriter (thus avoiding some user complications requiring, currently, that one modify makefiles to dump assembly) and porting ALITER to other platforms and ISAs.

The rest of the paper is organized as follows: In Section 2 we briefly cover our previous work and some related research. In Section 3 we classify sources of instrumentation overhead.

In Section 4, we describe how we can maintain a buffer and acquire events with minimal instructions and control transfers. In Section 5 we present experimental results to show that this asynchronous scheme is very efficient for fast profiling and tracing. In Section 6 we discuss some related issues and our future plans.

## 2 Related Work

Atom [12] is a binary rewriting tool for Alpha processors. The binary is parsed and instrumented statically. Among the other instrumentation tools we have used ATOM has the least overhead. PIN [8] is a JIT (just in time) instrumentation tool for Intel processors (IA32 and IA64). The binary is first loaded into program memory and then instrumented in a special area in memory called the code cache before each instruction sequence is executed. Dyninst [3] is a dynamic instrumentation tool kit available on most current platforms. Unlike most other binary instrumentation tools, Dyninst allows user to dynamically instrument and de-instrument the binary. However the cost of this feature is noticeably higher than other binary instrumentation tools. All these binary instrumentation tools are designed to instrument whatever function the user provides and wherever the user requests it. This versatility ensures the tools have a large audience. However the performance isn't as good as one would like, especially for fine-grain instrumentation.

In our previous work [6], we demonstrated how one can reduce instrumentation cost via user-maintained buffers. We also demonstrated how instrumentation points can be dramatically reduced via simple static analysis. The work presented in this paper differs from that in that now it is the instrumentation tool that maintains the buffer and determine instrumentation points. The only thing the user needs to provide is a function which will process the data stored in a given buffer. Besides the benefit of simplifying the tool user's tasks, buffers can be filled and checked much more cheaply without interference from the user. Thus the previous work is focused on how to reduce overheads by reducing the number of instrumentation points; this paper is focused on how to reduce overheads by reducing the cost of each instrumentation point.

## 3 Cost of Binary Instrumentation

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.

Factor 1 is an inherent artifact of gathering data using software-based techniques. This cost is out of the control of the instrumentation tool. The overhead of factors 2 and 3 is related to how many instrumentation points the user wants to insert into the binary. For generic binary instrumentation, the cost of saving machine states for each instrumentation point is substantial. If the instrumentation points are frequently visited during run time, the cost due to state saving can be significant. Different architectures have different performance tolerance for control interruptions. Newer architectures with deeper pipeline structures will mostly be hurt more by fine-grain instrumentation. These two overheads can be reduced by reducing the number of instrumentation points required to collect the data. We presented two methods in [6] based on static analysis to derive all memory addresses generated by a program from a minimum subset of instrumentation points.

On the other hand, the overhead caused by factors 2 and 3 can also be reduced by removing unnecessary control transfers to the user's instrumentation snippets. Fine-grained instrumentation tasks such as tracing memory can be effectively transformed into much coarser instrumentation. Normally it is not necessary to handle each address as it is captured; asynchronously handling a batch of them once a buffer fills is good enough for most simulation purposes. We describe the details of such an approach in the next section.

## 4 Implementation

### 4.1 Asynchronous Processing

Most trace-driven simulators could collect event traces and process them asynchronously and still be correct. In other words it is not necessary to process each event immediately after it happens and has been captured. Collecting the events is logically independent from the actual processing of traced events. For such asynchronous tracing and processing, the cost of frequent control transfer between instrumented code and user's analysis routines can be reduced significantly.

Most available binary instrumentation tools are generic and allow user analysis routines to respond immediately after each event happens. Such ability has value for time-critical tasks. For example for debugging it can be used to detect errors in the original code and respond immediately. Although synchronized processing is flexible and powerful, such ability comes at a heavy cost for tracing time and is rarely needed in trace-driven simulation. For most of the profiling and tracing tasks used in performance modeling and prediction there is no need to process the event right after it is captured. In fact, such practice should be discouraged due to adverse effects on cache performance. Buffering can reduce the overhead by 40 times [6]. Also such tracing tasks are important for long running applications (so that trace-driven simulation can be driven by realistic

data) which makes it all the more critical to reduce the overhead caused by instrumentation as much as possible.

Since many profiling and tracing tasks are inherently asynchronous, it would be beneficial to create a buffer and let the instrumentation tool decide how to store events to the buffer, how to check buffer status and the like. When buffers are managed by the instrumentation tool, there is no need to transfer control to user routines to write the event into the user's buffer. Many of the instructions related to buffer maintenance can be inlined in the instrumented code and well scheduled to take advantage of the processor's instruction level parallelism mechanisms. Besides the obvious benefit of removing control hazards, fewer instructions are needed to capture the events. In addition, users are saved from the tedious and error prone task of determining where to insert the instrumentation points. ALITER is able to automatically instrument and can also determine what is the best way to collect the traces, just based on the nature of the event the user specifies.

### 4.2 Implementation on POWER4

We have implemented ALITER as a rewriter for assembly code of IBM Power4 processors instead of as a binary rewriter because it is easier to analyze and modify assembly code. Some tedious tasks, such as offset calculation can be done by the assembler. However, the techniques presented in this section can be implemented in the style of a binary rewriter that writes a new instrumented binary, such as ATOM. We illustrate how we can collect memory traces with asynchronous instrumentation in this section. For other types of traces such as call graph determination, a similar method can be used. However the possible benefits of asynchronous instrumentation are more dramatic the finer-grained the trace task.

We first compile the source files to assembly codes, with the same optimization flags one would use to compile to executable directly. The assembly codes are then parsed to break the instructions into basic blocks. The instructions in each basic block are analyzed to determine the necessary instrumentation points. In this example, all the memory instructions are identified to determine how to instrument them. We use the chaining method discussed in [6]. Memory instructions are divided into chains based on their register values. Only one instruction in each chain is selected to be instrumented. In addition, to keep the extra instruction(s) simple, only the register values are stored. The static offset in the instructions are exported to a file and used to re-generate the effective addresses. How the tool instrument the memory is transparent to the user. The user can acquire all the memory instructions one by one using the tool provided interfaces. If in the future, the tool has a different way to organize the buffer, the user doesn't have to modify the analysis routine.

Based on our experiments, almost all basic blocks have at least one GPR (general purpose register) that is not used (read or write) in the block. We can identify that register and use it

to point to the location in the buffer where the values should be stored. Such a register is referred as “regbase” in the following discussion. To store the register values used by one memory instruction, the instrumentation tool simply issues one instruction which writes the register value to a location in the address buffer using regbase right before the original the memory instruction. For instructions using two register to address the memory, two such store instructions could be needed, depending on the outcome of the chaining technique. Regbase is not used by the original instructions so it maintains the pointer to the buffer throughout the block

For each block which is selected to instrument, we also insert a prologue before the first instruction and a postlogue before the last instruction. The prologue is responsible to save the original value of regbase. It also updates the value that indicates how many items have been stored in the buffer and calculates the new start position in the buffer for storing the values for this block. The size of the prologue is about 10 instructions on Power4 systems. To make the instructions for buffer maintenance efficient, more registers are needed. In reality for many basic blocks, there are registers assigned in the block before they are used. The values of such registers are dead when the execution enters the block. We are free to use these registers before they are assigned without having to save their values. It is easier to detect dead register values at the entry than the exit of the block, so most buffer maintenance code is issued in the prologue

In postlogue, the size of the buffer is compared to a threshold value. This threshold value is smaller than the real size of the buffer. The difference should be bigger than the maximum number of instrumentation points of any block in the given application. Function calls are treated as an end to a block. So the buffer is checked only in the postlogue and there is no possibility of buffer overflow. If the stored values are less than the threshold, the postlogue restores regbase to its original value saved in the prologue. Otherwise it jumps to a code sequence which stores all volatile registers and other system status, calls the user provided analysis routine and restores the register values after it returns. This sequence has significant instructions and necessitates a control transfer. However it is visited only when the buffer is nearly full. Currently we set the threshold to 10K items so it is visited infrequently

For the common case in which buffer size is less than the threshold and we can find an unused register for regbase, only ten instructions are required in the prologue and five in the postlogue plus one instruction for each instrumentation point. For example we need less than thirty instructions to collect all the memory traces in the most important of block of CG, which accounts for more than 60% total memory references of the whole program. This is significantly cheaper than making the user maintain the buffer and insert function calls after each instrumentation point. In addition, there are no control interruption in the middle of the block. Because no original instruction ever touch regbase, there is also no data dependency

caused by our inserted instructions. We can take full advantage of the instruction level parallelism mechanisms of Power4. We do have one branch instruction in the postlogue. However the direction of the branch is very friendly to any hardware-based prediction. We actually use extended branch mnemonics to provide a direction hint for Power4’s branch processing unit.

In the rare case that all the GPRS are touched in the block, we need four instructions to store register values to the buffer. These are responsible for saving the value of a temporary register, loading the position of the buffer where a value should be written to, storing the effective address to the buffer and at last restoring the value of the temporary register. In the prologue, the position of the buffer is also calculated using a temporary register. The calculated start point is saved in a pre-defined location so it can be loaded later to store the effective addresses. The postlogue is similar to the common case. Among over a thousand basic blocks in the selected benchmarks, less than ten of them touched all GPRs and require special treatment. In fact those blocks are part of xlc system routines.

For certain memory instructions, the static offset information isn’t available from the assembly code. On Power4, such memory instructions are accessed using register RTOC, a special register which points to the table of contents. It is the linker that finally determines the location of these items in the table and put the correct offset in those instructions. For such memory instructions we can’t export the offset (not known yet). It is necessary to use more instructions to first calculate the effective address and store it to the buffer.

Although ALITER inserts much less instructions into the original code compared to other instrumentation tools, it still can expands code size significantly. On Power4 systems, the distance of a conditional branch is limited by 14-bit offset. Although we use labels in the assembly code and rely the assembler to calculate the exact distances, the assembler could have trouble to encode the distance in the instrumented code into 14 bits. If that happens, we need to switch the branch condition and change the branch target. First we create a new label for the fall through block if it doesn’t have one already. The modified branch is now pointing to the fall through block using its label. A jump instruction is issued between the block branch and its original fall through instructions. This jump branch, which has 24 bits for offset, now points to the original target. ALITER parses the instrumented code a second time after all the instrumented codes have been issued. During the second parse, it calculate the distance for each conditional branch instruction. If the distance is too far, the branch is modified the way we just described.

### 4.3 User’s Responsibility

ALITER users are not required to go through the program structure to determine where to insert snippets so that full event traces can be collected (as for example to determine the min-

imum number of instrumentation points required to capture a complete address stream efficiently). These details are all handled by ALITER. Users are only responsible for providing analysis routines that work on an array of events (such as addresses). Because the event buffer is maintained by the instrumentation tool and analyzed by the users analysis routines, it does require mutual understanding of what is stored in the buffer. Although the users don't have to know the exact layout of the buffer, they do have to know whether the buffer holds path traces or memory address traces. How to expressively enable users to specify interesting events is part of future research.

## 5 Experiments and Results

We selected several benchmarks from the NAS Parallel Benchmark Suite [2] and three binary instrumentation tools to collect their memory traces. All the memory instructions in the binaries are instrumented with a one-line function, which writes the address into fixed location and returns. These experiments were designed to measure the cost of collecting the traces. In these experiments, the traces are collected and discarded immediately after. These experiments gave us a lower bound estimation of slowdown when memory instructions are instrumented. Table 1 list overhead of this experiment using three tools. The ATOM data was collected on Lemieux, an Alpha 21264 system at the Pittsburgh Supercomputer Center. The PIN data was collected on Itanium2 nodes belonging to the Teragrid, and located at the National Center for Supercomputer Applications (NCSA), and the Dyninst data was collected on Cheetah at Oak Ridge National Laboratory. We could not finish tracing LU and SP on Cheetah using Dyninst in five and a half hours, the slowdown is (under) estimated using 5.5 hours.

Application	ATOM	PIN	DyninstAPI
CG.A.4	53.5	86.6	878.6
FT.A.4	35.1	51.8	1003.7
MG.A.4	40.6	57.5	932.5
LU.A.4	53.0	66.5	>>301.4
SPA.4	31.4	47.7	>>203.66

Table 1: Slowdown of Acquiring Addresses Using Different Tools

Application	ALITER	ATOM (delayed)
CG.A.4	1.7	6.4
FT.A.4	1.9	2.0
MG.A.4	1.9	2.5
LU.A.4	1.2	1.6
SPA.4	1.1	2.5

Table 2: Slowdown of Acquiring Addresses Using ALITER and ATOM with delayed instrumentation

We then instrumented the same benchmarks, compiled with

the same flags, using ALITER. This data was also collected on Cheetah. Table 2 shows the slowdown of collecting memory addresses using ALITER. As can be seen, the benefit of inlining buffer management is tremendous. Table 2 also compares the overhead to the best results we measured from ATOM using static analysis and delayed instrumentation techniques described in [6]. With delayed instrumentation, the most visited blocks in the selected benchmarks, except for CG, only require one instrumentation point to store all the necessary register values per block. Considering ATOM is known to generate the better instrumented binary, this slowdown in the second column of Table 2 is arguable the best results we can get from any generic, synchronous binary instrumentation tools. The overhead of ALITER is still better. In addition, users are freed from the tedious and error prone job of static analysis of the instructions in the binary to determine a minimal set of instrumentation points as well as managing the buffer.

## 6 Future Work and Discussion

Sampling is a very important technique to control the overall overhead of tracing. Users can turn on and off tracing based on certain observed events in the traced data. With generic binary instrumentation tools, sampling can be turned on and off immediately when a threshold is hit. Therefore, at first glance, it may look like sampling is a synchronous task. However sampling can also be effectively implemented in an asynchronous scheme at the cost of continuing to gather data for awhile after the trigger event and then discarding the events put into thebuffer in the elapsed interval. Since collecting event traces into the buffer is cheap, coarser grained sampling may still be effective. Another interesting idead is to duplicate the assembly code of each basic block: one instrumented an one kept untouched, and switch between them as suggested in [1, 4] to implement sampling more efficiently.

Working on assembly is a bit of trouble for the user compared to binary instrumentation. First one has to be able to access all the source files. Then the makefile has to be changed to create all the assembly listings. Then all the assembly listings are instrumented (automatically) but then they have to be recompiled. Binary rewriting tools by contrast typically require just one relink. We automate the whole process of modifying the makefiles as long as they are simple and clean. We are planning to implement ALITER as a binary rewriting tool in the future. We also plan to port ALITER to other platforms and ISAs.

## 7 Conclusion

In this paper, we present ALITER, an instrumentation tool and scheme which is targeting tracing tasks that are asynchronous. In this scheme, the instrumentation captures user-specified dynamic events and puts them in a buffer. The control is transferred to the user's analysis routines only when there is a lot of data in the buffer. Such an asynchronous scheme can reduce the number of control transfers between original code

and user's analysis routine dramatically, thus improving the performance of the instrumented code significantly. With this new scheme it is possible to collect memory traces with less than 2 fold slowdown on the selected benchmarks.

## 8 Acknowledgments

This work was supported in part by NSF award CNS-0406312, the DOE Office of Science through the award entitled HPCS Execution Time Evaluation, by NSF NGS Award 0406312: Performance Measurement and Modeling of Deep Hierarchy Systems, and by the Department of Energy Office of Science through SciDAC award High-End Computer System Performance: Science and Engineering. Computer time was partly provided via an NSF NRAC award.

## References

- [1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01*, pages 168–179. ACM Press, 2001.
- [2] 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.
- [3] 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.
- [4] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02*, pages 199–209. ACM Press, 2002.
- [5] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis. In *Supercomputing '02*, 2002.
- [6] X. Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic memory traces. In *to appear in IISWC05*, 2005.
- [7] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *SIGMETRICS '93*, pages 146–157. ACM Press, 1993.
- [8] 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.
- [9] 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.
- [10] A. Snavely, L. Carrington, and N. Wolter. Modeling application performance by convolving machine signatures with application profiles, 2001.
- [11] 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.
- [12] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94*, pages 196–205. ACM Press, 1994.